Saarland University
Faculty of Mathematics and Computer Science

Master's Thesis

Logical Foundations Of Language Interoperability Between OCaml And C

Author
Johannes Hostert

Advisor
Simon Spies

Supervisor
Prof. Dr. Derek Dreyer

Reviewers
Prof. Dr. Derek Dreyer
Dr. Armaël Guéneau

Submitted: 31th July 2023

Saarbrücken, 31th July, 2023

Abstract

Programming Language Theory aims to formally understand different programming languages by exploring the programs that can be written in them, and analyzing which theorems about them can be proven. In the subfield of *Foundational Program Verification*, we often do so by building fully formally verified *program logics*, which aim to formalize and refine the intuitive reasoning programmers use when writing code. This has in recent years been successfully applied to many different languages, leading to a wide variety of program logics for a diverse set of programming languages. These program logics, however, are usually mutually incompatible and limited to programs written entirely in the one language under consideration. This is not sufficient for many real-world programs since these are often build by mixing different programming languages.

In this work, we describe a formal semantics and a program logic for programs written as a combination of OCaml and C, focusing on the interaction between these languages. This allows us to formally explain the existing *Foreign Function Interface* which is used by programmers when bridging between OCaml and C. The resulting theory is language-local: It allows one to verify large parts of a multi-language program using existing single-language formalisms, as if the program just was a single-language one.

This work is largely based on our recent paper [13], co-authored by the author of this thesis. In this thesis, we give a detailed account of the paper. Additionally, we examine of some alternative design choices for the *View Reconciliation* rules. These rules describes how different languages can interoperate on shared mutable state, even when this data is represented differently in these two languages. We further extend the program logic with more powerful reasoning rules, to allow verifying additional programs.

All of our work is mechanized in the Coq proof assistant, using the Iris framework.

Erklärung für Fachfremde

Diese Arbeit ist im Bereich Programmiersprachentheorie angesiedelt. In der Programmiersprachentheorie versuchen wir, verschiedene Programmiersprachen formal zu analysieren. *Formal* bedeutet dabei, dass wir erforschen, welche logisch-mathematischen Aussagen über einzelne Programme, oder die Menge aller Programme, getroffen werden können. Dies erlaubt uns dann, zu beweisen, ob ein Programm auch tatsächlich das Richtige tut, oder ob sich Programmierfehler eingeschlichen haben.

In den letzten Jahren gab es viele wissenschaftlichen Arbeiten, die verschiedene Programmiersprachen auf diese Weise analysiert haben. Allerdings folgen viele dieser Papiere einem ähnlichen Ansatz, welcher voraussetzt, dass das gesamte Programm *nur in einer* Programmiersprache geschrieben ist. In der Praxis schreibt man aber Programme, in dem man mehrere verschiedene Programmiersprachen *kombiniert*. Jede Programmiersprache hat ihre eigenen Stärken und Schwächen, sodass manche Sprachen für gewisse Probleme besser geeignet sind als andere. Damit kann für jedes Problem die für dieses am besten geeignete Sprache verwendet werden. Allerdings entsteht dadurch auch ein Mehr an Komplexität, denn diese Programmiersprachen sind oft recht unterschiedlich. Wenn man diese Sprachen zusammenarbeiten lassen will, muss man beispielsweise Daten zwischen den Formaten der Sprachen hin- und herkonvertieren, was fehleranfällig sein kann.

Wir sehen also zwei Probleme: Bisherige formale Ansätze zur Programmverifikation sind oft nicht auf Programme anwendbar, die in verschiedenen Sprachen geschrieben sind. Dazu kommt, dass durch die Interaktion selbst weitere Fehlerquellen entstehen. Wir präsentieren eine sogenannte *Programmlogik* für die zwei Programmiersprachen OCaml und C, sowie für die Interaktion zwischen diesen beiden Sprachen. Wir präsentieren einen Ansatz, die oben genannten Probleme zu lösen. Wir bauen also ein formales Modell, und beweisen damit Aussagen über Programme in diesen beiden Sprachen. Unser Ansatz ist zwar spezifisch für diese beiden Sprachen, wir hoffen aber, dass er, wenn auch nur in Teilen, auch auf andere Sprachkombinationen angewendet werden kann.

Acknowledgements

To start, thanks to Derek: For offering the semantics course, where I in particular learned Iris, for letting me do research in his group, for grading this thesis, for asking the right high-level questions when we were getting lost in technical details, for exquisite cocktails, and for recommending great baking recipes.

Thanks to Armaël for letting me participate in what was originally your research project, for grading this thesis, and for showing me around Paris.

I thank both of you in advance for working through this thesis, which has unfortunately grown larger than I expected it to.

Thanks to Simon for supervising this thesis, for the countless invaluable advice on the minutiae of academic collaboration, on slide design, on writing, and on doing research in general. Thanks for laughing at my puns, no matter how bad; and for mixing cocktails almost as exquisite as Derek's.

Thanks to the above, and to Lars and Michael for the great collaboration on our paper. In particular, thanks to Michael for helping out on a short notice when things seemed dire.

A huge thanks goes to Benjamin, Florian, Haoyi, Janine, Niklas, and Nils for proofreading my thesis. Further, I thank my friends for helping me distract myself from working too much in the last few months, in particular the aforementioned proofreaders, as well as Ina, Leon, Lisa, Luise, Lukas, Niklas, and Yasmine. I hope that I did not distract you too much in return.

Finally, thanks to my parents: for their support, and for reminding me that I also should explain my research to those without a formal education in Computer Science.

Contents

Chapter 1

Introduction

1.1 Outline and Key Ideas

This thesis introduces Melocoton [13], a program logic for verifying programs written in a combination of OCaml and C. It starts (in Chapter 2) with a by introducing some relevant background, notably Iris [18]. We also assume that the reader has basic knowledge of C and of functional languages like OCaml.

Verifying programs written as a combination OCaml and C is hard because these languages have little in common: They have different values, different memory models, different linkage models, different approaches at unsafety, *etc.*. In real programs, these differences are bridged using the *OCaml-C Foreign Function Interface* (FFI). This FFI introduces *FFI primitives*, which allow writing *glue code*, which can then bridge the differences between these languages. This is presented in Chapter 3.

We then start developing a formal model of this FFI. To start, Chapter 4 defines (as of yet incompatible) operational semantics and program logics for simplified versions of the single languages C and OCaml. Since we consider simplified versions of C and OCaml, we call these $\lambda_{\texttt{C}}$ and $\lambda_{\texttt{ML}}$.

A guiding principle in this work is *language locality*. A key aspect of this is that verification should be carried out in the single language program logics as much as possible. One key aspect of this are reasoning rules for external calls, which allow abstracting over the other language, already introduced in Section 4.1. Further, the resulting operational semantics and program logics should *embed* and *extend* those of the individual languages, so that single-language correctness results carry over.

Chapter 5 describes the operational semantics of the *wrapper*, which is the formal analogue of the OCaml runtime and makes $\lambda_{\texttt{ML}}$ compatible with the $\lambda_{\texttt{C}}$ ABI. It also gives an operational semantics to the FFI primitives. Chapter 6 then builds a *program logic* on top of the wrapper's operational semantics. This program logic embeds the program logics of $\lambda_{\texttt{C}}$ and $\lambda_{\texttt{ML}}$. It then allows reasoning across language barriers, and verifying glue code using FFI primitives.

In the program logic, the key idea are view reconciliation rules, which allow transferring ownership between the different languages. To go beyond the paper, this thesis aims to improve these laws. First, we allow fractional view reconciliation, described in Section 6.4, which already requires a large refactor of the program logic implementation, described in Section 6.3. We then investigate different approaches for implementing these view reconciliation laws, discussed in Chapter 7. Finally, Chapter 8 recaps the relevant background literature, and examines possible future work.

1.2 Authorship Disclaimer

This thesis is in large parts based on the paper "Melocoton: A Program Logic for Verified Interoperability Between OCaml and C" [13], published at OOPSLA 2023.

The paper was written by Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. As mentioned in the paper, the first three authors (which include the author of this thesis) should jointly be considered first authors.

All three authors jointly developed the central definitions in the paper, so that almost no part can be attributed to just one author. Armaël led the project, implemented the operational semantics of Chapter 5, the linker of Appendix A, and introduced many central ideas. Johannes developed the single languages of Chapter 4 including weakest preconditions with external calls Section 4.1, single-language linking of Section 4.1.2, implemented large parts of the program logic (including adequacy and the entire wrapper program logic) of Chapter 6, and verified many of the examples, often in collaboration with and under the supervision of Simon.

Chapters 4 to 6 are based on the mentioned paper, with little novel contributions going beyond what was introduced in the paper (but new and extended explanations). While the paper contains a section similar to Chapter 3, which explains the Foreign Function Interface intuitively, this thesis uses some different examples. We also discuss how our program logic is used to verify these examples in Section 6.2. Chapter 7, as well as Section 6.4 are completely new and not part of the paper.. Additionally, the construction of the program logic presented in Section 6.3 differs from the construction presented in the original paper; this is due to a refactoring done in preparation for Chapter 7.

Chapter 2

Background

This chapter describes our notations and recaps the relevant background on program logics, separation logic, and Iris. These primers are not intended to teach step-indexed separation logics to people without prior knowledge of it. Instead, they merely outline all the foundations needed to fully understand all technical parts of the thesis. For a slower, didactically more worthwhile introduction to these, see [3, 10, 47].

Readers closely familiar with these concepts, in particular with Iris, can skip Sections 2.2 to 2.4, jumping to Section 2.5, which describes how a program can be multi-language.

2.1 Notes on Used Notation

Since we work in a type-theoretic setting, we use some notation borrowed from type theory which might be unfamiliar to a reader used to more classical, set-theoretic mathematics. In particular, we use $\mathbb{P}\mathrm{rop}$ to denote the type of mathematical propositions (*e.g.*, $\exists x.\, x = 5$). In contrast, *iProp* is the type of separation logic/Iris propositions. While we often define propositions cascadingly, we use them as if they were cartesian. In other words, we can define $P : \mathbb{N} \to \mathbb{N} \to \mathbb{P}\mathrm{rop}$, but will still write $P(3, 4)$ instead of $(P\ 3)\ 4$. We also use $x : T$ to denote that x is an object of type T (*e.g.*, $3 : \mathbb{N}$), and use $T : \mathbb{T}\mathrm{ype}$ to denote that T itself is a type. The set-membership symbol $\in$ is reserved for actual sets, which usually are finite (*e.g.*, the domain of a finite map). Somewhat confusingly, the reverse set-membership symbol $T \ni x$ is to be understood as $x : T$, *not* $x \in T$. It is only used in definitions, most prominently in inductive definitions (*e.g.*, $\mathbb{N} \ni n ::= 0 \mid S\, n$). Plain (as opposed to (*co*)*inductive*) definitions are denoted using $\triangleq$.

We use $e[v/x]$ to denote that all free occurrences of x in e are replaced by v. A parallel substitution is denoted using $e[v_1/x_1, \dots, v_n/x_n]$.

Options over a type $T : \mathbb{T}\mathrm{ype}$ are indicated by $\mathsf{option}\, T ::= \mathsf{Some}(t : T) \mid \mathsf{None}$. We sometimes omit Some, treating it as an implicit coercion. Lists $\vec{t} : \vec{T}$ over T are indicated using the vector arrow. Their length is denoted by $|\vec{t}|$ and $t_i : T$ denotes the ith element, where the first element naturally has index 0. We often implicitly generalize relations to lists by lifting pointwisely: For example, if $x \mathrel{\mathcal{R}} y$ is a relation, then $\vec{x} \mathrel{\mathcal{R}} \vec{y}$ means that $\vec{x}$ and $\vec{y}$ have the same length, and that $\forall i.\, x_i \mathrel{\mathcal{R}} y_i$. Finite maps with keys $K : \mathbb{T}\mathit{ype}$ and values $V : \mathbb{T}\mathit{ype}$ are denoted by $\sigma : K \xrightarrow{\mathsf{fin}} V$. Looking up a value is denoted as $\sigma[k]$, which might return None to indicate that the value is absent. Such maps can be updated/extended using the notation $\sigma[k := v]$. Singletons can also be denoted as $\{k := v\}$. The disjoint union $\sigma_1 \mathbin{\dot\cup} \sigma_2$ is the union of σ_1 and σ_2, but is only defined when both maps/sets are disjoint, *i.e.*, $\sigma_1 \cap \sigma_2 = \varnothing$. To be more compact, we also write $\sigma_1 \mathbin{\#\#} \sigma_2$ to express that σ_1 and σ_2 are disjoint. Similar to how we implicitly generalized relations to lists, we also sometimes do so for maps. Specifically, if $x \mathrel{\mathcal{R}} y$ is a relation $X \to Y \to \mathbb{P}\mathit{rop}$, then $\sigma_1 \mathrel{\mathcal{R}} \sigma_2$ (where $\sigma_1 : K \xrightarrow{\mathsf{fin}} X, \sigma_2 : K \xrightarrow{\mathsf{fin}} Y, K : \mathbb{T}\mathit{ype}$) denotes that $\mathsf{dom}\ \sigma_1 = \mathsf{dom}\ \sigma_2$ and that $\forall kxy.\, \sigma_1[k] = x \to \sigma_2[k] = y \to x \mathrel{\mathcal{R}} y$.

2.2 Classical Hoare Logic

In 1969, Tony Hoare [14] introduced a logic–later named Hoare logic–for "proofs of the properties of a program." This logic could be used to prove functional correctness of programs in a simple, imperative language. His logic introduced the *Hoare triple*, consisting of a precondition P, a program s, and a postcondition Q, denoted as follows:

$$\{P\}\, s\, \{Q\}$$

This is a formal statement describing that the program s, when executed on a state initially satisfying P, terminates only in states satisfying Q. Sometimes, Hoare logic is changed to also entail program termination. In this thesis, we do not require this: For a diverging program, every postcondition is valid, since it never terminates, and thus never reaches a state that could violate that postcondition.

In 1975, Edsger W. Dijkstra [8] developed the *weakest precondition* calculus as a different approach for proving formal properties about programs. The weakest precondition calculus is used to compute the weakest precondition $\mathsf{wp}\ s\, \{Q\}$ of a program s for a postcondition Q. This weakest precondition is then implied by all other preconditions, which connects this calculus back to Hoare triples:

$$\{P\}\, s\, \{Q\} \iff P \to \mathsf{wp}\ s\, \{Q\}$$

Further, the weakest precondition is an example of a *predicate transformer*. In general, predicate transformers transform predicates on one kind of object to predicates on another kind of objects. Here, they transform the postcondition, *i.e.*, a predicate on values, to a predicate on expressions encompassing those expressions that evaluate to values satisfying the original predicate. We call logics like Hoare Logic *program* *logics*, since they are designed to reason about program executions. They include several rules/axioms that allow them to verify programs *compositionally*, while being designed to mirror the naive reasoning programmers often employ when reasoning about their programs intuitively.

The languages used by Dijkstra and Hoare to introduce their logics distinguish between statements and expressions: Statements are evaluated only for their side-effects, while expressions denote unique values (here: integers), but never cause side-effects. Their programs manipulated a set of global program variables, with each storing an integer. In Hoare logic, and in weakest precondition reasoning, these program variables are commonly identified with logical variables, *i.e.*, the postcondition $x = y$ denotes that the program variables x and y are defined and have the same value. This causes a variety of issues (*e.g.*, it is cumbersome to express a specification like "`x` remains unchanged throughout the program"). More modern programming languages introduce expressions which have side-effects and can direct control flow. Thus, in this thesis, we will consider "expression languages," where there are no statements, just expressions. To get started, we consider a very simple expression-based language. For now, it does not have state, since we only add state in the next section.

$$\begin{aligned}
\mathsf{Expr} \ni e &::= x : \mathsf{Var} \mid \overline{z} : \mathbb{Z} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid \ldots \\
&\quad \mid \texttt{let } x = e_1 \texttt{ in } e_2 \\
\mathsf{Ctx} \ni K &::= \bullet \mid K + e_1 \mid \overline{z_1} + K \\
&\quad \mid \texttt{let } x = K \texttt{ in } e_2 \mid \ldots
\end{aligned}$$

Operational Semantics Unlike Dijkstra and Hoare in the works above,[1] we start by defining a structural operational small-step semantics. We take this semantics as the "ground truth" definition of how our language computes. This allows us to later show that our program logic is indeed *adequate*, that is, that theorems proven in it reflect the actual computation behavior of the program, as defined by the operational semantics. Formally, such a semantics is a relation $\longrightarrow_{\texttt{simple}}$: $\mathsf{Expr} \to \mathsf{Expr} \to \mathbb{P}\mathrm{rop}$. We read $e_1 \longrightarrow_{\texttt{simple}} e_2$ as "e_1 steps to e_2." The semantics is then typically defined as an inductive relation, like in Figure 2.1. This figure shows two rules, for two primitive language constructs, namely SAdd for $+$ and SLetIn for `let-in`. Additionally, there is the rule Ctx, which allows reduction in sub-expressions.

Our operational semantics distinguish regular steps ($\longrightarrow_{\texttt{simple}}$) and head steps ($\longrightarrow_{\texttt{hsimple}}$): Head steps require the expression under reduction to not be nested below some operators. Regular steps are then defined by allowing head steps under

[1] Both were well aware of operational semantics, and that one should prove that all semantics are in correspondence.

$$\textsc{SAdd}\quad \frac{}{\overline{z_1} + \overline{z_2} \longrightarrow_{\texttt{hsimple}} \overline{z_1 + z_2}} \qquad \textsc{SLetIn}\quad \frac{}{\texttt{let } x = \overline{z} \texttt{ in } e \longrightarrow_{\texttt{hsimple}} e[\overline{z}/x]}$$

$$\textsc{Ctx}\quad \frac{e_1 \longrightarrow_{\texttt{hsimple}} e_2}{K[e_1] \longrightarrow_{\texttt{simple}} K[e_2]}$$

Figure 2.1: Operational Semantics for a simple language.

an arbitrary *evaluation context* K, using the rule Ctx. Such contexts are best understood as expressions with a hole. The operation $K[e]$ substitutes the expression e into the context K, replacing the hole $\bullet$ with e. Contexts define which subexpressions is reduced first, and in what order. Above, we defined contexts such that the left operand of an addition is evaluated first, by requiring that the right operand is only evaluated when the left operand has fully reduced to a value (here: an integer). Further, the `let-in` construct only allows reducing the `let` part, which must be substituted using SLetIn before the other subexpression can be evaluated. The expression that is evaluated next (to which a head reduction step applies) is called the *head redex*. Since addition (and all other binary operators omitted here) reduces the left side first, we call such languages *left-to-right*. In future, we omit the machinery for lifting head steps to regular steps, by simply indicating which expressions are evaluated first, by *e.g.*, specifying left-to-right evaluation order.

There are several terms in our language which do not reduce, *i.e.*, which block reduction when being head redex. An example is the expression x which denotes a variable: When it is head (redex) position, then the original program was malformed since it contained a free variable. We say that such a program is *stuck*. Another expression that does not reduce further is the integer constant expression $\overline{z}$. However, this expression (and only this expression) is not considered stuck. Instead, we consider a program that has reduced to an integer to have *terminated successfully*. Later, we generalize integers to the larger class of values. These are the "objects" our program language works with, by *e.g.*, storing them in variables. We consider a program to have terminated when it has reduced to a value, which is then the result/return value. These expressions that are either not stuck, or indicate termination (are values), are called *safe*. A stuck program is said to have undefined behavior, following the way undefined behavior is commonly understood in C. A program with undefined behavior is considered erroneous, but this error is silent: The compiler / execution environment is allowed to behave arbitrarily. This is in contrast to "safe" failure (like raising an exception), which is usually part of the definition of the operational semantics. This motivates the following definition of safe expressions, which do not cause undefined behavior:

$$\mathsf{safe} : \mathsf{Expr} \to (\mathbb{Z} \to \mathbb{P}\mathrm{rop}) \to \mathbb{P}\mathrm{rop}$$

$$\mathsf{safe}(e, Q) \triangleq \begin{cases} Q(z) & e = \overline{z} \\ \exists e', e \longrightarrow_{\texttt{simple}} e' & \text{otherwise} \end{cases}$$

Our definition of safe is further refined by a postcondition Q, which can be used to exclude undesired termination values. If we take $Q(z) \triangleq \top$, then we recapture the intuition from above that safe expressions are all those that are either an integer, or that step to another expression.

Program Logic We can now define a program logic for this language. We do so by (axiomatically) defining the weakest precondition. Since we switched from a statement-based language to an expression language, our weakest precondition changes: Instead of taking a postcondition Q defined on the state, we take a post-condition $Q : \mathbb{Z} \to \mathbb{P}\mathrm{rop}$, which describes a condition on the value an expression evaluates to. Further, a Hoare triple now still takes a regular precondition (of type $\mathbb{P}\mathrm{rop}$), but uses the same postconditions as the weakest precondition. Thus, Hoare triples can *be defined* using the weakest precondition:

$$\{P\}\, s\, \{Q\} \iff P \to \mathsf{wp}\; s\, \{Q\}$$

Further, a Hoare triple like $\{z > 0\}\overline{z} + \overline{z}\{\lambda r.\, r = z\}$ is to be read as $\forall z.\, z > 0 \to \mathsf{wp}\; \overline{z} + \overline{z}\, \{\lambda r.\, r = z\}$. In other words, we implicitly $\forall$-quantify over all free variables in it. Finally, we often write postconditions occurring in a weakest precondition (or a Hoare triple) as $\mathsf{wp}\; e\, \{r.\, Q(r)\}$ instead of $\mathsf{wp}\; e\, \{\lambda r.\, Q(r)\}$. We do not give a formal definition of the weakest precondition, delaying this to Section 2.4 and instead leave this axiomatic for now. The axioms used to define the weakest precondition can be found in Figure 2.2. The first rule (WP-Value) describes the base case of the weakest precondition: A program that has already evaluated to a value. This value must then satisfy the postcondition Q. The next two rules describe how addition and `let-in` reduce, assuming they are the head redex. These are actually just special cases of the more general rule WP-Step for arbitrary steps, as long as these steps are unique. The *bind rule* (WP-Bind) is used to reason about expressions occurring at the head redex position in another term: It describes that we can first reason about the inner expression, and then continue reasoning about the results of it, placed in the context of the inner expression. The last rule (WP-Weak) allows weakening the postcondition. This presentation of the weakest precondition, with emphasis on the bind rule, closely follows Timany *et al.* [3].

Note that although we call $\mathsf{wp}\; e\, \{Q\}$ the *weakest* precondition, we do not show that the axioms given in Figure 2.2 are complete, and fully expect them not to (especially

$$\textsc{WP-Value}\ \frac{Q(z)}{\mathsf{wp}\ \overline{z}\ \{Q\}} \qquad \textsc{WP-Add}\ \frac{\mathsf{wp}\ \overline{z_1 + z_2}\ \{Q\}}{\mathsf{wp}\ \overline{z_1} + \overline{z_2}\ \{Q\}} \qquad \textsc{WP-LetIn}\ \frac{\mathsf{wp}\ e[\overline{z}/x]\ \{Q\}}{\mathsf{wp}\ \texttt{let}\ x = \overline{z}\ \texttt{in}\ e\ \{Q\}} \qquad \textsc{WP-Bind}\ \frac{\mathsf{wp}\ e\ \{\lambda z.\ \mathsf{wp}\ K[\overline{z}]\ \{Q\}\}}{\mathsf{wp}\ K[e]\ \{Q\}}$$

$$\textsc{WP-Weak}\ \frac{\mathsf{wp}\ e\ \{Q\} \qquad \forall z.\ Q(z) \to R(z)}{\mathsf{wp}\ e\ \{R\}} \qquad \textsc{WP-Step}\ \frac{\mathsf{wp}\ e_2\ \{Q\} \qquad e_1 \longrightarrow_{\texttt{simple}} e_2}{\mathsf{wp}\ e_1\ \{Q\}}$$

Figure 2.2: Weakest Precondition axioms for a simple language.

$\lvert\{e' \mid e \longrightarrow e'\}\rvert$	Intended meaning
0	Undefined behavior / being stuck
1	One unique successor value[2]
$\geqslant 1$	Demonic choice out of all possible e'

Table 2.1: Demonic choice and *undefined behavior* for normal languages.

in the following sections). In fact, our weakest precondition is better understood as one rather weak precondition, which is in practice weak enough to establish most program specifications one hopes to prove. While our weakest precondition is not complete, it is sound: If we prove $\mathsf{wp}\ e\ \{Q\}$, then our program will only reduce to values satisfying Q, or it will diverge by reducing forever. In particular, our program never encounters undefined behavior. Formally, this can be argued by proving that all finite prefixes of execution traces are *safe*:

$$\mathsf{wp}\ e\ \{Q\} \quad \Longrightarrow \quad \forall e'.\ e \longrightarrow^*_{\texttt{simple}} e' \to \mathsf{safe}(e', Q)$$

This uses the reflexive-transitive closure $\longrightarrow^*_{\texttt{simple}}$ of $\longrightarrow_{\texttt{simple}}$, which expresses that we take a number $n \geqslant 0$ of reduction steps. After taking these steps, our partially reduced program must be safe, *i.e.*, must have completely reduced to a value, or must be able to take another step (which is then again to a safe value).

Demonic Non-Determinism In our simple language, the relation $\longrightarrow_{\texttt{simple}}$ is (partially) functional: Every expression reduces to at most one successor expression. Later on, we introduce languages where one program state can have several successor states. These will then indicate a *demonic* choice: In order to be correct, the program needs to handle *all* possible successor states. In terms of weakest preconditions, this means that if we want to show that $\mathsf{wp}\ e\ \{Q\}$, we need to prove $\mathsf{wp}\ e'\ \{Q\}$ for all e' such that $e \longrightarrow e'$. The simple step rule WP-Step, which only considers a single

[2]Note that this is a special case of a demonic choice–if there is only one value to choose, the demon has no real choice.

step, is then no longer applicable. Table 2.1 summarizes the intended meaning of having none, a unique, or several successors under an arbitrary step relation $\longrightarrow$.

2.3 Separation Logic

Separation Logic [38] was introduced to address several weaknesses with the existing (Hoare-style) logics for languages with mutable state. Our simple language of Section 2.2 had no state. In contrast, the languages originally considered by Hoare and Dijkstra had a simple state consisting of the program variables modified by these programs. Unfortunately, the program logics originally presented by them are not well-suited for reasoning about programs using shared mutable state, in particular not when one whishes to do so modularly. Consider, for example, the following two programs which access global variables x, y, and t. The function foo is better understood as a template, *i.e.*, the variables x and y are passed by reference.

```
foo(){                     swap(x, y){
  t = 1; r = 2;              t = x; x = y; x = t;
  swap(t, r);              }
}
```

While the classical Hoare triple $\{x = 1 \wedge y = 2\}\mathrm{swap}(x, y)\{x = 2 \wedge y = 1\}$ is provable, executing foo() does *not* result in a state where $r = 1$. This is because swap internally re-uses the variable t, conflicting with foo, which already uses it. Notably, the fact that swap uses t does not show up in the specification of swap. One approach of solving this is by explicitly stating which variables are modified by which program. When we then use an already-verified subroutine, we need to check the variables modified by it against the variables our program itself is using. This leads to a tedious amount of book-keeping. Instead, we use *separation logic*, which makes this tracking implicit. If we designed a separation logic-based program logic for a language like the one above, we would not be able to establish the shown specification for swap. Instead, the specification would need to explicitly mention t. We do not further discuss how a separation logic-based program logic for the above language would look like, instead focusing on developing one for the simple language from the last section. First, we will add mutable state to that language:

$$\begin{aligned}
\mathit{Val} \ni v &::= z : \mathbb{Z} \mid \ell : \mathrm{Loc} \\
\mathit{Expr} \ni e &::= x : \mathit{Var} \mid \overline{v} \\
&\quad \mid \texttt{alloc}() \mid !e \mid e_1 := e_2 \mid \cdots \\
\Sigma &\triangleq \mathrm{Loc} \xrightarrow{\mathrm{fin}} \mathit{Val}
\end{aligned}$$

We added three new expressions, for freshly allocating, loading from, and storing to a memory cell. Additionally, we have introduced the syntactic category of values,

$$\textsc{SAlloc}\quad \frac{\ell \notin \mathsf{dom}\ \sigma}{(\mathsf{alloc}(), \sigma) \longrightarrow_{\mathtt{hsimple}} (\overline{\ell}, \sigma[\ell := 0])} \qquad \textsc{SLoad}\quad \frac{\sigma[\ell] = v}{(!\ell, \sigma) \longrightarrow_{\mathtt{hsimple}} (\overline{v}, \sigma)}$$

$$\textsc{SStore}\quad \frac{\ell \in \mathsf{dom}\ \sigma}{(\ell := \overline{v}, \sigma) \longrightarrow_{\mathtt{hsimple}} (\overline{0}, \sigma[\ell := v])} \qquad \textsc{CtxState}\quad \frac{(\sigma_1, e_1) \longrightarrow_{\mathtt{hsimple}} (e_2, \sigma_2)}{(\sigma_1, \mathsf{K}[e_1]) \longrightarrow_{\mathtt{simple}} (\mathsf{K}[e_2], \sigma_2)}$$

Figure 2.3: Operational Semantics for a simple language with state.

which can now be either integers or location values. A location value ℓ is also encoded as an integer, but is in a distinct category, since it can not be used for arithmetics (while plain integers are not to be used for loading from a memory cell).[3] The operational semantics now operate on pairs $\mathsf{Expr} \times \Sigma$ of the current state and the current expression to be executed. We also call these configurations. Most rules in the operational semantics leave the state unchanged. The others are shown in Figure 2.3. The rule SAlloc describes allocation. It is non-deterministic, and evaluates to a new location (which is initialized to 0). SLoad describes loads, and SStore describes stores. The latter evaluates to 0, and we usually ignore the value, since it is only executed for its side-effects. The CtxState rule replaces the Ctx rule, to inherit the state change from the head redex to the total reduction step. Finally, we change our notion of safety to configurations:

Definition 2.1 (Safety for a stateful language)

$$safe : \Sigma \to \mathsf{Expr} \to (\mathsf{Val} \to \mathbb{P}\mathsf{rop}) \to \mathbb{P}\mathsf{rop}$$

$$safe(\sigma, e, Q) \triangleq \begin{cases} Q(v) & e = \overline{v} \\ \exists e'\ \sigma'.\ (e, \sigma) \longrightarrow_{\mathtt{simple}} (e', \sigma') & \textit{otherwise} \end{cases}$$

Plain Separation Logic The key idea of separation logic is that it is a logic for resources. Whereas plain mathematical logic deals with mathematical facts, which remain true once established, separation logic deals with resources, which can be created, updated, and destroyed. What these resources crucially lack is the ability to be duplicated: In general, resources can not be duplicated arbitrarily, but can only be created, changed, or destroyed under very specific circumstances. In our case, these resources will express ownership of parts of the heap. A formula of separation logic, denoted using $P : iProp$, expresses that some property is true on some part

[3]This distinction is erased by compilation to assembly, but it makes reasoning much nicer, so we keep it in the source semantics.

of the heap. Additionally, the formula will carry *ownership* of this part of the heap. Other formulas that are true *separately* are not able to describe facts about this part of the heap.

The formal semantics of separation logic are defined against a particular heap. Formally, we say that a formula $P : iProp$ of separation logic is satisfied by a heap fragment σ, denoted $\sigma \models P$.

$$\begin{aligned}
\sigma \models Ppure &\Leftrightarrow Ppure && Ppure : \mathbb{P}\mathsf{rop}\\
\sigma \models P * Q &\Leftrightarrow \exists \sigma_1 \sigma_2 .\, \sigma_1 \dot{\cup} \sigma_2 = \sigma \wedge \sigma_1 \models P \wedge \sigma_2 \models Q\\
\sigma \models P \mathrel{-\!\!*} Q &\Leftrightarrow \forall \sigma_a .\, \sigma_a \models P \wedge \sigma_a \mathrel{\#\#} \sigma \implies (\sigma_a \cup \sigma) \models Q\\
\sigma \models \ell \mapsto v &\Leftrightarrow \{\ell := v\} \subseteq \sigma\\
\sigma \models P \wedge Q &\Leftrightarrow \sigma \models P \wedge \sigma \models Q\\
\sigma \models P \vee Q &\Leftrightarrow \sigma \models P \vee \sigma \models Q\\
&\vdots
\end{aligned}$$

Besides inheriting the existing logical connectives, and a trivial embedding of regular propositions $\mathbb{P}\mathsf{rop}$, separation logic introduces several new ones. The first one is the separating conjunction $*$, which denotes that P and Q hold on disjoint parts of the heap (remember that $\dot{\cup}$ is a disjoint union). The next one, the magic wand $\mathrel{-\!\!*}$, is the right-adjoint of $*$–in other words, we have $P \mathrel{-\!\!*} Q \mathrel{-\!\!*} R \Leftrightarrow P * Q \mathrel{-\!\!*} R$. Finally, we have the points-to connective $\ell \mapsto v$, which describes that v is stored at location ℓ, and that ℓ is part of the heap that the formula expresses ownership of. To understand the difference between regular and separating conjunction, consider the formula $\ell_1 \mapsto 1 \wedge \ell_2 \mapsto 1$. This formula expresses that both ℓ_1 and ℓ_2 are active locations in the heap, both storing 1. However, these locations might alias, i.e. $\ell_1 = \ell_2$ might be true. In contrast, the separating conjunction $\ell_1 \mapsto 1 * \ell_2 \mapsto 1$ allows us to derive that $\ell_1 \neq \ell_2$, since the heap must be decomposable in disjoint parts, each validating one of the points-tos. Note that we use $\vdash P$ as syntactic sugar for $\forall \sigma, \sigma \models P$. While $\models P$ might be more appropriate, we do so to follow Iris' syntax.

A Program Logic Using Separation Logic We now show how separation logic is used for building program logics. For this, we define WP not as a regular proposition, but as a formula of separation logic. Similarly, the rules of Figure 2.4 are to be understood as separation logic rules (using magic wands and combining the preconditions using separating conjunctions), which is indicated by using a magic wand $\mathrel{-\!\!*}$ as the separating line. Note that this figure also generalizes the postcondition to act on values $v : \mathsf{Val}$, not just integers. Figure 2.4 shows the new rules we gain from using separation logic. Additionally, all rules of Figure 2.2, except for WP-Weak, continue to apply (when reinterpreted as separation logic rules using magic wands). The rule WP-Alloc is used to verify an allocation. It allows us

$$\textsc{WP-Alloc}\quad \frac{\ell \mapsto 0 \mathrel{-\!\!*} \mathsf{wp}\,\overline{\ell}\,\{Q\}}{\mathsf{wp}\,\mathtt{alloc}()\,\{Q\}}\,{*} \qquad \textsc{WP-Load}\quad \frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}\,\overline{v}\,\{Q\}}{\mathsf{wp}\,!\overline{\ell}\,\{Q\}}\,{*} \qquad \textsc{WP-Store}\quad \frac{\ell \mapsto v \qquad \ell \mapsto v' \mathrel{-\!\!*} \mathsf{wp}\,\overline{0}\,\{Q\}}{\mathsf{wp}\,\overline{\ell} := \overline{v'}\,\{Q\}}\,{*}$$

$$\textsc{WP-Wand}\quad \frac{\mathsf{wp}\,e\,\{Q\} \qquad \forall z.\, Q(z) \mathrel{-\!\!*} R(z)}{\mathsf{wp}\,e\,\{R\}}\,{*}$$

Figure 2.4: Weakest Precondition axioms for a simple language with state.

to continue verifying with the remaining program, while additionally assuming ownership of a new, *separate* piece of the heap containing our allocated cell. This in particular uses the fact that allocation is defined (operationally) to always return a previously unused, fresh location. Further, the rule WP-Store is used to verify a store. To apply it, one needs to isolate the part of the heap on which $\ell \mapsto v$ holds, where ℓ is the location to be modified. Thus, all other parts of the heap, and all other statements expressing facts about them remain unchanged, since they are now separate from the piece of the heap undergoing modification. The rule states that we need to "give up" this piece of heap, by handing it to the rule. After this, we no longer assert anything about this piece of heap. The rule then gives $\ell \mapsto v'$ back to us, which now reflects the changes made by the store. The rule WP-Load can be understood similarly, except that it does not change the points-to. Finally, we have the rule WP-Wand, which replaces WP-Weak. This rule allows us to weaken postconditions, as before. However, it can also be used to prove the *frame rule*:

$$P * \mathsf{wp}\,e\,\{v.\, Q(v)\} \mathrel{-\!\!*} \mathsf{wp}\,e\,\{v.\, P * Q(v)\}$$

This rule formalizes the previous intuition that resources not used when proving a weakest precondition are not affected by program execution–they still hold once the program has finished.

Adequacy Our adequacy statement for this new program logic now also needs to refer to the heap. We need a proof that the weakest precondition is satisfied on some initial heap σ, and can then derive that the postcondition will be satisfied by the heap σ' present at termination.

Theorem 2.2 (Adequacy of the program logic using separation logic)

$$\sigma \models \mathsf{wp}\,e\,\{Q\} \quad\Longrightarrow\quad \forall e'\sigma'.\,(e'\sigma) \longrightarrow^*_{\mathtt{simple}} (e',\sigma') \rightarrow \mathit{safe}(\sigma', e', \lambda v.\, \sigma' \models Q(v))$$

The proof requires the definition of the weakest precondition, which we do not discuss here. Since safe is monotone in Q, this core adequacy theorem can be used

```
let l = alloc() in
let _ = l := ‾42‾ in
let r = !l in
r
```

```
{⊤}
  let l = alloc() in
{l = ℓ ∗ ℓ ↦ 0}                 l = ℓ
  let _ = l := ‾42‾ in
{l = ℓ ∗ ℓ ↦ 42}
  let r = !l in
{r = 42 ∗ ℓ ↦ 42}               r = 42
  r
{r. r = 42 ∗ ℓ ↦ 42}
```

Figure 2.5: A simple program (left) with a corresponding Hoare outline (right).

to derive a more useful one, where the postcondition of *safe* is merely implied by $\sigma' \models Q(v)$.

Hoare Outlines Correctness proofs in separation logic can be rendered using Hoare outlines. A Hoare outline shows the resources collected at each step of the program. As an example, we will verify that the program on the left side of Figure 2.5, starting in the empty heap, terminates only with 42. In a Hoare outline (shown on the right in Figure 2.5), we re-use Hoare triple notation $\{P\}\, s\, \{Q\}$ to keep track of the resources as they are modified by the program. Each line in {brackets} denotes the resources, the others denote expressions in our program. While the active resources are repeated each time (to indicate that they are still available and not changed), pure facts (like $r = 42$) are not, since they will remain true forever. To indicate that something is pure, we sometimes write it on the right, next to the outline. This is supposed to indicate that this fact remains true as we go further down in the program.

After convincing us that the original Hoare triple holds after constructing this Hoare outline, and using that $\models \top$, we can use the adequacy theorem to infer that our original program continuously reduces to new expressions, and only terminates with a value v in a state $\sigma' \models v = 42 \ast \ell \mapsto 42$, *i.e.*, with the value 42.

2.4 Program Logics in Iris

Separation logic has been very successful, with many generalizations to it developed in the years after Reynolds' original paper [38]. We do not aim to recount the complete history here (see O'Hearn's summary [32] for a comprehensive review), but instead focus on *Iris* [19, 22, 18], a *step-indexed* separation logic featuring *higher-order, custom ghost state*, that also is *fully formalized in Coq*. The remainder of this thesis uses Iris as the separation logic. Iris also includes support for concurrency, which is not needed in this thesis.

In the separation logic introduced previously, the only true separation logic primitive was the points-to. Iris instead allows one to define one's own separation logic primitives, by defining an algebraic structure called *OFE* (an ordered family of equivalences, see in particular [18]). Instead of defining the model against heaps, the model is then defined against an OFE. Since these resources usually do not directly describe the actual program state, they are called ghost state. Finite maps and points-tos are then just one particular example of ghost state, which are later linked to the actual heap of the operational semantics. An element a of an OFE can be turned into a separation logic proposition $\boxed{a}^{\gamma}$ expressing that a is *owned*. Since one often works with several OFEs, a ghost name γ is used to denote which particular instance is meant.

Additionally, Iris is step-indexed: The model is defined not just against such OFEs, but also against a step-index. This step-index can be used to define several interesting separation logic predicates that would not be admissible otherwise (due to essentially unguarded recursion), by defining them by recursion on the step-index (see [47] for examples). Notably, the weakest precondition itself can be defined in Iris, by decreasing the step-index by one for each reduction step, thereby linking the step-index to program steps. For readers not familiar with step-indexing, this can be ignored, since most of our program logic does not use step-indexing. To reason about these features, Iris includes a number of *modalities*: The *later modality* $\triangleright P$ expresses that P is valid at the next lower step-index. Since properties are downwardly closed, this modality is easy to prove, but typically hard to eliminate. The *persistance modality* $\boxdot P$ turns P into a persistent resource. A persistent resource (of which $\boxdot P$ is just one example) can be freely duplicated. Since it is freely duplicable, $\boxdot P$ is harder to prove: One can only use other persistent resources in the proof. Intuitively, a persistent resource is a fixed, unchanging property about some part of the heap, and so it can be "remembered" even when only reasoning about other parts of the heap.

To reason about custom resources, Iris includes the *ownership primitive* $\boxed{a}^{\gamma}$, expressing that a, a member of an OFE, is currently owned, *i.e.*, valid for the current resources. To allow having several different kinds of resources, and several instances of the same resource, a ghost name γ denotes which instance of which OFE we are referring to. Finally, the *update modality* $\dot{\Rrightarrow} P$ expresses that P is true, after updating some of the ghost state. It is often used together with the wand, so that there is the syntactic sugar $\Rrightarrow\!\!\ast \triangleq \mathrel{-\!\!\ast} \dot{\Rrightarrow}$.

Some Useful Ghost State As mentioned, Iris uses the generic ghost state mechanism to define the ghost theory of heaps and points-tos. A points-to $\ell \mapsto_{\gamma} v$ (now including a ghost name γ) works like before. To ground this points-to, Iris includes an *authoritative* counterpart $\boxed{\bullet\sigma}^{\gamma}$, which denotes the full heap. When taken together, they can be updated, to *e.g.*, implement a store rule:

$$\boxed{\bullet\sigma}^{\gamma} * \ell \mapsto_{\gamma} v \Rrightarrow\!\!* \boxed{\bullet\sigma[\ell := v']}^{\gamma} * \ell \mapsto_{\gamma} v'$$

These are just a particular example of the general authoritative ghost state construction, which allows having an authoritative part $\bullet a$, and several fragments $\circ f$, with an arbitrary relation $a \; \mathcal{R} \; f$ that must hold between the authoritative part and the fragment. Above, points-tos are just syntactic sugar for a particular fragment. Iris further generalizes the points-to to be fractional. A fractional points-to $\ell \mapsto^{q} v$ where $0 < q \leqslant 1$ is said to be fully owned when $q = 1$, in which case it behaves like the regular points-to (and we may omit the fraction). Points-tos can be combined and split, so that their fractionalities combine. A partially owned points-to (with $q < 1$) can not be used for modifications, since it is not exclusive and so other fractional points-tos for the same location might be used elsewhere. There also is the persistent points-to $\ell \mapsto^{\Box} v$, which is persistent. This also ensures that the points-to is never mutated again. In Iris, fractional and persistent points-tos are unified using *discardable fractions* $\mathbb{Q}_{\boxdot} \ni d ::= (q : \mathbb{Q}) \mid \Box$.

Another useful construction are *ghost variables*. Using them, a ghost name γ can be assigned to a value $x : X$ (for any type X), denoted as $\gamma \overset{\text{👻}}{\mapsto} x$. What makes them great is that they can split fractionally, so that there can exists two parts $\gamma \overset{\text{👻}}{\mapsto}_{1/2} x * \gamma \overset{\text{👻}}{\mapsto}_{1/2} y$, which must then be in *agreement*: It must hold that $x = y$. Similar to points-tos, the ghost variable can only be changed when it is fully owned. Yet another useful feature of Iris is that of *invariants*. An invariant $\boxed{P}^{\mathcal{E}}$ is a "persistent box" around a proposition P. We say that the ownership is stored in the invariant. This ownership can only temporarily be taken out of the invariant. To track which invariants are currently open, and to force these invariants to close again, a mask $\mathcal{E}$ is used. For technical reasons, we do not get ownership of P, but only of $\triangleright P$. Thus, we often put *timeless* propositions into invariants, which are propositions that do not depend on the step-index, allowing us to remove this later sometimes. In regular Iris, this mask is tracked in the weakest precondition, [4] and ensures that we do not open invariants twice. Regular Iris also forces these invariants to only be opened around atomic propositions, like compare-and-swap. Thereby, these invariants can safely be shared between threads, since no thread can violate the invariant while another one is accessing it. In this thesis, we do not use the concurrency support of Iris. Thus, we use non-atomic invariants, which can remain open longer. To express that they should be closed again, there is the separation logic token $[\mathsf{NaInv} : \mathcal{E}]$, which tracks

[4]It is actually tracked by mask-changing updates, which the weakest precondition uses prominently.

the open invariants. To force invariants to close again, one includes the obligation $[\mathsf{NaInv} : \top]$, which asserts that all invariants are closed.

Defining The Weakest Precondition Iris' logic is powerful enough to define the weakest precondition $\mathsf{wp}\ e\,\{Q\}$ within the logic itself.

Definition 2.3 (Weakest Precondition in Iris)

$$
\begin{aligned}
\mathrm{reducible}(e, \sigma) &\triangleq \exists e'\,\sigma'.\,(e, \sigma) \to (e', \sigma') \\
\mathrm{SI}(\sigma) &\triangleq \boxed{\bullet\sigma}^{\gamma} \\
\mathsf{wp}\ e\,\{Q\} &\triangleq \begin{cases}
\dot{\Rrightarrow} Q(v) & e = \overline{v} \\
\forall \sigma.\, \mathrm{SI}(\sigma) \Rrightarrow\!\!\ast & \\
\quad \mathrm{reducible}(e, \sigma) & \textit{otw.} \\
\quad \ast\, \forall e', \sigma'.\,(e, \sigma) \longrightarrow_{\texttt{simple}} (e', \sigma') \mathrel{-\!\!\ast} & \\
\qquad \dot{\Rrightarrow} \triangleright \dot{\Rrightarrow} \mathrm{SI}(\sigma') \ast \mathsf{wp}\ e'\,\{Q\} &
\end{cases}
\end{aligned}
$$

Note that the actual definitions uses fancy updates / view shifts. Since we do not need them in our development, we elide them.

This definition is quite hard to parse, especially when one is not used to Iris (see [18] for a full explanation). The basic idea is that it is defined using the operational semantics, by encoding the definition of safety. To prove $\mathsf{wp}\ e\,\{Q\}$, if e is a value, is very simple: Just prove $Q(v)$, potentially changing the ghost state. The other case is more complicated. It first requires the notion of a *state interpretation* $\mathrm{SI} : \Sigma \to \mathit{iProp}$, which defines how the physical state is related to the ghost state. In our case, it carries an authoritative part of the heap+points-to ghost theory, of which the points-tos are the fragment parts. Then, the $\mathsf{wp}\ e\,\{Q\}$ in the interesting case allows us to assume the state interpretation for our current state. We then need to prove two things. The first property is that we are not stuck, *i.e.*, that there is a successor we can reduce to. Then, we handle demonic non-determinism, by assuming an arbitrary successor (σ', e') of our current configuration (σ, e). Then, we must prove that *later* (*i.e.*, we are allowed to decrease the step-index), we can update our state interpretation to σ' and also need to prove $\mathsf{wp}\ e'\,\{Q\}$.

This definition is well-formed by recursion on the step-index, which decreases each step. The base case at the step-index 0, which is always eventually reached by diverging programs, is just that the program is reducible, which justifies that the weakest precondition of diverging programs is always $\top$, too. Using this definition, one can derive the rules shown in Figures 2.2 and 2.4. To get a useful result, we again need an adequacy theorem, which now looks as follows:

Theorem 2.4 (Adequacy of Iris) *Let σ be a state such that $\vdash \mathsf{SI}(\sigma)$, and $Q' : \Sigma \to \mathsf{Expr} \to \mathbb{P}\mathrm{rop}$ such that $\forall \sigma' e'.\, \mathsf{SI}(\sigma') * \mathsf{safe}(\sigma', e', Q) \Rrightarrow\!\!* \; Q'(\sigma', e')$. Then the following holds:*

$$\vdash \mathsf{wp}\, e\, \{Q\} \implies \forall e'\, \sigma'.\, (e, \sigma) \longrightarrow^{*}_{\texttt{simple}} (e', \sigma') \implies Q'(\sigma', e')$$

Iris includes all of this development for a *generic* language, as long as the language fits into the general recipe. Such a language is defined by:

- A type of values, Val
- A type of expressions, Expr
- An injection Val $\to$ Expr, along with a partial inverse
- A type of state, Σ
- A head step relation $\to$: $\mathsf{Expr} \times \Sigma \to_{\mathtt{h}} \mathsf{Expr} \times \Sigma \to \mathbb{P}\mathrm{rop}$
- A type of evaluation contexts Ctx
- Composition and instantiation operations for contexts.
- A state interpretation $\mathsf{SI} : \Sigma \to$ *iProp*

Additionally, this structure must satisfy some laws, like that composition and instantiation behave well, that values do not step, or that head redexes are unique or contain only values.

2.4.1 Transfinite Iris

For reasons explained in Section 5.1, we have to use *Transfinite Iris* [45]. Transfinite Iris is a version of Iris where the step index is not just an integer, but instead an ordinal number. The main motivation for Transfinite Iris is the so-called existential property. This property allows extracting the witness of an existential quantifier proven in Iris' separation logic. Concretely, if we have $\vdash \exists x.\, \Phi(x)$, we want to move this existential quantifier to the meta-logic, that is, conclude $\exists x.\, \vdash \Phi(x)$. In ordinary Iris, this is not possible, since the witness within separation logic can be chosen depending on the step-index. Formally, we would have to turn a $\forall\exists$ into an $\exists\forall$. In Transfinite Iris, this is possible by choosing a large enough ordinal as the step-index, and using the *Axiom of Choice* to choose a witness valid at every step-index at once. Unfortunately, using a transfinite step-index breaks some of the rules known from regular Iris. Those two rules, shown below, allow commuting the $\triangleright$ modality and the existential quantifier / the separating conjunction:

$$\textsc{LaterExists}\quad \frac{X \text{ inhabited}}{\triangleright(\exists x : X.\, \Phi(x)) \vdash \exists x : X.\, \triangleright \Phi(x)} \qquad \textsc{LaterSep}\quad \frac{}{\triangleright(P * Q) \vdash (\triangleright P) * (\triangleright Q)}$$

Thus, all proofs in this thesis have to be carried out without using these rules. In particular, avoiding LaterSep makes some definitions a bit more awkward, for example in Section 4.5.

2.5 What Is A Multi-Language Program?

In many programming languages, a program is defined as a composition of several functions. Each such function is invoked with a collection of arguments, and returns a result at the end. When invoked, the function performs some computation, potentially invoking other functions (including itself, recursively). Additionally, the functions can have side effects, modifying some state.

Many programming languages are constructed around this model. In C, programs are composed of several *translation units*, which themselves consist of functions (and global variables, which we do not consider). To create a program out of these translation units, all the functions defined in them are combined into one large program. This step is called *linking*. In this thesis, we call this specific kind of linking the *C linkage model*. In this work, we take a more abstract view of a linkage model. For us, the C linkage model describes that functions are called by their name, are passed arguments which are "C values," and again return a C value. Additionally, we understand the linkage model to include the state the programs operate on, that is, a C heap. When two languages can be linked using the same linkage model, we say they are *ABI-compatible*.[5]

In other languages, the means for structuring and linking programs are a bit different. For example, large OCaml programs are typically subdivided into modules, which couple functions and type definitions to provide additional abstraction barriers. During OCaml linking, these modules are all bundled together with a top-level expression, that is then executed and starts using the functions defined in modules. Another very simple linkage model is presented by the languages described in Section 2.4: The "one big expression" model. Instead of functions, there there are first-class closures, and these can be composed using let-in expressions. This model is simple but powerful, since linked programs can directly be verified using the WP-Bind rule. Instead of functions Unfortunately, it breaks down when multiple languages are involved, or when there are no first-class closures. Nonetheless, this linkage model is what we use for our formal version of OCaml, since this is close enough to the OCaml module system [39].

For historical reasons, the C linkage model has evolved to become the default linkage model in many computing systems. This is because often, the best way of communicating with the operating system is using the C standard library, which (being

[5]ABI: Application Binary Interface. This actually means that both language produce compatible assembly code. Since linkers operate at assembly level, this is what makes language compatible for linking.

written in C) uses the C linkage model. Thus, almost all programming languages include a way to interact with other code using the C linkage model. This specific feature of a programming language is usually called the *Foreign Function Interface*, which (again due to historical reasons) usually specifies how interoperability with C can be achieved.

From a certain point of view, a multi-language program (particularly with the C linkage model) is a bit like a single-language program. It consists of a list of functions, which all take parameters, return results, and call each other. All functions are compatible, since they use the same linkage model. However, the functions can be implemented in different languages. When linking two languages with compatible value models (*e.g.*, C and C++[6]), this is straightforward, since C++ can directly work with and produce C values. When combining a language like OCaml with C, this is much more complicated, since languages like OCaml usually have no support for C values, or C state. THis makes using the Foreign Function Interface between OCaml and C is a lot more complicated: It now involves writing *glue code*, which has to bridge between those two languages. The next chapter discusses this Foreign Function Interface, explaining when glue code is necessary and how it is written.

[6]C++ has classes, but we can roughly work with them in C by treating them like structs.

Chapter 3

Explaining the OCaml-C Foreign Function Interface

In this chapter, we gradually introduce the Foreign Function Interface used to bridge C and OCaml. The Foreign Function Interface requires the user to write *glue code* in C, which bridges between OCaml and C values. To do so, the FFI provides a set of *FFI primitives*, which must be properly composed to convert between values.

This section is based on the OCaml Manual [1, Chapter 22], which explains the FFI. In some cases, it explains code that would be considered wrong according to the rules explained in the manual. Nonetheless, the explained patterns are sometimes used in productive OCaml libraries, and rely on before-now undocumented compiler and runtime invariants.

This chapter introduces the FFI as it is used in reality. In Chapter 4, we introduce formal models of our single languages, and in Chapters 5 and 6, we do so for the FFI itself.

We introduce three kinds of colors, to improve readability. A pinkish red is used to highlight programs, values, and concepts related to C. Blue is similarly used for OCaml. Purple is used for concepts and functions that are part of the OCaml-C Foreign Function Interface. These colors are used throughout this thesis.

Most of this chapter introduces the FFI. Readers familiar with the FFI can skip directly to Section 3.5, which describes which features of the FFI we do (not) model.

Why Is The FFI Necessary?

Before we discuss the FFI, we explain why this is even necessary for OCaml. The reason the FFI is needed is that OCaml and C very different languages. In particular, these languages disagree on the type of values, and on the kind of state. OCaml works with OCaml values, which can be structured, and even include first-class closures. In C, all there is are pointers and integers, and even there, the division is blurry. Further, the OCaml heap stores array of OCaml values, and is garbage-collected. The C memory is flat, and requires manual memory management. Therefore, even seemingly related types like `int ref` or `int*` have very little in common. Although both point to integers, these are not compatible since they are part of different languages. Even further, the pointers are pointers of completely different heaps, which (in this work) the other language can not access without using the FFI. Looking at more complicated types like `(int * (int -> bool)) option`, it becomes more obvious that these values can not "simply" be used from C.

The job of the FFI is now to ensure that C can work with such values at all, even if it is "complicated." It defines how external calls from OCaml to C work, how OCaml values end up looking in C, and how C can manipulate these values, or create new ones. It does so by providing methods and macros, that need to be used in C to extract useful C values from OCaml values. The C code that does so, using these *FFI primitives*, is called *glue code*. The FFI is designed such that in OCaml, calling a C function is relatively straightforward, by forcing all the glue code to be written in C. This design decision is partially explained by the fact that it aims to make C functions accessible from OCaml. This works because C has a straightforward linkage model, where functions can simply be called by their name, since a program is just a list of (named) functions. Since OCaml itself does not really follow this linkage model, but is more based around closures and modules, C can not simply invoke a named OCaml function (since such a thing does not really exist.)[1] Instead, the FFI provides mechanisms to invoke OCaml closures, to allow control flow to pass back and forth between those two languages. This mechanism is looked at in Section 3.4.

3.1 Basics

To begin, consider the following OCaml program:

```
let plus1 n = n + 1 ;;
print_int (plus1 4)
```

This program defines a function `plus1 : int -> int`, which adds 1 to a number. It then applies this function to 4, which thus evaluates to 5, which is then printed. As our first multi-language program, we want to write this `plus1` function in C. To do so, we split our program into two parts, one in OCaml, and one in C. The code is shown in Figure 3.1

We note that this program can actually be compiled and executed, by saving these files in a folder `plus1`, as is also shown in Figure 3.1. When we read it, we see the OCaml program use some special syntax for declaring a function external:

```
external <name> : <type> = "<C name>"
```

[1]Actually, the OCaml FFI provides a mechanism for "naming" certain OCaml functions, but it is rather obscure.

plus1/main.ml:

```
external plus1 : int -> int
       = "caml_plus1" ;;
print_int (plus1 4)
```

plus1/main.c:

```
#include <caml/mlvalues.h>
value caml_plus1(value v) {
    int n = Int_val(v);
    return Val_int(n+1);
}
```

```
$ ocamlopt -o plus1/bin plus1/main.ml plus1/main.c
$ ./plus1/bin
  5
```

Figure 3.1: Writing, compiling, and executing a program written in both OCaml and C.

Using this syntax, we tell the OCaml compiler that `plus1` will be implemented in C, and that calls to it should be replaced by calls to the C function `caml_plus1`. The name we choose for the C function is arbitrary, but a common pattern is to use the OCaml name, prefixed by `caml_`. Once we have done so, using the function in OCaml works as if it was declared normally. Thus, in OCaml, calling an external function is very easy. All the difficulty of dealing with values in different representation must be handled on the C side. Again, this is a design choice particular to the OCaml-C FFI. In other language combinations, the glue code could be distributed differently.

In C, we have the function `caml_plus1`, which is called by OCaml. This function already looks rather weird. First, it takes an argument of type `value`, and has to again return such a `value`. The type `value` is the type used to represent OCaml *values* in C. In C, we are working in the low-level representation of OCaml values, as they are actually implemented in the OCaml runtime. While in OCaml, this low-level implementation is abstracted away by the OCaml runtime, so that OCaml programs seem to manipulate the familiar, high-level values. Understanding how to translate between those two representations becomes easier when we introduce a third layer: the block-level representation. Logically, this layer sits between the OCaml and the C level. While it, like the OCaml level, does not actually exist in hardware, it is already a lot closer to the physical representation of an OCaml value. The main goal of the block level is to separate the encoding of high-level OCaml values into an integer-pointer value system, which is rather canonical, from certain low-level encoding tricks done by the runtime to increase efficiency. At the block level, values are very simple: they are either an integer, or a pointer to a block managed by the runtime. It turns out that encoding an OCaml integer as a block-level integer is also straightforward: Integers remain integers. Other OCaml values have more interesting encodings, which we discuss in Section 3.2.

The block-level-to-C encoding is more involved. To ensure an efficient and nice runtime implementation, this encoding ensures that each runtime value can be encoded into one C machine word, usually considered to have 32 bits. To do so, the last bit is used to discriminate block addresses from integers. Since blocks are allocated with an alignment of at least 2, their addresses are always even. Thus, such a value is considered a pointer iff the least significant bit is not set. Integers must now be encoded into a 32-bit word, where the lowest bit is already set. Thus, the integer is encoded into the remaining 31 bits, by shifting it 1 bit to the right. This also explains why in OCaml, integers overflow below $2^{30}-1$, not below $2^{31}-1$. In C, the type `value` is used to denote a value encoded using this bit pattern. Additionally, the invariant of such a `value` is that all `value`s encoding a pointer must encode a pointer pointing to special memory, managed by the runtime (more on this in Sections 3.2 and 3.3).

This encoding also explains our first two FFI primitives, `Int_val` and `Val_int`. `Int_val` converts a `value` encoding an integer n to the actual C integer n. `Val_int` performs the inverse direction.[2] In the OCaml runtime, these are actually macros, with the following implementation:

```
typedef long            intnat;
typedef unsigned long   uintnat;
#define Val_long(x)     ((intnat) (((uintnat)(x) << 1)) + 1)
#define Long_val(x)     ((x) >> 1)
#define Val_int(x)      Val_long(x)
#define Int_val(x)      ((int) Long_val(x))
```

Also, integers are actually `long`s, which the OCaml runtime expects to be 64-bit on 64-bit systems, and 32-bit on 32-bit systems. Since we ignore overflow, we do not make this distinction. We can even go further: we consider the specific encoding used to discriminate block-level integers from block pointers an *implementation detail*. While we model that this distinction exists, we do not verify programs that manipulate values directly using this low-level representation. Instead, we pretend that instead of transparent macros, these are simply functions that the runtime provides to C, but whose implementation is opaque.[3]

We thus start taking a step back, and instead of describing how these macros/functions are implemented, we limit ourselves to describing how they are intended to be used–that is, we only give specifications. The specification of `Int_val` and `Val_int` are simple to state: They convert between encoded block-level integers, and C integers. For this, we also consider them to have undefined behavior if no integer

[2]The names can be rather confusing. The way to read them is that `Int_val` means "make `int` from `value`", *i.e.*, the target comes first.

[3]The actual runtime provides macros here for performance reasons: Macros do not inhibit compiler optimizations.

is actually encoded. If we were to properly model integer overflow, then re-encoding an integer that is too large would also be considered undefined behavior.

With all of this out of the way, we can now actually understand what the example program from above does: It extracts the integer encoded in the argument `value v`, using the `Int_val` FFI primitive. Then, it adds 1, and re-encodes this integer into a value using `Val_int`. This is then returned. Therefore, it correctly implements a function that jus adds 1 to integers.

3.2 Structured Values

Next, we look at how the other values of OCaml are encoded, and at how the runtime blocks (pointed at by block pointers) look like. First, we handle the simple cases. An OCaml boolean is encoded as a block-level integer. There, `true` is 1, and `false` is 0. If any other value is used where the runtime expects a boolean, we consider this undefined behavior. The case of units is even simpler: Units `()` are simply encoded as 0.

Actual OCaml has inductive types, and the rules outlined above are just special cases of the rules for inductive types. In such definition, like `foo` here, we differentiate two kinds of constructors.

```
type foo = Foo1 | Foo2 | Foo3 of int | Foo4 of int * int
```

Those constructors like `Foo1` and `Foo2` are *constant*, since they have no arguments. Intuitively, they define a constant. The other constructors, `Foo3` and `Foo4`, are *non-constant*, since they take further values. The rule for converting constant constructors to their block-level representation is that they are numbered in their order of declaration, starting at 0, where this numbering skips non-constant constructors. Then, each constant constructor is represented by its assigned number. Thus `Foo1` would be represented by 0, while `Foo2` is represented as 1. Since booleans and units are defined as shown below, the above rules for them are just a special case of the general rule.

```
type bool = true | false
type unit = ()
```

To handle non-constant constructors, we use blocks. A block is a region of memory in the OCaml runtime (block-level) heap, which is managed by the OCaml runtime. Each block has a *header*, and a *body*. The *header* of a block stores the *length* of a block, as well as its *tag*. The header is a single machine word, of which 8 bits are used to store the tag. Thus, the tag is between 0 and 255, both inclusive. Most of the remaining bits store the length of the block, so that on a 32-bit system a block is limited to $2^{22} - 1 = 4194303$ machine words (since the length stores the number of machine words), not including the header. Two further bits are used internally by the garbage collector, they do not concern us. The body of such a block is then an array of block-level values, of the given length. To see how it is used, we best consider some examples. The simplest is that of a binary pair, *e.g.*, `(3,4)`: This is encoded as a block of tag 0, storing $[3,4]$. When passed to C, such a pair is thus present as a *pointer* to such a block, which is allocated in the block-level heap. In general, the contents of an arbitrary-sized pair are encoded as blocks of tag 0, of the appropriate length. Units, which can be understood as nullary pairs, are the exception to this rule. To handle the general case of an inductive type, the tag is used. We again consider our `foo` type, reproduced here for posterity:

```
type foo = Foo1 | Foo2 | Foo3 of int | Foo4 of int * int
```

To encode the non-constant constructors, we again give each a unique index, starting at 0, similar to how we indexed the constant constructors. Again, the other kind of constructor is not counted. Then, the constructor `Foo3`, with index 0, is encoded as a block of tag 0. This block has size 1, and stores the one integer that is the argument of this constructor. The constructor `Foo4`, with index 1, is encoded as a block with tag 1. It now has length 2, storing both its arguments in two fields. For constructors with larger *arity* (number of arguments), a larger block is used. The attentive reader might wonder what happens when we have more than 256 constructors. The answer is that this is a compile error. In fact, the actual limit of non-constant constructors is a bit lower, and depends on the specific OCaml version.[4] This is because OCaml reserves some tags for special kinds of blocks, some of which we describe in Sections 3.4 and 3.5.

As a refresher, binary sums, which use the following definition, are thus encoded as blocks of length 1, with the tag being either 0 or 1. This also shows that polymorphism is erased at runtime, since each OCaml value is encoded as exactly one word, so it does not matter which type it hard. Thus, OCaml types need not be monomorphized.

```
type ('a, 'b) sum = inl of 'a | inr of 'b
```

Not described so far are references, or more general, arrays (with references just being arrays of size 1). The block-level representation of an OCaml reference ℓ, pointing to an array of size n, is simply a block, tag 0, of size n, that stores the representation of each member of the array. There is a key difference between the block backing an array, and that backing *e.g.*, a pair: The former is *mutable*, while the latter is *immutable*. The OCaml compiler can make optimizations based on the assumption that immutable blocks, that is, blocks backing pairs, sums, and inductive types in general, never change. For arrays, this requirement does not exist (since arrays are supposed to change their contents). Whether or not a block is mutable

[4] In OCaml 5.0.0, at most 244 non-constant constructors are allowed.

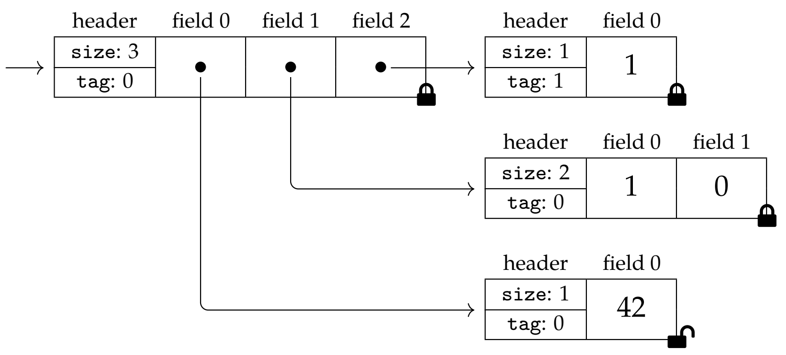

Figure 3.2: The block-level representation of `(ref 42, (true, ()), inr 1)`.

is not part of the physical machine state, it is purely ghost state. Yet, we consider modification of any block not explicitly mutable to be undefined behavior. We also note that the OCaml runtime does not keep track of whether a block is (im)mutable. This distinction is only present in the OCaml compiler, which compiles high-level OCaml code to a lower-level version that is then executed by the runtime. When doing so, it can perform type-based optimizations like removing redundant loads from blocks that are immutable.

As an example summarizing all of this, consider Figure 3.2. There, we display the block-level heap representing of `(ref 42, (true, ()), inr 1)`. Additionally, immutable blocks are marked as locked using 🔒, while the block belonging to the reference is mutable, shown using 🔓.

3.2.1 Working with Blocks

We now discuss the FFI primitives used to manipulate blocks. The first is `Is_block`, which looks at the least significant bit of a `value` to check whether it is a block or an integer. Further, we have `Wosize_val` and `Tag_val`, which allow reading the size and the tag of a block. Since these are implemented as macros, `Tag_val` is actually an lvalue, so that the tag can be assigned to. We do not support such assignments, since changing the tag after a block is created is (to the best of our knowledge) undefined behavior. Finally, we have `Field` and `Write_field`, which are used to access and change the fields of blocks.

As an example, consider the following C function:

```c
value caml_plus1_ref(value v) {
    assert(Is_block(v));
    assert(Wosize_val(v) == 1);
    assert(Tag_val(v) == 0);
    int n = Int_val(Field(v, 0));
    n++;
    Store_field(v, 0, Val_int(n));
    return Val_int(0);
}
```

In OCaml, this function could be used like so:

```
external plus1_ref : ref int -> unit = "caml_plus1_ref" ;;
let l = ref 41 in
    plus1_ref l ;
    print_int(!l)
```

This OCaml client creates a new reference (a one-element array) storing 41, passes this reference to `caml_plus1_ref`, and prints the contents of the reference afterwards. When executing this program, once we reach `caml_plus1_ref`, the argument `v` will be a `value` pointing to a block representing this reference. Thus, all initial asserts will pass, since a reference is indeed encoded as a block of tag 0 and length 1. Then, the content of the reference is read, by reading the content of first field (field 0) of the block, and then extracting the encoded integer. This works since the reference does indeed store an integer (namely 41). After this integer is incremented, it is re-encoded into a `value` and stored back into the reference. This in particular works since references are mutable, allowing us to modify its content. Finally, we return `Val_int(0)`, which encodes the unit value. In OCaml, functions return `unit` instead of `void` when they are invoked only for their side effects, and we must also do so here. Therefore, the overall program, when executed, prints 42.

3.3 Roots and Garbage Collection

With the primitives introduced so far, we can inspect many of the values the FFI hands to us. Additionally, we can also modify them, provided they are not immutable. What we lack so far is a mechanism for creating new values, particularly for allocating new blocks. It turns out that this is not as straightforward as one would hope, due to the garbage collector. The actual method for allocation a block, `value caml_alloc(mlsize_t, tag_t)`, is not hard to use. It takes the size of the new block (as returned by `Wosize_val`), as well as the initial tag. It then returns the new block, with all values default-initialized to zero. To highlight where this goes wrong, we show a wrong example. In this example, we try to define a function `caml_swap_pair` that, given a pair `(a,b)` returns a new pair `(b,a)`. Since pairs a immutable, a new block must be allocated. The OCaml client is straightforward:

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair" ;;
let (b,a) = swap_pair (4,2) in
    assert(b == 2);
    assert(a == 4)
```

It swaps the pair (4,2) and asserts that the new pair has the proper content. A naive attempt at implementing `caml_swap_pair` then looks as follows:

```
 5 value caml_swap_pair(value v) {
 6     value res = caml_alloc(2, 0);
 7     value fst = Field(v, 0);
 8     value snd = Field(v, 1);
 9     Store_field(res, 0, snd);
10     Store_field(res, 1, fst);
11     return res;
12 }
```

Sadly, this code is incorrect. The reason is that `caml_alloc` might do a *garbage collection* (GC) run. OCaml has a garbage collector, which is used by `caml_alloc` to reclaim space if no space for the allocation can otherwise be found. When the garbage collector runs, it can deallocate blocks that it deems unreachable. Blocks become unreachable once the OCaml program no longer needs them. Additionally, the GC can "compact" the heap by rearranging it. To do so, it can decide to move a block around. It will then take care to update all references to this block stored in other parts of the runtime heap and used elsewhere in the OCaml program. Unfortunately, the garbage collector does not know what our C function is doing with the values passed to it. Thus, it might notice that the input pair is no longer used by OCaml (only by our C function), and thus deallocates it. Even worse, it might just move the block to another place in memory, without notifying us of this. Thus, in line 7 or 8, the C function might read from deallocated memory, or from a part of the runtime heap that now stores something that is no longer the input `v`. In the previous examples, this was not an issue, since all FFI primitives outlined until then never cause a GC run. What can, however, also cause a GC run is returning to OCaml itself. Thus, storing a `value` in a global variable can also be a bug, since that value might be moved by the GC while OCaml code is running.[5].

The solution offered by the FFI is that of *rooting*. We can register (the address of) a `value` as a root, which blocks the runtime from considering the referenced block (as well as all blocks transitively reachable from it) unreachable, and thus prevents it from being deallocated. Additionally, if the runtime decides to move the block referenced by this `value`, it will *update* the `value` to the new location (hence roots are registered by-reference). Registering a root storing a block-level (encoded) integer is allowed, and simply skipped by the garbage collector. Once we are no longer interested in a rooted `value`, we need to *unregister* it. Not doing so would constitute a memory leak, since the GC is prevented from deallocating blocks that are no longer needed. Even worse, if we `free` a `value` that was rooted (or let it go out of scope), without properly unregistering it first, this is a soundness issue: At the

[5]It is only a bug if the value is accessed when OCaml code runs in-between

next GC run, the GC will look at (or even modify) all roots, and if the root is now in deallocated memory, this causes a use-after-free bug. Thus all roots must be properly unregistered before the backing storage is deallocated. If this does not happen, we say that a *dangling root* is produced.

To actually register roots, the FFI offers two approaches. Global roots can be registered using `caml_register_global_root(value*)`, and are again unregistered using `caml_unregister_global_root(value*)`. These functions are intended for global variables, or for memory allocated on the C heap. The other approach is for local variables. There, one uses the macro `CAMLlocal1` to declare a local variable of type `value`, which is registered as a root and initialized to `Val_int(0)`. If one needs to register multiple roots, one can use the macro multiple times, or register up to five roots at once by using *e.g.*, `CAMLlocal5`. Using these macros is tricky, since they can not be used on their own. Instead, they rely on two other macros to properly set up the hidden machinery for tracking roots, and to unregister them when the function returns. The first macro is `CAMLparam1`, which is used to register parameters as roots. Like with `CAMLlocal`, there also is a variant for up to 5 parameters. Unlike `CAMLlocal`, this macro must only be used *once*, and it must be used before `CAMLlocal` is used, since this macro also sets up the machinery for `CAMLlocal`. If the function has more than five parameters, the remaining ones can be registered using `CAMLxparam1`, which only registers parameters as roots. When there are no arguments, but local roots are still needed, the macro `CAMLparam0` must be used to initialize the local root machinery. The point of all this machinery is to ensure that all local roots are unregistered when the function returns. This works by replacing the usual `return` keyword with the macro `CAMLreturn`, or with `CAMLreturn0` in case of a void method. This macro then unregisters all roots before returning. In order for this machinery to work properly, one must ensure that local roots do not go out of scope until the function returns using `CAMLreturn`. Otherwise, this can lead to the earlier-explained use-after-free bug.

We can now fix our example. In fact, there are multiple ways. We here present two:

```
value caml_swap_pair(value v) {
    CAMLparam1(v);

    value res = caml_alloc(2, 0);
    value fst = Field(v, 0);
    value snd = Field(v, 1);
    Store_field(res, 0, snd);
    Store_field(res, 1, fst);
    CAMLreturn(res);
}
```

```
value caml_swap_pair(value v) {
    CAMLparam1(v);
    CAMLlocal3(res, fst, snd);
    res = caml_alloc(2, 0);
    fst = Field(v, 0);
    snd = Field(v, 1);
    Store_field(res, 0, snd);
    Store_field(res, 1, fst);
    CAMLreturn(res);
}
```

Both versions are almost the same, the only difference being that the right one additionally registers `res`, `fst`, and `snd` as roots. This is the recommended approach [1]: Ensure all `value`s are rooted all the time. While this approach is correct, it is not the most efficient. Instead, we can use the knowledge that the GC only runs in line 8.[6] Thus, we do not need to root `value`s that do not need to "survive" the GC event in line 8. It turns out that `res`, `fst`, and `snd` are only initialized after the GC runs. In particular, the value `res`, returned by `caml_alloc`, is valid even if `caml_alloc` performs a GC run, since it is allocated after this run.[7] Thus the left version is also valid, and perhaps a bit more efficient. To see that it is valid, however, requires careful tracking of when GC runs can occur, and which variables are accessed *across* these runs. Finally, we remark that we could also have used `caml_register_global_root` to root our local variables, as long as we unregister them with the matching function before returning. This is considered unorthodox, and likely inefficient, since global roots are expected to stay roots "longer."

3.4 Advanced Topics

The FFI primitives presented until now allow us to create, inspect and modify all basic OCaml values, in particular all those that can be defined (co)inductively or as records. These are not the only values OCaml offers. Notably, it also has closures, which we look at in this chapter. Further, it also offers a way of embedding C data into an OCaml value using *custom blocks*. In this chapter, we explain both these mechanisms.

3.4.1 Callbacks

OCaml is a functional language, which includes (recursive) closures. The OCaml FFI includes support for invoking OCaml closures from C. In a somewhat disappointing move, it does not allow creating an OCaml closure from C. To invoke a closure, the FFI primitive `caml_callback(value, value)` is used. The first argument of this is the closure, the second is its argument. It returns the result of applying the closure to a value. Since in OCaml, functions are cascading by default, this suffices to apply a closure to several arguments, by just calling the function several times. The block-level representation of a closure is a special kind of block, with tag 247 (all tags greater than 243 are considered "special"). This block stores two values: The second is an integer encoding some metadata, while the first points to an opaque blob storing the code, as well as the captured arguments. The FFI does not provide methods for deeply inspecting such a closure, only for invoking it. It also does not provide methods for constructing new closures, that can only happen in OCaml proper. The OCaml FFI does not speak of closures, but of callbacks, since invoking a closure means that we call back to OCaml code. Since this OCaml code can do arbitrary things, it can also cause a GC run. To ensure safety, values that need to

[6]This work does not consider Multicore OCaml. But even then, the FFI and the GC are cleverly designed so that this is still true [43], by making certain accesses atomic.

[7]Otherwise, it would be impossible to properly use `caml_alloc`.

survive this call must be properly rooted. Additionally, the code in such a closure can again call an external function, implemented in C, which can again call a closure, and so on. In fact, all of this can happen recursively.

3.4.2 Custom Blocks

The final feature we include in this thesis are custom blocks. A custom block is a block that can be used as an OCaml value, while storing C data. This can be used to create new types. For example, we can create a buffer library, where the buffers store C bytes and are allocated in C. To safely pass such a pointer to OCaml, the pointer is stored in an OCaml custom block. This also ensures that OCaml code does not inspect this pointer, since it can only be accessed using the FFI. To create such a custom block, OCaml provides the primitive `caml_alloc_custom`, which takes four parameters. The second parameter is the size of this custom block. If we only want to store a pointer, it would be `sizeof(void*)`. The first parameter is (a pointer to a) `struct custom_operations`. This struct can be used to override the standard operations OCaml provides on all values, which are:

- A finalization method invoked just before the GC frees this custom block.
- A comparator, used by `<, <=, =, ...`
- A hashing operator, used by the `Hashtbl` module.
- Serialization and deserialization operators.
- Additionally, a custom name for the kind of custom data is needed.

The last two parameters, called used and max are used to fine-tune the garbage collector. When we allocate several custom blocks, which manage resources not tracked by the garbage collector, we might want that our finalization methods are called often enough so that these resources are released in a timely fashion. To achieve this, we can tell the garbage collector how much off-heap resources we are managing (the used argument), and what the limit is (the max argument). When the sum of used exceeds the max, the garbage collector will do a full run and finalize all now-unused custom blocks. When we do not care about this, we can set used to 0 and max to 1. Similarly, if we do not care about overriding any of the standard operations, OCaml provides default implementations (which do nothing or raise exceptions, as appropriate).

As an extended example, we want to implement a library that provides access to a very simple kind of buffer, where each buffer stores one C integer. Such a library would use the following OCaml signature:

```
type buffer
external buf_init  : int -> buffer         = "caml_buf_init"
external buf_read  : buffer -> int         = "caml_buf_read"
external buf_write : buffer -> int -> unit = "caml_buf_write"
```

The C implementation of the methods looks like this:

```
void caml_buf_finalize(value buf) {
    free(*((int**)(Data_custom_val(buf))));
}
static struct custom_operations buf_ops = {
  "melocoton.example.buf",
  caml_buf_finalize,
  custom_compare_default, //default values, which raise exceptions
  custom_hash_default,
  custom_serialize_default,
  custom_deserialize_default,
  custom_compare_ext_default,
  custom_fixed_length_default
};
value caml_buf_init(value i) {
    value buf = caml_alloc_custom(&custom_ops, sizeof(int*), 0, 1);
    int* cbuf = malloc(sizeof(int));
    *cbuf = Int_val(i);
    *((int**)(Data_custom_val(buf))) = cbuf;
    return buf;
}
value caml_buf_read(value buf) {
    return Val_int(**((int**)(Data_custom_val(buf))));
}
value caml_buf_write(value buf, value nv) {
    **((int**)(Data_custom_val(buf))) = Int_val(nv);
    return Val_int(0); // unit
}
```

This example shows how the data in the custom block is accessed: using the macro/FFI primitive `Data_custom_val`. This macro returns a pointer to the first user-controlled byte in such a custom block. Since this pointer has type `void*`, we must cast it to an `int**`, as the custom block stores an `int*`. Further, this example uses a custom finalizer to `free` the used buffer when it is no longer used.

Finally, the OCaml signature warrants a closer look. This signature declares, but does not define, the type `buffer`. This type is thus an *existential type*, and the OCaml type system enforces that only these three functions implemented in C can operate on these values. The actual values of this type are custom blocks, which (even without the type system) can not be inspected in OCaml, but only using the FFI. In fact, any attempt at defining `buffer` in OCaml would already be incorrect, since OCaml can not even express that a type should be inhabited by custom blocks. In summary, these custom blocks are completely opaque to OCaml.

3.5 Features Not Considered

The features explained so far are the features we model in this thesis. Besides these, the OCaml FFI has many other features which we do not consider.

Exceptions

OCaml has support for exceptions, which can be raised and caught again. The runtime supports raising exceptions using `caml_raise`, and provides an exception-aware callback function `caml_callback_exn`, which might return a special exception object. Internally, the runtime implements exceptions using `setjmp` and `longjmp`. Performing a `longjmp` to the exception handler causes C to pop several stack frames. To prevent local roots from causing use-after-frees, this also unregisters all local roots up to but excluding the method catching the exception. Interestingly, the runtime can almost be used to add exception support to C. Thus, modelling it breaks the assumption that all function calls are well-bracketed. Since we reckon that this would make the formal model significantly more complicated, we do not handle it.

As an aside, when a function like `caml_callback_exn` returns an exception, this is encoded by setting the least significant bit to 0 (like for regular pointers), but by also setting the second-least significant bit to 1. Since blocks have an alignment of at least 4, such blocks do not otherwise occur. This exception value is however unsafe: When stored in a root during garbage collection, it causes a soundness issue. Thus, programmers working with rooted exception must quickly test for (and then unset) this bit, before the garbage collector can run.

Multicore OCaml

OCaml 5 added proper support for threading to OCaml, including a concurrent memory model. This memory model is also interesting from a theory perspective, since it "binds data races in space and time" [9]. A concurrent separation logic for Multicore OCaml has also been build [27]. This memory model may impose additional complexity to the glue code, since C code is subject to a weaker memory model, and thus might need to use the proper atomic stores and loads to ensure that functions behave consistent with the OCaml memory model. We do not consider multi-threading, and thus ignore all this.

More Special Kinds of Values

We can look at the list of all special tags to see some more missed features. This thesis, based on OCaml 5[8], has 12 special tags, namely 244 to 255 (both inclusive). Tags 244, 245, 246, and 250 are used for `lazy` OCaml values, which represent thunked computations. FFI support for them is almost non-existent. Tag 247 is for closures, which we have discussed. We also discussed custom values, using tag 255. Tag 248 is for objects. The O in OCaml stands for *objective*, since OCaml has support

[8]ignoring OCaml's multi-core features

for object-oriented development. The FFI has support for this, which in particular allows calling instance methods. Tag 249 is an internal tag. Tag 251 is used for abstract datatypes. In general, tags $\geqslant 251$ store data that is not inspected by the garbage collector. Tag 252 is used to store byte buffers, as well as strings. At runtime, both look the same, but string buffers are sometimes considered immutable.[9] There is enough FFI support for productively working with OCaml strings or byte buffers. But the FFI is not really needed, since this data can just be accessed using normal C functions for accessing data, like `memcpy`. Tag 253 is used to store doubles, which are encoded as a block just storing that single double. If instead an array of doubles is needed, they are stored using tag 254. All other tags are used to encode the non-constant constructor arguments.

In this work, we ignore all of them except for callbacks and custom blocks. We also do not have user-defined inductive types, thus the only tags that actually appear are 0, 1, 247, and 255.

Uninitialized Data

The function `caml_alloc` default-initializes its memory with `Int_val(0)`, which is useful since this in particular makes this block safe to inspect by the garbage collector. However, this default-initialization can be redundant if all the zeros are overwritten later. Thus, there is `caml_alloc_shr` and `caml_alloc_small`, which can be used to allocate an uninitialized block (depending on the size). This block, must then be completely initialized before the next GC run happens. To initialize a block, `caml_initialize` can be used, which is more efficient since it requires that no old value is being overwritten. Blocks allocated with `caml_alloc_small` are allocated in the so-called young generation. The OCaml GC reserves a special area for newly allocated blocks, which it scans more often. This is because it is assumed that most fresh blocks quickly become unreachable again. If blocks remain reachable long enough, they are graduated into the old generation, where they are checked less often. Blocks allocated with `caml_alloc_shr` are directly allocated in the old generation. Additionally, a slight optimization is possible for blocks that are not part of the young generation. When a field in block $\mathtt{b}_2$ is initialized with a block $\mathtt{b}_1$, and $\mathtt{b}_1$ is not a young block (*e.g.*, because $\mathtt{b}_1$ was allocated with `caml_alloc_shr`), then initializing that field in $\mathtt{b}_2$ can be done by a direct C assignment, instead of by using `caml_initialize`. Similarly, a direct assignment can be used if a field is initialized with an integer.

In our formal model, we do not consider any of this. There, the only method for allocation will be `caml_alloc`.

[9]It turned out that making them always immutable broke existing OCaml libraries, so this is now a compiler flag.

Simplifications

In the next chapters, we develop a formal model of the OCaml FFI. To do so, we further simplify some of the features.

Some simplifications are already introduced by the formal model of OCaml itself, presented in Section 4.3. Since we do not model all inductive types, but only binary products and co-products (sums), we do not need the machinery for general inductive types. Further, we do not consider records, in particular not mutable records. Mutable records are interesting because the mutability is annotated per-field, so that some fields of a record can be mutable, but others are immutable. In our formal model, mutability will be tracked per-block. If needed, a mutable record can be modelled as a heterogeneous array (which would however be ill-typed).

We also radically simplify custom blocks. Instead of allowing arbitrarily large C blobs, we limit them to one machine word. This machine word is also no longer modified using plain C reads and assignments; instead two separate FFI primitives (for reading and writing) must be used. We do not consider this a real restriction: If one previously had to store a lot of data in a custom block, this data could instead be stored on the C heap, and only a pointer to this data is stored in the custom block. This even has the added benefit that the data does not need to be moved by the garbage collector when it decides to compact the heap. We also remove the mechanism for fine-tuning the garbage collector, which is useless once we remove finalizers since they do not affect observable behavior. To simplify even more, we remove the means of overriding the standard operations for them. Thus, comparing them will always trigger undefined behavior. (Our formal version of OCaml already removes all the other features for which behavior can be defined using this struct, like (de)serialization.)

This also removes finalizers. Finalizers are often considered an anti-pattern, since they can be fragile: A object undergoing finalization is still alive, but no longer reachable. Finalizers are often prohibited from making the object reachable again, since this would result in references to an object that is already deallocated. Further, finalizers are hard to model in a program logic. They can be considered a kind of non-linear control flow, which we wanted to avoid in this thesis and the underlying paper [13]. Even further, properly reasoning about them in Separation Logic is an unsolved research problem.

Finally, since we removed exceptions, we can simplify the roots machinery. We remove all the macros related to `CAMLlocal`, and instead consider only the functions for global roots, *i.e.*, `caml_register_global_root`. This means that our program logic does not handle the rather complicated interaction between `CAMLparam`, `CAMLlocal`, and `CAMLreturn`. We do not consider this a strict weakness: Instead, these macros can now be build as actual C macros, which internally construct a linked list of root pointers, which are then registered using the primitive that models `caml_register_global_root`. The other main advantage of these macros–performance–is also irrelevant for formal verification. So, our formal model of rooting just provides two primitives, one for registering, and one for unregistering a root. These are orthogonal to other runtime features, and one should be able to construct versions of `CAMLlocal` or `CAMLreturn` on top of these primitives. We also hope that these are general enough to not just handle the rooting mechanisms currently implemented by the FFI, but also allow specifying new proposed rooting mechanisms like Boxroot [29].

3.6 Anticipating the Formal Model

We now have a rough idea how the FFI works. In particular, we have that external calls from OCaml to C are easy to use in OCaml, where we can just call a C function by its name. In C, implementing these functions is harder, since one to write some glue code using FFI primitives before actual computation with the passed values can happen.

We saw that both languages need a kind of external call: In OCaml, these calls go to C, and in C, these calls are used to invoke FFI primitives. In C, these kinds of calls are needed in general to invoke other C functions. We thus extend our general concept of languages with external calls, and formalize the notions of linkage models and ABI-compatibility. These external call specification are a key ingredient for language locality, since they allow abstracting over the other side's implementation, and allow reasoning about external calls as if they were implemented in the same language.

In Chapter 5, we then actually link these two languages, by constructing a formal model of the runtime build atop a linker (which is described in Appendix A). This formal model "wraps" an OCaml external call to C, so that it becomes ABI-compatible with C. It also implements all the FFI primitives.

Chapter 4

Modelling Single Languages

In this chapter, we describe the operational semantics and the program logics of our two individual languages. Before doing so, we must first define what a language with external calls is, and also adapt the Iris framework around such languages.

We then develop simplified formal models of C and OCaml. The literature already offers accurate formal models of C [23, 4, 40] and OCaml [27]. Unfortunately, these accurate formal models often involve some amount of additional book-keeping, since they have to account for the idiosyncrasies of OCaml and C. Additionally, they include features not used by the Foreign Function Interface we are considering. We thus take liberty to simplify and streamline our formal models of C and OCaml, while taking care to not affect those features actually used by the Foreign Function Interface.

The formal model of C is called $\lambda_{\mathtt{C}}$, described in Section 4.2, while that of OCaml is called $\lambda_{\mathtt{ML}}$, and is described in Section 4.3, along with a logical relation in Section 4.5. Section 4.4 examines the differences between both languages.

We use the *color scheme* already introduced in the last chapter to separate C code from OCaml code. Values, expressions, and assertions about C and $\lambda_{\mathtt{C}}$ are typeset in a pinkish red. Values, expressions, and assertions about OCaml and $\lambda_{\mathtt{ML}}$ are typeset in blue.

4.1 Programs and Functions

To formally account for programs using a linkage model based on explicitly named functions, like the C linkage model, we adjust our definition of a language, and redevelop Iris' theory for our changed definition. We thus introduce programs and function calls into our languages. A function call $\mathsf{call}\ \mathtt{fn}\ \vec{e}$ is a special expression (similar to how values also are special expressions), consisting of the function name $\mathtt{fn} : \mathsf{string}$, as well as a list of arguments. The order these arguments are evaluated in depends on the evaluation order, but they must be fully evaluated before the function call is executed, so that only values are passed. To execute a function call, the called function must exist in the current *program*. Such a program is a named set of functions $\mathsf{p} : \mathsf{Prog}$. Formally, this "named set" is just a map from identifiers to function definitions. The type of function definitions, Func, depends on the language.

$$\mathsf{Prog} \ni \mathsf{p} \triangleq \mathsf{string} \overset{\mathsf{fin}}{\rightharpoonup} \mathsf{Func}$$

Every language must define how a function, applied to a list of arguments, is evaluated. Formally, this is specified by a function $\mathsf{applyFunc} : \mathsf{Func} \to \vec{\mathsf{Val}} \to \mathsf{option}\,\mathsf{Expr}$. The result is optional, since function application can fail (for example, if the number of supplied and expected arguments does not match). Additionally, calls to functions defined within the current program are now required to step, assuming that function application is successful:

$$\textsc{SFunCall}\quad \frac{\mathsf{p}[\mathsf{fn}] = \mathsf{F} \qquad \mathsf{applyFunc}(\mathsf{F}, \vec{v}) = e}{(\mathsf{call}\ \mathsf{fn}\ \vec{v}, \sigma) \longrightarrow^{\mathsf{p}}_{\mathsf{h}} (e, \sigma)}$$

As seen there, the (head) step relation gains an additional parameter, the current program p. This is unused in all rules except the one for (internal) calls.

Note that we do not need to explicitly track a call stack. The reason is that our languages do not include an explicit return statement. Instead, the head redex (which already is within a context) is replaced by the result of the function application. In a way, the call stack is implicitly implemented by the existing context mechanism. This is not surprising, given that all our languages are expression-based.

We divide function calls into two kinds: *internal* and *external*. A function call to a function named fn is internal if $\mathsf{fn} \in \mathsf{dom}\,\mathsf{p}$. Otherwise, it is external. We require that external functions do not step, *i.e.*, they have undefined behavior. This is because from the point of view of the program, this is like calling a function that does not exist. It is only later, during linking, that these methods are added and that these external calls start having proper behavior.

Program Logic Next, we develop a useful weakest precondition for such a language with function calls. To reason about internal calls, we can simply add the current program as a parameter to our weakest precondition. We thus change our weakest precondition to the following:

$$\mathsf{wp}\ e\ @\ \mathsf{p}, \Psi\ \{Q\}$$

Formally, we have $\mathsf{wp} : \mathsf{Expr} \to \mathsf{Prog} \to \mathsf{Proto} \to (\mathsf{Val} \to \mathit{iProp}) \to \mathit{iProp}$, where the role of $\Phi : \mathsf{Proto}$ is explained shortly. The rule we want for internal calls, WP-Call-Internal, is then shown in Figure 4.1. It straightforwardly reflects the operational

$$\textsc{WP-Call-Internal}\quad \frac{p[fn] = F \qquad \mathsf{GenericApplyFunc}\ F\ \vec{v} = e \qquad \mathsf{wp}\ e\ @\ p, \Psi\ \{Q\}}{\mathsf{wp}\ \mathsf{call}\ fn\ \vec{v}\ @\ p, \Psi\ \{Q\}}\,*$$

$$\textsc{WP-Call-External}\quad \frac{fn \notin \mathsf{dom}\ p \qquad \Psi\ fn\ \vec{v}\ Q}{\mathsf{wp}\ \mathsf{call}\ fn\ \vec{v}\ @\ p, \Psi\ \{Q\}}\,*$$

Figure 4.1: Program logic rules for internal and external calls.

semantics of function calls. The other parameter is needed for external calls. We mentioned earlier that the default linkage model for our languages is that several collections of functions can be combined into one, to create a complete program. While we could limit ourselves to verifying programs where all calls are internal, this would not be *modular*. Further, it would make it impossible to reason about *foreign* calls, which are calls made to functions implemented in another language. To reason about such calls, we introduced the additional parameter $\Psi : \mathsf{Proto}$ to our weakest precondition. These *protocols* are defined as follows:

$$\mathsf{Proto} \triangleq \mathsf{string} \to \vec{\mathsf{Val}} \to (\mathsf{Val} \to \mathit{iProp}) \to \mathit{iProp}$$

Protocols give specifications to external calls, by encoding the condition that needs to be proven in order to show that a call is correct. Protocols are not a new idea: they have already been used to formally reason about *e.g.*, effect handlers [7], where they define the semantics of effects, and can be seen as an alternative formulation of open simulations [15]. Similar to there, our protocols are *predicate transformers* (on separation logic predicates), that describe the semantics of an external call by expressing which predicates can be established on their return value. For example, the specification that a function `"foo"`, applied to the one argument n, returns $n + 1$, is be expressed as follows:

$$\Psi\ fn\ \vec{v}\ Q \triangleq \exists n.\ fn = \texttt{"foo"} * \vec{v} = [n] * Q\ (n + 1)$$

This is because for all n, $\Psi\ \texttt{"foo"}\ [n]\ (\lambda v.\ v = n + 1)$ is true.

The weakest precondition, which previously was a predicate transformer from predicates on values to predicates on expressions, is now also a transformer from predicates on external functions calls to predicates on expressions. In fact, it will turn out that the two predicate transforms, the weakest precondition itself and the external call specifications, are mutually intertwined by program linking. When stumbling upon an external call while verifying a program, the weakest precondition simply looks up the specification of that external call in Ψ, and then continues once the call returns. This explains the rule WP-CALL-EXTERNAL. To avoid unsoundness, this rule is applicable only to external calls. Thus, the verification certificate for a program using external calls still has "holes," since the specification of an external call is just assumed. Justifying the reasoning rules of Figure 4.1 requires changing the definition of the weakest precondition.

Definition 4.1 (Weakest Precondition with External Calls)

$$\mathit{reducible}(p, e, \sigma) \triangleq \exists e'\, \sigma'.\, (e, \sigma) \longrightarrow^{p} (e', \sigma')$$

$$\mathsf{wp}\; e \mathbin{@} p, \Psi\, \{Q\} \triangleq \forall \sigma.\, SI(\sigma) \Rrightarrow\!\!* \begin{cases} SI(\sigma) * Q(v) & e = \overline{v} \\[1ex] fn \notin \mathsf{dom}\;\; p\, * & \\ \dot{\Rrightarrow} \exists Q'.\, SI(\sigma) * \Psi\; fn\; \vec{v}\; Q' * & e = K[\mathsf{call}\; fn\; \vec{v}] \\ \quad \triangleright \forall v'.\, Q'\; v' \mathrel{-\!\!*} & \\ \qquad \mathsf{wp}\; K[v'] \mathbin{@} p, \Psi\, \{Q\} & \\[1ex] \mathit{reducible}(p, e, \sigma) * & \\ \forall e'\, \sigma'.\, (e, \sigma) \longrightarrow^{p} (e', \sigma') \mathrel{-\!\!*} & \textit{otw.} \\ \quad \dot{\Rrightarrow} \triangleright \dot{\Rrightarrow} SI(\sigma') * \mathsf{wp}\; e' \mathbin{@} p, \Psi\, \{Q\} & \end{cases}$$

We add a new case to the weakest precondition, which looks up the specification of external calls if the current head redex is an external call. Afterwards, we continue with $\mathsf{wp}\; K[v'] \mathbin{@} p, \Psi\, \{Q\}$. Since we give up $SI(\sigma)$ "before" the call, the call has access to the state interpretation, and can thus modify the state. The reason we introduce Q' is very technical: It makes proving WP-WAND slightly easier, since Ψ is not required to be monotone in its predicate argument. Similarly, the reason that the call is allowed to occur in any context, is to still validate the WP-BIND rule.

It turns out that protocols Proto form a lattice, by pointwisely inheriting the lattice structure of $\mathbb{P}\mathrm{rop}$. The protocol $\bot \triangleq \lambda\, fn\, \vec{v}\, Q.\, \bot$ denotes that no external calls are valid, whereas $\top$ is the protocol where every external call is possible, but diverges (since Q can be false). The join (lowest upper bound) $\Psi_1 \sqcup \Psi_2 \triangleq \lambda\, fn\, \vec{v}\, Q.\, \Psi_1\; fn\, \vec{v}\, Q \vee \Psi_2\; fn\, \vec{v}\, Q$ is interesting, especially when the names of the functions described by Ψ_1 and Ψ_2 are disjoint. The combined specification then describes a program providing both calls. This way, our protocols can express the behavior of entire sets of functions, instead of just one. It also includes total order $\Psi_1 \sqsubseteq \Psi_2 \triangleq \forall fn \vec{v} Q.\, \Psi_1\; fn\; \vec{v}\; Q \mathrel{-\!\!*} \Psi_2\; fn\; \vec{v}\; Q$, which expresses that Ψ_1 is stronger than Ψ_2.

Somewhat unintuitively, the a specification Ψ_1 is stronger (in the intuitive sense) than Ψ_2 when $\Psi_2 \sqsubseteq \Psi_1$. Intuitively, a specification is stronger if it has weaker precondition (allows to be called with more values), and stronger postconditions (gives a finer description of the result values). Now, for a concrete call, $\Psi\ \mathsf{fn}\ \vec{v}\ Q$ are the preconditions of that call, while the postconditions are encoded in a contravariant way by usually including something like $\forall v.\ \mathsf{RealPostCond}(v) \to Q(v)$ as a precondition. When thus considering that a stronger specification Ψ_1 should have weaker preconditions than Ψ_2, writing it as $\Psi_2 \sqsubseteq \Psi_1$ makes sense, since that means that the preconditions of Ψ_2 are stronger than that of Ψ_1. Since postconditions are usually encoded in a contravariant way, this notation can also be used when a stronger postcondition is needed.

Adequacy Finally, we need to prove adequacy for our new weakest precondition. As mentioned earlier, a proof of a weakest precondition for a program with external calls still has holes, thus it is not a suitable precondition for adequacy. Instead, we can only prove adequacy for programs without external calls. This is indicated by setting the protocol to $\bot$, *i.e.*, the empty protocol, where no calls are available. After generalizing our notion of safety to programs in the obvious way, we can prove adequacy.

Theorem 4.2 (Adequacy of Weakest Precondition with External Calls) *Let* p *be a program,* σ *be a state such that* $\vdash \mathrm{SI}(\sigma)$*, and* $Q' : \Sigma \to \mathsf{Expr} \to \mathbb{P}\mathrm{rop}$ *such that* $\forall e'\ \sigma'.\ \mathrm{SI}(\sigma') * \mathsf{safe}(p, \sigma', e', Q) \Rrightarrow Q'(\sigma', e')$*. Then the following holds:*

$$\vdash \mathsf{wp}\ e\ @\ p, \bot\ \{Q\} \implies \forall e'\ \sigma'.\ (e, \sigma) \longrightarrow^{p*} (e', \sigma') \implies Q'(\sigma', e')$$

Proof Like the regular Iris proof. When encountering the new case in wp, we have a proof of $\Psi\ \mathsf{fn}\ \vec{v}\ Q \ast\!\ast\ \bot$, as $\Psi = \bot$, which is a contradiction. □

Note that in Coq, we do not give a direct proof of the above lemma. Instead, this theorem will follow from the adequacy of the weakest precondition for modules, Theorem 6.1. We nonetheless show the theorem to highlight that external calls are assumptions, and that adequacy only applies when a program does not make any external calls.

Language-Locality We note that this mechanism for abstractly specifying external calls is *one of the key ideas* of this work. Using these protocols, we can abstract away the implementation of a function. Thus, the implementation language of that function becomes irrelevant. All that is needed is that the target function is ABI-compatible, *i.e.*, that it has a compatible linkage model. This allows us to reason about external functions *as if* they were implemented in the same language we are currently working in. This is one ingredient of working language-locally: As long as some piece of code is written only in one language, we can reason about it as if the other language did not exist. Only when making external calls do we need to consider the other language, and even then, we can choose to ignore it if the other language can be given a proper specification that explains it behavior using only concepts known from the local language. The fact that we can reason as if the other language did not exist is further justified by proving various compositionality lemmas, that allow us to *lift* correctness proofs. If one part of a multi-language system has been verified in a single language, we later want to lift this to the multi-language setting without causing many further proof obligations.

4.1.1 More on Protocols

To increase readability, we define Hoare Triple-like notation for protocols:

$$\forall \vec{x}.\, \langle P \rangle\, fn\, \vec{v}\, \langle v.\, \exists \vec{y}.\, Q \rangle \triangleq \lambda fn' \vec{v}' Q'.\, fn' = fn * \exists \vec{x}.\, \vec{v}' = \vec{v} * P * (\forall v, \vec{y}.\, Q \mathrel{-\!\!*} Q'\, v)$$

This then allows us to rewrite the "`foo`" specification from above like so:

$$\forall n.\, \langle \top \rangle\, \texttt{"foo"}\, [n]\, \langle v.\, v = n + 1 \rangle$$

Note that such a triple denotes an assumption, not a verified function.

Further, we can define a kind of weakest precondition on entire programs, which are again a protocol. This "weakest precondition protocol" $\mathsf{progwp}\ p\ \Psi : \mathsf{Proto}$ is the protocol induced by p, with external calls themselves defined by Ψ. In other words, it is the protocol that describes what happens when the programs in p are actually executed. Its formal definition is as follows:

$$\mathsf{progwp}(p, \Psi)\ fn\ \vec{v}\ Q \triangleq fn \in \mathsf{dom}\ \ p * \mathsf{wp}\ \mathsf{call}\ fn\ \vec{v} @ p, \Psi \{Q\}$$

To simplify reasoning, we introduce specification judgments $\Psi_1 \vdash p : \Psi_2$. This describes that the functions in program p satisfy specification Ψ_2, provided that the external calls in p are interpreted using Ψ_1. Formally, this is just notation for $\Psi_2 \sqsubseteq \mathsf{progwp}(p, \Psi_1)$. We can also give a consequence rule for these judgments, which allows strengthening the assumed specification, and weakening the specification that p must satisfy. Remember that, unlike with regular condition, a stronger protocol is entailed by a weaker protocol.

Theorem 4.3 (Consequence Rule For Specification Judgmenets)

$$\frac{\Psi_{\mathsf{ass}} \sqsubseteq \Psi_{\mathsf{ass}'} \qquad \Psi_{\mathsf{ass}} \vdash p : \Psi_{\mathsf{spec}} \qquad \Psi_{\mathsf{spec}'} \sqsubseteq \Psi_{\mathsf{spec}}}{\Psi_{\mathsf{ass}'} \vdash p : \Psi_{\mathsf{spec}'}}\ \textsc{JudgmentWeaken}$$

Proof By definition, using WP-Wand. □

4.1.2 Intra-Language Linking

This extended weakest precondition, with a protocol for external calls, was motivated by external function calls to functions implemented in a different language. While

$$\begin{array}{ll}
\text{even}(m) \triangleq & \text{odd}(m) \triangleq \\
\quad \mathsf{if}\ m = 0\ \ \mathsf{then\ true\ else\ odd}(n-1) & \quad \mathsf{if}\ m = 0\ \ \mathsf{then\ false\ else\ even}(n-1) \\
 & \\
\Psi_{\text{even}} \triangleq & \Psi_{\text{odd}} \triangleq \\
\forall n.\, \langle n > 0 \rangle\ \texttt{"even"}\ [n] & \forall n.\, \langle n > 0 \rangle\ \texttt{"odd"}\ [n] \\
\quad \langle b.\, b = \mathsf{true} \iff n \mod 2 \equiv 0 \rangle & \quad \langle b.\, b = \mathsf{true} \iff n \mod 2 \equiv 1 \rangle
\end{array}$$

Figure 4.2: Two mutually recursive functions with their specifications.

we have not outlined how linking works in this case, we can already verify a linking operator when all functions have the same language. Semantically, if we have two disjoint programs p_1 and p_2, we can easily combine them into a larger program $p_1 \cup p_2$. It turns out that in the program logic, we can similarly compose specification judgments. Assume that $\Psi_2 \vdash p_1 : \Psi_1$ and $\Psi_1 \vdash p_2 : \Psi_2$. In other words, the functions in p_1 satisfy the specifications in Ψ_1, assuming that external calls in p_1 behave as described by Ψ_2, and vice-versa. We can then prove that $\bot \vdash p_1 \cup p_2 : \Psi_1 \sqcup \Psi_2$, *i.e.*, that the combined program satisfies both specifications, and does no longer have any external calls. For example, imagine two functions which mutually recursively call each other, like in Figure 4.2. There, we can establish that $\Psi_{\text{odd}} \vdash p_{\text{even}} : \Psi_{\text{even}}$ and $\Psi_{\text{even}} \vdash p_{\text{odd}} : \Psi_{\text{odd}}$, expressing that each function is correct under the assumption that the other function also is. Using the above result, this can then be turned into a closed program satisfying both specifications. Note that the use of mutual recursion does not pose an issue, thanks to step-indexing.

The full theorem is more general, since it allows the resulting combined program to still make external calls to the functions in Ψ_{axiom}:

Theorem 4.4 (Intra-language linking) *Let* $\Psi_1, \Psi_2, \Psi_{\text{axiom}} : \mathsf{Proto}$ *and* $p_1, p_2 : \mathsf{Prog}$ *with* $p_1 \mathbin{\#\#} p_2$*. Further,* Ψ_{axiom} *must be* $\bot$ *for all* $s \in \mathsf{dom}\ p_1 \cup \mathsf{dom}\ p_2$*. Then, assuming* $(\Psi_2 \sqcup \Psi_{\text{axiom}}) \vdash p_1 : \Psi_1$ *and* $(\Psi_1 \sqcup \Psi_{\text{axiom}}) \vdash p_2 : \Psi_2$*, we have* $\Psi_{\text{axiom}} \vdash p_1 \cup p_2 : \Psi_1 \sqcup \Psi_2$*.*

This theorem can also be expressed as an inference rule:

$$\frac{\begin{array}{c} p_1 \mathbin{\#\#} p_2 \qquad \Psi_{\text{axiom}} \mathbin{\#\#} (\mathsf{dom}\ p_1 \cup \mathsf{dom}\ p_2) \\ (\Psi_2 \sqcup \Psi_{\text{axiom}}) \vdash p_1 : \Psi_1 \qquad (\Psi_1 \sqcup \Psi_{\text{axiom}}) \vdash p_2 : \Psi_2 \end{array}}{\Psi_{\text{axiom}} \vdash p_1 \cup p_2 : \Psi_1 \sqcup \Psi_2}\ \textsc{WP-Link-Intra}$$

Proof by Löb induction/wp simulation. To prove the weakest precondition of a function $F \in p_1 \cup p_2$, we use the result for p_1 (or symmetrically p_2). We follow the proof that the body of F executes correctly in p_1, which is trivial except when it does an call, where there are three cases:

$$
\begin{aligned}
\mathsf{Val} \ni w &::= (n : \mathbb{Z}) \mid (a : \mathsf{Addr}) \mid \mathsf{NUL} \mid \mathsf{fn} \\
\ominus &::= - \mid ! \mid \sim \mid \mathsf{p2i} \mid \mathsf{i2p} \\
\otimes &::= + \mid - \mid \times \mid \div \mid \% \mid \& \mid | \mid \hat{} \mid \ll \mid \gg \mid < \mid \leqslant \mid = \\
\mathsf{Expr} \ni c &::= w \mid x \mid \ominus c \mid c \otimes c \mid \mathsf{malloc}(c) \mid \mathsf{free}(c, c) \mid *c \mid *c \leftarrow c \\
&\quad \mid \mathsf{let}\ x = c\ \mathsf{in}\ c \mid \mathsf{if}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}(c)\ c \mid \mathsf{call}\ c_F\ \vec{c} \\
\mathsf{Cell} \ni cl &::= \star \mid \dagger \mid w \\
\Sigma_C \ni \sigma_C &\triangleq \mathsf{Addr} \xrightarrow{\mathsf{fin}} \mathsf{Cell} \\
\mathsf{Func} \ni F &::= \mathsf{Fun}(\vec{x}, c)
\end{aligned}
$$

Figure 4.3: The formal syntax of λ_C.

- The call is internal into p_1. Since $p_1 \subseteq p_1 \cup p_2$, the called function is also present in the larger program. Correctness thus is achieved using the inductive hypothesis.
- The call is external, justified using Ψ_2. Then it is to a function in p_2, which is also present in the larger program. The correctness result follows from $(\Psi_1 \sqcup \Psi_{\mathsf{axiom}}) \vdash p_2 : \Psi_2$, using WP-BIND and the inductive hypothesis for both.
- The call is external, justified using Ψ_{axiom}. Since we still have Ψ_{axiom} as the external call protocol, we are done. $\square$

The previously explained lemma is achieved by setting $\Psi_{\mathsf{axiom}}. \triangleq \bot$. By Ψ_{axiom} ## $(\mathsf{dom}\ p_1 \cup \mathsf{dom}\ p_2)$, we denote that Ψ_{axiom} does not assert any behavior about calls defined in p_1 or p_2. This theorem is already a kind of lifting lemma that enables language-local reasoning. Although it does not yet involve different languages, it already demonstrates how we want to later lift verification of individual parts of a (multi-language) program to the correctness of the whole program. When linking a multi-language program, we want essentially the same theorem as Theorem 4.4, but for different languages. Indeed, Theorem 6.4, proven in Section 6.1.2, provides precisely this.

4.2 λ_C

We can now define our formal version of C, as an instance of a language with external calls as defined in the previous section. The formal syntax of λ_C can be found in Figure 4.3, while the operational semantics can be found in Figure 4.4. The language, including function calls, is evaluated *left-to-right*.

Our language features four kinds of *values*: Integers, locations, the null pointer

$$\textsc{SCLet}\quad \frac{}{(\mathsf{let}\ x = w\ \mathsf{in}\ c, \sigma_C) \longrightarrow_C (c[w/x], \sigma_C)}$$

$$\textsc{SCUnOp}\quad \frac{\ominus w = w_r}{(\ominus w, \sigma_C) \longrightarrow_C (w_r, \sigma_C)}$$

$$\textsc{SCBinOp}\quad \frac{w \otimes w' = w_r}{(w \otimes w', \sigma_C) \longrightarrow_C (w_r, \sigma_C)}$$

$$\textsc{SCIfTrue}\quad \frac{w \text{ is truthy}}{(\mathsf{if}\ w\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma_C) \longrightarrow_C (c_1, \sigma_C)}$$

$$\textsc{SCIfFalse}\quad \frac{w \text{ is not truthy}}{(\mathsf{if}\ w\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma_C) \longrightarrow_C (c_2, \sigma_C)}$$

$$\textsc{SCWhile}\quad \frac{}{(\mathsf{while}(c_1)\ c_2, \sigma_C) \longrightarrow_C (\mathsf{if}\ c_1\ \mathsf{then}\ c_2; \mathsf{while}(c_1)\ c_2\ \mathsf{else}\ 0, \sigma_C)}$$

$$\textsc{SCFunCall}\quad \frac{p[fn] = fn(\vec{x}) := c \qquad |\vec{w}| = |\vec{x}| = n}{(\mathsf{call}\ fn\, \vec{w}, \sigma_C) \longrightarrow_C (c[w_1/x_1, \ldots, w_n/x_n], \sigma_C)}$$

$$\textsc{SCStore}\quad \frac{\sigma_C[a] \neq \mathsf{None}, \dagger}{(*a \leftarrow w, \sigma_C) \longrightarrow_C (0, \sigma_C[a := w])}$$

$$\textsc{SCMalloc}\quad \frac{n > 0 \qquad \forall 0 < i < n.\ a + i \notin \mathsf{dom}\ \sigma_C}{(\mathsf{malloc}(n), \sigma_C) \longrightarrow_C (a, \sigma_C[a + 0 := \star, \ldots, a + n - 1 := \star])}$$

$$\textsc{SCLoad}\quad \frac{\sigma_C[a] = w}{(*a, \sigma_C) \longrightarrow_C (w, \sigma_C)}$$

$$\textsc{SCFree}\quad \frac{\forall 0 \leqslant i < n.\ \sigma_C[a + i] \neq \mathsf{None}, \dagger}{(\mathsf{free}(a, n), \sigma_C) \longrightarrow_C (a, \sigma_C[a + 0 := \dagger, \ldots, a + n - 1 := \dagger])}$$

Figure 4.4: The head reduction rules of the operational semantics $\longrightarrow_C$ of λ_C.

constant `NUL`, and function pointers (which simply are strings, denoting the name of a function). Unlike in actual C, arithmetic operations do not overflow. We implement most of C's arithmetic and logical operators, which have undefined behavior when not used properly (*e.g.*, when multiplying pointers). Notably, our pointers allow address arithmetics, specifically both pointer-offset and pointer-difference.[1] Like in C, there are no first-class booleans, instead, 0 and null pointers are considered false, all other values are considered true (we say they *are truthy*), and comparison operators appropriately evaluate to 0 or 1. Additionally, our language is now *expression-based*, eliding statements. The `if`-statement is now subsumed by the `?:` ternary expression, which we rename into if for easier reading. Blocks and local variables are implemented using `let-in`, where possible. Sequencing is also implemented using `let-in`, but we may use the syntactic sugar $c_1;c_2$. Loop expressions evaluate to 0, since they are only evaluated for their side effects.

While C has a memory model based on bytes, where a word is stored by decomposing it into several bytes, the *memory model* of λ_C is word-based, with each address storing a value (an integer or a location). Additionally, each cell can be in one of two special states: It can be uninitialized, indicated by $\star$. Memory cells are allocated uninitialized, and must first be initialized by writing to them. Further, deallocated memory cells are set to $\dagger$, which makes all access to the memory cells undefined behavior. The cell is not removed from the heap to ensure that no two freshly allocated locations ever compare equal.[2] The expression $\mathsf{free}(c_1, c_2)$ takes two arguments. The first is the pointer to the first location to be freed, and the second is the number of locations do be deallocated. This allows us to not track the size of allocations (unlike malloc in regular C), and instead shift this burden to the programmer. To simplify things, if this number is $\leqslant 0$, nothing happens.

λ_C does not handle structs or unions. Instead, there is only pointer addition, so structs must be implemented by allocating a large chunk of memory and accessing it using offsets. Handling local struct variables is more difficult. In general, properly translating a C program with local variables to λ_C can be more difficult than one initially suspects. While variables that are assigned once are easy to handle, variables that have their address taken, or that are modified repeatedly in a loop, are more difficult to handle. Our solution here is to *heapify* those variables, after running an SSA-style transformation to find the variables that actually need this. Concretely, we translate a variable declaration to a heap allocation, and insert a call to free() at the point the variable would go out of scope. Accessing and modifying the variable is then implemented using loads and stores, while taking its address becomes easy. This might seem unorthodox, but it actually is a faithful implementation of the rules

[1] We also ignore provenance.

[2] Additionally, the default ghost state for heaps in Iris does not actually support deallocation. We thus use this trick, which is also used by Iris' demonstration toy language (HeapLang).

$$\textsc{CWP-Malloc}\quad \frac{n > 0 \qquad \mathop{\ast}_{0 \leqslant i < n} a + i \mapsto_C \_ \twoheadrightarrow \mathsf{wp}\ a \mathrel{@} p, \Psi \{Q\}}{\mathsf{wp}\ \mathsf{malloc}(n) \mathrel{@} p, \Psi \{Q\}}\ast$$

$$\textsc{CWP-Free}\quad \frac{\mathop{\ast}_{0 \leqslant i < n} a + i \mapsto_C \_ \qquad \mathop{\ast}_{0 \leqslant i < n} a + i \mapsto_C \dagger \twoheadrightarrow \mathsf{wp}\ a \mathrel{@} p, \Psi \{Q\}}{\mathsf{wp}\ \mathsf{free}(a, n) \mathrel{@} p, \Psi \{Q\}}\ast$$

$$\textsc{CWP-Load}\quad \frac{a \mapsto_C{}^{d}\ w \qquad a \mapsto_C{}^{d}\ w \twoheadrightarrow \mathsf{wp}\ w \mathrel{@} p, \Psi \{Q\}}{\mathsf{wp}\ {*a} \mathrel{@} p, \Psi \{Q\}}\ast \qquad \textsc{CWP-Store}\quad \frac{a \mapsto_C \_ \qquad a \mapsto_C w \twoheadrightarrow \mathsf{wp}\ 0 \mathrel{@} p, \Psi \{Q\}}{\mathsf{wp}\ {*a} \leftarrow w \mathrel{@} p, \Psi \{Q\}}\ast$$

$$\textsc{CWP-Call}\quad \frac{p[fn] = fn(\vec{x}) := c \qquad |\vec{w}| = |\vec{x}| = n \qquad \mathsf{wp}\ c[w_1/x_1, \ldots, w_n/x_n] \mathrel{@} p, \Psi \{Q\}}{\mathsf{wp}\ \mathsf{call}\ fn\, \vec{w} \mathrel{@} p, \Psi \{Q\}}\ast$$

Figure 4.5: Program Logic Rules for C.

outlined in the C standard for *automatic storage duration* [17, 6.2.4p6].

Finally, C programs include *function calls*, which we already explained in Section 4.1. We use the existing machinery described there, which constructs programs $p : \mathsf{Prog}$ as lists of functions. This program is then also a parameter of the step relation. The rule SCFunCall already includes the definition of $\mathsf{applyFunc}$, unfolded, so that this rule is compatible with the abstract version SFunCall. The actual definition of $\mathsf{applyFunc}$ is as follows:

$$\mathsf{applyFunc}(\mathsf{Fun}(\vec{x}, c), \vec{w}) \triangleq \begin{cases} c\,[w_1/x_1, \ldots, w_n/x_n] & |\vec{x}| = |\vec{w}| \\ \mathsf{None} & \text{otw.} \end{cases}$$

The *program logic* is quite similar to the one presented in Section 2.4, but uses the weakest precondition with (external) functions calls discussed in Section 4.1. We have the C points-to $a \mapsto_C w$, which also has the variant $\mapsto_C \_$ (which includes uninitialized data) and $\mapsto_C \dagger$ (indicating that the heap cell was deallocated). Further, it is fractional, so that $a \mapsto_C^d w$ denotes that we only have a fraction of it. If $d = \Box$, then it is persistent and immutable. Our program logic, shown in Figure 4.5, now is a rather straightforward combination of the concepts presented in the previous few sections. The rules for accessing memory are straightforward generalizations of the ones shown in Figure 2.4. The CWP-Call rule is also just a restatement of the general WP-Call-Internal we saw in Figure 4.1. Similarly, the rule WP-Call-

$$
\begin{aligned}
\mathsf{Val} \ni V &::= (n : \mathbb{Z}) \mid (\ell : \mathsf{Loc}) \mid \mathsf{true} \mid \mathsf{false} \mid \langle\rangle \mid \langle V, V\rangle \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid \mathsf{rec}\, f\, x.\, e \mid \textcircled{\iota} \\
\ominus &::= - \mid\ ! \\
\otimes &::= + \mid - \mid \times \mid \div \mid \% \mid \& \mid \,|\, \mid \hat{} \mid \ll \mid \gg \mid < \mid \leqslant \mid = \\
\mathsf{Expr} \ni e &::= V \mid x \mid \mathsf{rec}\, f\, x.\, e \mid e\ e \mid \ominus e \mid e \otimes e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
&\quad \mid \langle e, e\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl} \Rightarrow e \mid \mathsf{inr} \Rightarrow e \\
&\quad \mid \mathsf{alloc}\, e\ \mathsf{of}\ e \mid e.(e) \mid e.(e) \leftarrow e \mid \mathsf{length}\, e \mid \mathsf{call}\ fn\, \vec{e} \\
\mathsf{Block} \ni bl &::= \vec{V} \mid \lightning \\
\Sigma_{ML} \ni \sigma_{ML} &\triangleq \mathsf{Loc} \xrightharpoonup{\text{fin}} \mathsf{Block} \\
\mathsf{Func} &\triangleq \bot
\end{aligned}
$$

Figure 4.6: The formal syntax of λ_{ML}.

External presented there is also applicable to this program logic, but we do not restate it, since it looks exactly the same.

4.3 λ_{ML}

We now define λ_{ML}, which also is a language with external calls as presented in Section 4.1. The formal syntax of λ_{ML} can be found in Figure 4.6, while the operational semantics can be found in Figure 4.7.[3] The language, including function applications, is evaluated *right-to-left*. The list of operators is similar to that of C. There also is no integer overflow. Logical operators are overloaded to also operate on booleans. Unlike in C, there is *no address arithmetic*.

It is immediately obvious that λ_{ML} has much more structured values than C. Indeed, it has first-class booleans, pairs, and tagged unions. Further, it is *higher-order*, featuring first-class recursive closures. It also has foreign values, which are used to implement foreign blocks when interoperating with C. These values can only be used opaquely in λ_{ML}, *i.e.*, they can be be passed as the arguments of functions, but they can not be inspected. Looking at the expressions, we can see that there are some expressions that look like values, namely $\langle \cdot , \cdot \rangle$, inl , inr and $\mathsf{rec}\, f\, x.\, e$. The difference between the expression and value variant is that the expressions allow reduction of sub-expressions, so that $\mathsf{inl}\ (1 + 2)$ evaluates to the value $\mathsf{inl}\ 3$. Another difference is quite subtle: The expressions allow their sub-expressions to be substituted. Values, on the other hand, are always opaque for substitution. This is particularly relevant for closures: While the closure is still an expression, values can be substituted into it. Once the closure itself is evaluated, it reduces to the value version of itself, which can no longer *capture* anything (and should not have any

[3]In fig. 4.7, write σ instead of σ_{ML} to save space.

SMBeta
$$\frac{}{((\mathsf{rec}\, f\, x.\, e)\ V, \sigma) \longrightarrow_{\mathtt{ML}} (e[\mathsf{rec}\, f\, x.\, e/f, V/x], \sigma)}$$

SMClosure
$$\frac{}{(\mathsf{rec}\, f\, x.\, e, \sigma) \longrightarrow_{\mathtt{ML}} (\mathsf{rec}\, f\, x.\, e, \sigma)}$$

SMInl
$$\frac{}{(\mathsf{inl}\ V, \sigma) \longrightarrow_{\mathtt{ML}} (\mathsf{inl}\ V, \sigma)}$$

SMInr
$$\frac{}{(\mathsf{inr}\ V, \sigma) \longrightarrow_{\mathtt{ML}} (\mathsf{inr}\ V, \sigma)}$$

SMLCaseL
$$\frac{}{(\mathsf{case\ inl}\ V\ \mathsf{of\ inl} \Rightarrow V_l \mid \mathsf{inr} \Rightarrow V_r, \sigma) \longrightarrow_{\mathtt{ML}} (V_l\ V, \sigma)}$$

SMLCaseR
$$\frac{}{(\mathsf{case\ inr}\ V\ \mathsf{of\ inl} \Rightarrow V_l \mid \mathsf{inr} \Rightarrow V_r, \sigma) \longrightarrow_{\mathtt{ML}} (V_r\ V, \sigma)}$$

SMPair
$$\frac{}{(\langle V, V' \rangle, \sigma) \longrightarrow_{\mathtt{ML}} (\langle V, V' \rangle, \sigma)}$$

SMFst
$$\frac{}{(\mathsf{fst}\ \langle V, V' \rangle, \sigma) \longrightarrow_{\mathtt{ML}} (V, \sigma)}$$

SMSnd
$$\frac{}{(\mathsf{snd}\ \langle V, V' \rangle, \sigma) \longrightarrow_{\mathtt{ML}} (V', \sigma)}$$

SMIfTrue
$$\frac{}{(\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2, \sigma) \longrightarrow_{\mathtt{ML}} (e_1, \sigma)}$$

SMIfFalse
$$\frac{}{(\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2, \sigma) \longrightarrow_{\mathtt{ML}} (e_2, \sigma)}$$

SMUnOp
$$\frac{\ominus V = V'}{(\ominus V, \sigma) \longrightarrow_{\mathtt{ML}} (V', \sigma)}$$

SMBinOp
$$\frac{V \otimes V' = V_r}{(V \otimes V', \sigma) \longrightarrow_{\mathtt{ML}} (V_r, \sigma)}$$

SMLength
$$\frac{\sigma[\ell] = \mathsf{Some}\, \vec{V}}{(\mathsf{length}\, \ell, \sigma) \longrightarrow_{\mathtt{ML}} (|\vec{V}|, \sigma)}$$

SMLoad
$$\frac{\sigma[\ell] = \mathsf{Some}\, \vec{V} \qquad V_n = V'}{(\ell.(n), \sigma) \longrightarrow_{\mathtt{ML}} (V', \sigma)}$$

SMLoadOOB
$$\frac{\sigma[\ell] = \mathsf{Some}\, \vec{V} \qquad n < 0 \vee n \geqslant |\vec{V}|}{(\ell.(n), \sigma) \longrightarrow_{\mathtt{ML}} (\ell.(n), \sigma)}$$

SMStore
$$\frac{\sigma[\ell] = \mathsf{Some}\, \vec{V}}{(\ell.(n) \leftarrow V', \sigma) \longrightarrow_{\mathtt{ML}} (\langle\rangle, \sigma[\ell := \vec{V}[n := V']])}$$

SMStoreOOB
$$\frac{\sigma[\ell] = \mathsf{Some}\, \vec{V} \qquad n < 0 \vee n \geqslant |\vec{V}|}{(\ell.(n) \leftarrow V', \sigma) \longrightarrow_{\mathtt{ML}} (\ell.(n) \leftarrow V', \sigma)}$$

SMAlloc
$$\frac{\ell \notin \mathsf{dom}\ \sigma}{(\mathsf{alloc}\, n\ \mathsf{of}\ V, \sigma) \longrightarrow_{\mathtt{ML}} (\ell, \sigma[\ell := \underbrace{[V, \ldots, V]}_{n \text{ entries}}])}$$

SMAllocOOB
$$\frac{n < 0}{(\mathsf{alloc}\, n\ \mathsf{of}\ V, \sigma, \sigma) \longrightarrow_{\mathtt{ML}} (\mathsf{alloc}\, n\ \mathsf{of}\ V, \sigma, \sigma)}$$

Figure 4.7: The head reduction rules of the operational semantics $\longrightarrow_{\mathtt{ML}}$ of $\lambda_{\mathtt{ML}}$.

free variables). This implements how closures capture their environment. This also explains the reduction rules SMInl, SMInr, SMPair, and SMClosure, which perform the expression-to-value conversion step. Except for in these rules, all other ambiguous uses of expressions/values in the reduction rules should be read as values. Our language does not feature loops, since it already has recursive higher-order functions. The case construct also requires some explanation: Its two match arms must be of function type, and the case selects the appropriate one, applying to it the value stored in the disjoint union. We nonetheless often use the syntactic sugar $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e \mid \mathsf{inr}\ x \Rightarrow e$, implemented using closures. Further syntactic sugar are first-order closures $\lambda x.\, e$ let-in expressions $\mathsf{let}\ e = x\ \mathsf{in}\ e$, and sequencing $e; e'$, all implemented using closures. We can also add an assert expression, by using an if that either does nothing if the condition is true, or otherwise step to an expression with undefined behavior, like $\langle\rangle + 1$.

The *memory model* of λ_{ML} is block-based. Each location stores a block, which has a fixed length, and which then stores several values. The memory can store all structured values. We say that the memory stores arrays, and call one-element arrays references. Read and write accesses require a location ℓ and an offset n. If the offset is not inbounds, the memory access silently diverges[4] (see the SMLoadOOB and SMStoreOOB rules). Allocation similarly diverges if the size of the new block is not positive (see SMAllocOOB). Otherwise, it allocates a new reference, where all fields are initialized using the second argument V of $\mathsf{alloc}\ n\ \mathsf{of}\ V$. We consider the language to be *garbage-collected*, hence there is operation for freeing blocks. So far, the garbage collector is merely conceptual (*i.e.*, the operational semantics implement a no-op garbage collector). Instead of a block, the memory might also store $\lightning$ at a location. This value is never produced by the semantics, its only use is to later enable interoperability with C.

Finally, our syntax also includes a construct for function calls $\mathsf{call}\ fn\ \vec{e}$. Since we are a language with external calls, the framework also provides us with machinery for internal calls and programs as lists of functions. We do not use that machinery, by setting $\mathsf{Func} \triangleq \bot$, which expresses that λ_{ML} functions *do not exist*, because the linking Thus, the only valid λ_{ML} program is the empty program $\varnothing$. Therefore, all function calls are external calls. The machinery for programs and internal calls is not used, since this implies a linkage model that is not actually used by OCaml. Instead, the linkage model based on "one big expression" is more appropriate, since λ_{ML} already includes first-class recursive closures, which can all be combined using a large let-in.

Note that while actual OCaml has external call declarations, which declare an OCaml function that is implemented as an external call, λ_{ML} has external call expressions. We do not model the declarations, but instead inline them at the call site, by using an

[4]Throwing an error would be more appropriate. However, we do not support exceptions.

external call expression. Further, while the external call declaration includes types, the external call expressions do not. In general, the operational model of OCaml does not feature a type system. In Section 4.5, we define a type system and a logical relation. This type system includes the ability to give types to external calls, in a way resembling the external call declaration.

Notable *OCaml features not modeled* are exceptions and (co)inductive datatypes. While the later type system supports recursive types, it does not support defining custom inductive types with custom constructors. The only constructors available are pairs and tagged unions. While this does not limit the expressivity of the language (as these can encode all inductive datatypes), this matters for the Foreign Function Interface, since inductive datatypes get special treatment. Similarly, the Foreign Function Interface supports raising and catching exceptions. By eliding them, we restrict ourself to not considering exceptions in our formal model.

We already discussed how this affects our model of the FFI in Section 3.5.

We again summarize the three *features included for interoperability*. The first are external function calls, which simply cause undefined behavior in isolated $\lambda_{\mathtt{ML}}$. Later, these can be used to call $\lambda_{\mathtt{C}}$ functions. Further, we have foreign values $\textcircled{\iota}$, using a foreign ID ι. These are later be used to embed $\lambda_{\mathtt{C}}$ data. Finally, there is the special memory cell $\lightning$, which is included for technical reasons only.[5] In Chapter 7, we discuss different approaches where no such special heap cell is needed.

The reason we need special syntax in $\lambda_{\mathtt{ML}}$ to express external function calls is obvious: actual OCaml also has special syntax for external calls.

The motivation for foreign values is less obvious. They are necessary since we must define all available values when formalizing OCaml. In actual OCaml, one can simply declare a new type, by writing `type buffer` and not giving a definition. This type is then existential, and the inhabitants can be defined as an implementation invariant (by using a semantic type). In actual OCaml, one can use the fact that OCaml borrows its at-runtime[6] notion of values from the OCaml runtime,[7] so that one can use as value everything the runtime supports. Because we wish to include support for custom blocks in our formal runtime model, foreign values provide an "escape hatch" that allows us to have a value simply be the location of a block, while being opaque to OCaml.

In $\lambda_{\mathtt{C}}$, which does not feature a runtime, and which already has a rather pedestrian system of values, new "kinds of values" are usually represented as pointers, which are not even required to point to a location on the $\lambda_{\mathtt{C}}$ heap. Since in $\lambda_{\mathtt{C}}$, pointers

[5]One could call it a dirty hack.
[6]Runtime values are the values that exist while the program runs.
[7]The runtime is the system that executes OCaml bytecode

are already unsafe, such an encoding is not an issue. The foreign values $\textcircled{\iota}$ can be understood as adding an unsafe kind of pointer to $\lambda_{\mathtt{ML}}$, since regular locations ℓ are unsuitable, as they are always expected to point to an actual heap element.

We also build a *program logic* for $\lambda_{\mathtt{ML}}$. The points-to for $\lambda_{\mathtt{ML}}$, $\ell \mapsto_{\mathrm{ML}} \vec{V}$, is defined with a location ℓ on the left, and a list of values $\vec{V}$ on the right. Such a points-to conveys ownership of an entire block. Unlike in Cosmo [27], this ownership can not natively be divided into ownership over individual entries in this block.[8] It is, however, fractional insofar as that we can have half ownership of the entire block. The fractional variant is denoted as $\ell \mapsto^{d}_{\mathrm{ML}} \vec{V}$. Again, if $d = \square$, then it is persistent and immutable.

Like for $\lambda_{\mathtt{C}}$, the weakest precondition for $\lambda_{\mathtt{ML}}$ is the one with support for external calls. But unlike before, the only $\lambda_{\mathtt{ML}}$ program is the empty program, thus one parameter is fixed to $\varnothing$.

4.4 Examining the Differences

When looked at from afar, the two languages we just presented can look quite alike. They are both expression-based, have load, store, and allocation instructions, and almost all unary and binary operators are available in both languages. But we already explained in Chapter 3 that these languages are very different. In fact, the entire chapter described the OCaml-C FFI, which is needed to bridge the differences. So, what are these differences?

There are also some differences we consider *trivial*: One such fact is that $\lambda_{\mathtt{C}}$ is evaluated left-to-right, while $\lambda_{\mathtt{ML}}$ is evaluated the other way around. This is because the language does not fundamentally change when this is flipped. In particular, such properties do not affect linking. Instead, we only care about the differences that make these languages ABI-incompatible.

This brings us to the first difference: The memory model of $\lambda_{\mathtt{C}}$ is addressed using addresses Addr, while in $\lambda_{\mathtt{ML}}$, these are locations Loc. This is not just a difference in name, since $\lambda_{\mathtt{C}}$ locations support address arithmetic, which is undefined for $\lambda_{\mathtt{ML}}$ locations.

Besides, the type of *heap cells* is different for both–the second difference. Each $\lambda_{\mathtt{C}}$ heap cell stores a $\lambda_{\mathtt{C}}$ value, but can also be uninitialized or deallocated. In $\lambda_{\mathtt{ML}}$, there are no deallocated or uninitialized heap locations. Additionally, each heap cell stores a list of $\lambda_{\mathtt{ML}}$ values,[9] instead of just a single value. In other words, it is an array.

Thirdly, the memory of $\lambda_{\mathtt{ML}}$ is *garbage-collected*. While the Garbage Collector is implicitly defined such that it never deallocates a value that could still be used in

[8] Iris should be strong enough to define such ghost state on an ad-hoc basis, if needed.

[9] It can also store $\lightning$, but we do not consider this a proper part of $\lambda_{\mathtt{ML}}$

$\lambda_{\mathtt{ML}}$, there is no such guarantee for potential $\lambda_{\mathtt{C}}$ code interacting with it.

Fourthly, the *values* of both languages are completely different. In $\lambda_{\mathtt{C}}$, the value model is flat, featuring only integers and pointers/addresses. In $\lambda_{\mathtt{ML}}$, there also are integers and locations, but also more structured values, *i.e.*, pairs and sums.

Finally, the function call mechanism for $\lambda_{\mathtt{C}}$ and $\lambda_{\mathtt{ML}}$ are completely different. In $\lambda_{\mathtt{C}}$, there is a fixed list of functions, that can be called using their name. In $\lambda_{\mathtt{ML}}$, functions are implemented as closures, which are a particular kind of value. Functions are not called via their name, instead a closure can be applied to an argument. For this, the closure must be explicitly passed to the function that wants to use it.

This also affects the model of linking in both programs. In $\lambda_{\mathtt{C}}$, two programs are linked by merging their set of functions, as described by Theorem 4.4. In $\lambda_{\mathtt{ML}}$, linking is not as well-defined, since the entire program is just one big expression. There, different parts of the program are linked by just combining them into the one big expression, *e.g.*, using a large let-in.

4.5 A Logical Relation for $\lambda_{\mathtt{ML}}$

To show that $\lambda_{\mathtt{ML}}$ is a type-safe language, we build a logical relation for $\lambda_{\mathtt{ML}}$. This logical relation is almost entirely standard, and in large parts based on the one presented in a paper by Timany *et al.* [47].

We make three orthogonal changes to the logical relation presented in there. First, we change the invariants to non-atomic invariants, since we do not consider concurrent programs. Second, we need to do some slight refactoring to work around the rules no longer available with Transfinite Iris (see Section 2.4.1). Finally, we include typing rules for external calls, which is inspired by the typing annotations of external calls used in regular OCaml.

To save space, we only describe our changes to the logical relation presented by Timany *et al.* [47]. We recommend the reader compare our definitions to their original definitions, presumably by opening their work side-by-side. First, we assume that we are, for the remainder of this chapter, given a specification of external calls Ψ, which parametrizes this entire development.

The changes required for *non-atomic invariants* are minimal: We change all invariants to be non-atomic, and further change the expression relation as follows:

$$[\![\tau]\!]^{\mathsf{e}}_{\delta} = \lambda e.\, [\mathsf{NaInv} : \top] \twoheadrightarrow \mathsf{wp}\ e\ @\ \varnothing, \Psi\, \{V.\, [\mathsf{NaInv} : \top] * [\![\tau]\!]_{\delta}\ V\}$$

This means that for semantic types defined using invariants, these invariants can be opened for longer if the typing rule covers an expression that takes several steps to execute. In particular, if the expression is an external call, the invariant could be open for that entire external call.

Before we get to the interesting change, external calls, we note that we needed to change the semantic type of locations, to *work around Transfinite Iris*. Specifically, instead of the single invariant presented by Timany *et al.* [47, Fig. 5, p20], we have the following semantic type:

$$
\begin{aligned}
[\![\mathsf{array}\ \tau]\!]_\delta \triangleq \lambda V.\, \exists \gamma\, \ell.\, (V = \ell) * {} & \\
& \boxed{\exists \vec{V}.\, \gamma \overset{👻}{\mapsto}_{1/2} \vec{V} * \mathop{\Large\ast}_{V_i \in \vec{V}} [\![\tau]\!]_\delta V_i}^{\mathcal{N}_{\texttt{typing}}.\ell} * \\
& \boxed{\exists \vec{V}.\, \gamma \overset{👻}{\mapsto}_{1/2} \vec{V} * \ell \mapsto_{\mathrm{ML}} \vec{V}}^{\mathcal{N}_{\texttt{timeless}}.\ell}
\end{aligned}
$$

Notably, we split the invariants in two, one storing the points-to, the other storing the proof that the value is well-typed. Both are connected using a ghost variable, so that if both are open, we know the stored values agree. The reason we do so is to ensure that the contents of the latter are *timeless*, which allows us to crucially eliminate a later $\triangleright$. Before, this split was not necessary since one could use the LaterSep and LaterExists rules to push the later inwards, until it was in front of a timeless proposition. By using Transfinite Iris, such transformations become impossible.

Another difference there is that we now have arrays, instead of just single references. Thus, the semantic type contains a big separated conjunction, ensuring that all members of the array are well-typed. Since we modified the semantics to not have undefined behavior on out-of-bound accesses (see rules SMAllocOOB, SMLoad-OOB, and SMStoreOOB), we are able to verify all well-typed memory accesses, even though the typing rules (not shown here) allow any integer (including negative ones) to appear as the size/offset.

We now turn to the part actually relevant for this thesis, the *typing rules for external functions*.

First, we introduce *program contexts* $P \triangleq \mathsf{string} \overset{\text{fin}}{\rightharpoonup} \mathrm{ProgType}$, where $\mathrm{ProgType}$ is the (Coq) type of all external program types. These types, similar to plain function types, describe the number and types of the function arguments, as well as their return type. They differ from plain function types since external calls are not cascading, but instead pass several arguments at once. Their formal definition is as follows:

$$
\mathrm{ProgType} \ni T_P ::= \mathsf{FunType}(\vec{\tau}, \tau)
$$

Next, we modify the typing judgment. Timany *et al.* [47, Fig. 2, p8] had typing judgments of shape $\Delta, \Gamma \vdash e : \tau$, where $\tau : \mathit{Type}$ is a type, and Δ and Γ are the contexts tracking free type variables and assigning types to regular variables, respectively. In Coq, the context Δ is implicit, since types are mechanized using de Bruijn indices [6], in particular using AutoSubst [42].

The typing judgement then gains such a program context as an additional parameter, becoming $P, \Delta, \Gamma \vdash e : \tau$. The program context is uniform, that is, no typing judgement modifies it before typing a subexpression. The only non-trivial use is in the rule for typing an external call $\mathsf{call}\ \mathsf{fn}\ \vec{e}$:

$$\frac{P[\mathsf{fn}] = \mathsf{FunType}(\vec{\tau}_a, \tau_r) \qquad |\vec{\tau}_a| = |\vec{e}| \qquad \forall i.\ P, \Delta, \Gamma \vdash e_i : \tau_{a,i}}{P, \Delta, \Gamma \vdash \mathsf{call}\ \mathsf{fn}\ \vec{e} : \tau_r}\ \textsc{T-ExtCall}$$

Informally, this rule requires that a type of that function is present, that the number of arguments matches, that all arguments are correctly typed, and that the return type is as specified.

The key insight now is that interpreting this function type environment semantically induces a *protocol*. Given a function type environment P, we define the protocol Ψ_P as follows:

$$\begin{aligned}\Psi_P\ \mathsf{fn}\ \vec{V}\ Q \triangleq \exists \vec{\tau}_a\ \tau_r.\ & P[\mathsf{fn}] = \mathsf{FunType}(\vec{\tau}_a, \tau_r) * |\vec{V}| = |\vec{\tau}_a| * \\ & [\mathsf{NaInv} : \top] * \left(\forall V'.\ [\![\tau_r]\!]_\delta V' * [\mathsf{NaInv} : \top] \mathrel{-\!\!*} Q(V')\right) * \\ & \mathop{\LARGE *}_{V_i \in \vec{V}} [\![\tau_{a,i}]\!]_\delta V_i\end{aligned}$$

This protocol simply states that the arguments have the correct number, that each argument has the correct type, and that the returned value also inhabits the return type. Additionally, it also embeds the token for non-atomic invariants, which allows an external call to open invariants for the duration of the entire call.

The definition of being semantically well-typed is now changed as follows:

$$P, \Delta, \Gamma \vDash e : \tau \triangleq \delta\ \vec{V}.\ \mathsf{dom}\ \Delta \subseteq \mathsf{dom}\ \delta \mathrel{-\!\!*} [\![\Gamma]\!]^c_\delta \mathrel{-\!\!*} \Psi_P \sqsubseteq \Psi \mathrel{-\!\!*} [\![\tau]\!]^c_\delta(e[\vec{V}/\vec{x}])$$

The only change is that we additionally include that Ψ_P is included in the specification Ψ. Intuitively, this means that the external calls described in P are valid external calls according to Ψ. In particular, Ψ entails that for each type $\mathsf{FunType}(\vec{\tau}_a, \tau_r)$ ascribed to an external call using P, when invoked with arguments having the proper types $\vec{\tau}_a$, it returns a value inhabiting the proper return type τ_r.

With these changes, we again validate the *fundamental theorem* for this logical relation:

Theorem 4.5 (Fundamental Theorem for λ_{ML})

$$\frac{P, \Delta, \Gamma \vdash e : \tau}{P, \Delta, \Gamma \vDash e : \tau}$$

Proof By induction on the typing judgment, as outlined by Timany *et al.* [47]. The case of T-ExtCall is handled using Ψ_P and WP-Call-External. □

With this fundamental theorem, we know that well-typed programs do not go wrong. Further, we know that well-typed programs do only make "well-typed external calls," *i.e.*, that all they only call external functions using a type. We use this later to argue that functions defined in C are safe to call from OCaml.

Note that the entire development of program typing and induced protocols is not strictly necessary. We could (and in fact later sometimes do this) simply not have any typing rules for external calls, and instead only show that external calls are semantically well-typed. We still include the program typing context, since it gives a formal foundation for the typing annotations found in actual OCaml.

Chapter 5

The Combined Operational Semantics

In this chapter, we develop operational semantics to model the OCaml FFI described in Chapter 3. To do so, we develop modules, which generalize the languages of Section 4.1. A key difference between modules and languages is that modules support for angelic non-determinism. Unfortunately, introducing angelic non-determinism makes head redexes harder to characterize formally. Additionally, some of the constructions outlined in this chapter become harder to state when they have to properly define head-redexes. We side-step issues related to this by simply removing the concept of head redexes. This simplification is possible since properly accounting for angelic non-determinism allows us to no longer care about whether an expression is (head)-reducible. We first (in Section 5.1) discuss what a module is, and how modules allow angelic non-determinism. Then, Section 5.2 defines the *wrapper*, which takes an OCaml program, and makes it ABI-compatible with C. This wrapper is thus our formal model of the runtime. This also wrapper is also a module, which *wraps* the $\lambda_{\texttt{ML}}$ semantics to something that is ABI-compatible with $\lambda_{\texttt{C}}$.

Note that we take this formal model as a ground truth. In particular, we do not attempt to verify that some actual implementation of an OCaml runtime is correct. Instead, this work can be understood as searching for a definition what it even means to be a correct runtime.

5.1 Modules and Angelic Non-determinism

In Section 5.2, will define a formal model of the OCaml-FFI mechanism, including the translation between block-level and high-level values. For reasons we explain there, we need angelic non-determinism.

What Is Angelic Non-Determinism?

We have previously seen demonic non-determinism, which informally describes a situation where the program behavior is not precisely constrained. Instead, we (as the programmer) only know that one of several options will happen, but we do not know which. Thus, we must write our program so that we can handle all possible

$$\textsc{StepAngelic}\qquad\qquad\qquad\qquad \textsc{StepDemonic}$$

$$\frac{m \in X}{n \twoheadrightarrow_{\mathsf{angelic}} X} \qquad\qquad \frac{\forall m.\, m \in X}{n \twoheadrightarrow_{\mathsf{demonic}} X}$$

$$\begin{tikzcd} & \vdots & \\ & \{2\} \arrow[r] & 2 \\ 0 \arrow[ur] \arrow[r] \arrow[dr] & \{1\} \arrow[r] & 1 \\ & \{0\} \arrow[r] & 0 \end{tikzcd} \qquad\qquad \begin{tikzcd} & & \vdots \\ & & 2 \\ 0 \arrow[r] & \mathbb{N} \arrow[ur] \arrow[r] \arrow[dr] & 1 \\ & & 0 \end{tikzcd}$$

Figure 5.1: Two example relations, demonstrating multi-relation semantics, along with their branching trees.

cases, and to argue correctness, we must argue it for each possible choice. Angelic non-determinism is considerably more obscure. In an angelically non-deterministic choice, there also are several options, but the program gets to choose the value it likes most. To argue correctness, it in particular suffices to only argue correctness for one of the possible choices. Since angelic non-determinism is more obscure, it can be hard to find examples of it. Readers with familiarity in foundational systems programming might be aware of pointer provenance, which is sometimes modeled using angelic non-determinism: In some languages, pointers keep track of their origin, called provenance, and can only be used to access memory locations related to that origin. When one now introduces integer-pointer casts, one needs to specify which provenance is associated to the result of such a cast. This provenance is then sometimes specified to be chosen angelically. For us, angelic non-determinism becomes necessary in Section 5.2. To support this, we need to adapt some of the machinery we created in Section 4.1, like the definition of a weakest precondition. We first discuss this machinery, and then construct the wrapper on top of it.

Angelic Non-Determinism, Formally

To formally model non-determinism, we introduce a new kind of language, that we call a *module*. Like a language, a module has a notion of values, of expressions, of states, and also has a step relation. The notion of external calls and programs as sets of functions applies similarly. What changes with modules is that they used multi-relation-based semantics [24], also taking inspiration from DimSum [41]. Whereas previously, the step relation related two configurations $c_1, c_2 : \mathsf{Cfg}$ by expressing that c_1 can step to c_2. There, we understood the case of multiple successors to denote a demonic choice over all available successors. To add angelic choices, we change the step relation, so that it is a relation on $\mathsf{Cfg} \times (\mathsf{Cfg} \to \mathbb{P}\mathsf{rop})$. Intuitively, we step to a set of multiple possible successors. Note that our relation is still not required to be functional, which means that we can step to multiple sets, each again containing multiple successors. This gives a two-layered choice for choosing the successor: First, the set $X : \mathsf{Cfg} \to \mathbb{P}\mathsf{rop}$ is chosen, then an element of this set is chosen. We use the double-headed arrow $\twoheadrightarrow$ for this kind of semantics. For us, we introduce the convention that the choice of which set to step to is *angelic*, while the element of the set is selected *demonically*. This is opposite to the convention introduced in DimSum [41]. We consider this directions is more natural, since it requires establishing the "preconditions" of a step before assuming its "postcondition." To exemplify this, in Figure 5.1, we give two example semantics, where Cfg are just integers. The first, defined by StepAngelic, chooses the next successor angelically (where the choice is over all natural numbers $\mathbb{N}$). The second, StepDemonic, chooses the successor demonically. Both relations look very similar, the only difference is that one features an additional universal quantifier in the premise. This quantifier, however, is important. It ensures that for StepDemonic, the only suitable candidate for X is the full set. Since the choice of X is angelic, this angelic choice is forced: the full set must be chosen in order to step. For StepAngelic, any inhabited X is a suitable choice. If we, as the person allowed to execute angelic choices (the *angel*), want to ensure that we end up in precisely the state we intend to end up in, we must choose a singleton set containing only the intended state. This forces the hand of whoever executes demonic choices (the *demon*), since they must choose the value presented by the angel, *i.e.*, the choice is again forced.

The diagrams below show the choices available for finding a successor state of 0 (the starting state does not matter, since the step behavior is the same for all n). Starting at the left, they first show the possible angelic choices, and then for each angelic choice, the possible demonic choices. On the right, we see the just-described behavior. The angelic choice is trivial: the angel is forced to choose $\mathbb{N}$. The demon can then produce a step to every number (in $\mathbb{N}$). On the left, we see many angelic choices. In fact, the diagram is not complete, since we can step to any set (except for the empty set). Thus, if the angel was feeling lucky, they could also pick a larger set, and leave some actual choices to the demon. Since the goal of the angel is to ensure the program is correct, the angel will want to limit the demon's choice as much as possible. Thus, we simplify our definitions, by introducing the convention that the angel always chooses the smallest possible set. Formally, this means that we require that our semantics is up-closed in the second argument. Mathematically, if $c \twoheadrightarrow X_1$, and $X_2 \supseteq X_1$, then also $c \twoheadrightarrow X_2$ must hold. The step relations shown in Figure 5.1 are up-closed.

Parsing Multi-Relation Semantics

Figure 5.2 shows another example semantics, this time using both angelic and demonic non-determinism within a single rule. This relation defines steps where,

$$\textsc{Step-Both}\quad \frac{\mathrm{even}(k_a) \qquad (\forall k_d.\, \mathrm{prime}(k_d) \rightarrow (n + k_a + k_d) \in X)}{n \twoheadrightarrow X}$$

Figure 5.2: Another example relation, exhibiting both angelic and demonic non-determinism.

starting at a number n, first an even number must be chosen angelically. Afterwards, a prime is chosen demonically. Both of these choices are added to the original number to arrive at the next state. In general, we usually define such a semantics using inference rules. If we have several inference rules, then these inference rules define several angelic choices (although usually there is only one inference rule per origin state). Similarly, every free variable (like k_a) in this inference rule is also instantiated by an angelic choice. The conditions on top of the inference rule (in the above example, evenk_a) must be proven by the angel. In other words, these further constrain the angelic choice. The demonic choice is then controlled by the last precondition of the inference rule, which usually has the form $(\forall y \dots z.\, P \rightarrow Q \rightarrow \cdots \rightarrow f(x, \dots) \in X)$. The universally quantified variables, like k_d in the example above, are then understood as defining the demonic choices. The conditions that follow (like prime(k_d)) further constrain this demonic choice, *i.e.*, the demon must prove that these hold true. Afterwards, we can actually see the proper successor state, which is $n + k_a + k_d$ in the previous example.

To generalize this, observe that multi-relation semantics are yet another variation of predicate transformers. In fact, in our angelic-demonic variation, where the choice of $x \in X$ is demonic, their polarity is similar to that of protocols from Section 4.1. The facts needed to establish a step define the preconditions, and the facts describing which values must be contained in Q are the postcondition.

Undefined and No Behavior

We should look at what happens when no X can be stepped to, or when we can step to the empty set $\varnothing$. Before, we equated not being able to step with *undefined behavior*. It turns out that this still holds, but *only* because we made it so that multiple successor sets constitute an angelic choice. Remember that we, the programmer, need to justify angelic choices by showing that we can find a suitable set X. When we are stuck, this is like proving an existential quantifier over the empty set, which is false. Thus, the program is incorrect, *i.e.*, it exhibits *undefined behavior* (UB). A more interesting case happens if we can prove a step to the empty set $\varnothing$. Then, the demon would next be forced to choose a value, and we as the programmer (the angel) would be required to write the program such that it can handle all values the

$$\textsc{TraceEnd}\quad \frac{c \in X}{c \twoheadrightarrow^{\mathsf{trace}} X} \qquad\qquad \textsc{TraceStep}\quad \frac{c \twoheadrightarrow Y \qquad \forall y \in Y.\, y \twoheadrightarrow^{\mathsf{trace}} X}{c \twoheadrightarrow^{\mathsf{trace}} X}$$

Figure 5.3: The co-inductive trace relation $\twoheadrightarrow^{\mathsf{trace}}$ of $\twoheadrightarrow$.

demon can choose. When the set of possible demonic choices is empty, the demon will not be able to choose a value. Thus, the program does not need to handle any possible next step, it is automatically correct. We call this *no behavior* (ND). Again, a program having *no behavior* is weird, since in practice, computers do not magically disappear into the empty set. Instead, a more intuitive understanding of *no behavior* is that the program stops, since at this point, the program has executed successfully and nothing further needs to be proven to establish this. Thus, it is natural for values to have *no behavior*. For all other expressions, *no behavior* is likely indicative of a modeling mistake, since this implies that our model is not actually realizable on an actual machine. To prevent this, we add an additional property to our definition of modules: except for value, no expression may exhibit *no behavior*. This forces us to formally prove the absence of *no behavior* whenever we define a new module. Formally, we require that $\forall c\, X.\, c \twoheadrightarrow X \implies \exists x. x \in X$.

Note that in non-total weakest preconditions, having *no behavior* is equivalent to divergence, since both allow establishing correctness for any postcondition.

5.1.1 Executions and Safety

Previously, we stated safety as being able to step, *i.e.*, as not having undefined behavior. This notion is easy to generalize: a configuration c is safe iff there is a X such that $c \twoheadrightarrow X$. Since we have proven the absence of ND, this set will not be empty, and thus a new actual successor state must exist. Then, our adequacy theorem stated that for any possible execution, it arrives only at safe values. We now need to generalize the concept of an execution. While this is possible, we can do better. Instead of defining adequacy by arguing that every execution arrives at a safe state, we can define the adequacy theorem to instead construct a *co-inductive trace*. Intuitively, such a trace encodes the strategy of the angel. It also is a multi-relation, but the set X is understood a bit differently: It is the "happy set," describing all the configurations we want to end up in. Later, this will just be all states that satisfy the postcondition. By working with traces, we can sidestep the discussion of safe states (*i.e.*, states that have reached the postcondition, or that are reducible). Being co-inductive, such a trace is best understood as a generator, that can continue to produce steps, until a configuration reaching the postcondition is reached. By the magic of co-induction, it can also produce an infinite series of steps, should the program diverge. When interacting with such a trace, one takes the role of the demon. As such, one must provide the generator with the next demonic choice, for it to produce a next step and resolve angelic choices. The definition of co-inductive traces can be found in Figure 5.3. Remember that it is co-inductive, so that execution trees can be infinite. Again, until terminating in a state satisfying X (TraceEnd), a trace yields an instance Y that c steps to, that is, it resolves the angelic choice. It then expects the other party to resolve the demonic choice by supplying it a $y \in Y$, so that the same process can begin anew.

5.1.2 Lifting Languages to Modules

We now describe how we can lift our existing languages to a module. This module $\uparrow\lambda$ lifting a language λ exhibits *undefined behavior* precisely when the existing language exhibits *undefined behavior*. Similarly, it reflects the demonic non-determinism of the language being lifted. Since languages never have *no behavior*, this module also does not have no behavior. Similarly, there are no non-trivial angelic choices. The new step relation $\twoheadrightarrow_{\mathsf{lift}}$ is completely defined by the following inference rule:

$$\textsc{Lift}\quad \frac{\neg\text{IsVal } e \Rightarrow \mathrm{reducible}(p, e, \sigma) \qquad \forall e'\, \sigma'.\, (e, \sigma) \longrightarrow_{\mathsf{p}} (e', \sigma') \Rightarrow X(e', \sigma')}{(e, \sigma) \twoheadrightarrow_{\mathsf{lift}} X}$$

Note that $\longrightarrow_{\mathsf{p}}$ is the old reduction relation.

This definition achieves two things: First, it ensures that being stuck, according to the old operational semantics, is *undefined behavior*. If a step is possible, then all possible steps are interpreted angelically. As a special case, values are encoded to have *no behavior*, since they represent program termination. Apart from the new relation, the values, expressions, states, and contexts are not affected by the lifting. The expressions of the module are precisely the expressions of the lifted language, *etc.*. To conclude this definition, we need to prove that it is indeed a valid module, which entails showing that it never exhibits *no behavior*.

Theorem 5.1 (Absence Of *No Behavior* For Lifting) *Let* $e : e_\lambda$ *not be a value and* $\sigma : \sigma$ *be given. If* $(e, \sigma) \twoheadrightarrow_{\mathsf{lift}} X$*, then* X *is nonempty.*

Proof If $(e, \sigma) \twoheadrightarrow_{\mathsf{lift}} X$, then by Lift, we have $\neg\text{IsVal } e \Rightarrow \mathrm{reducible}(p, e, \sigma)$. Since e is not a value, we get $\mathrm{reducible}(p, e, \sigma) \triangleq \exists e'\, \sigma'.\, (e, \sigma) \longrightarrow^{p}_{\lambda} (e', \sigma')$. Thus $(e', \sigma') \in X$, as $(e, \sigma) \longrightarrow^{p}_{\lambda} (e', \sigma')$. □

5.1.3 Linking and Private State

We define a linking operator for modules in Appendix A. To make linking between different languages meaningful, we refine our notion of ABI-compatibility. Previously, ABI-compatibility required that both languages have the same notion of state.

We now weaken this by requiring that they only have the same *public state*. We also introduce *private state*, which is internal to a module, and not shared with other modules when linking. The *overall state* of a module is the combination of both. When making an external call to another module, that overall state is split into its private and public parts, with the public part being handed over to the other module, so that it can operate on it. When returning from this call, the private state (which remained unchanged) is joined back with the public state, which was potentially changed by the external call. The relation $\mathsf{Split} : \Sigma \to \Sigma_{\mathsf{pub}} \to \Sigma_{\mathsf{priv}} \to \mathit{iProp}$ governs how the state can be split and merged. The linking operator requires that the state can be split and merged when at a boundary, where boundaries are tracked in ghost state, using the $\mathsf{atBoundary}$ token.

For the language lifting operator $\uparrow\lambda$, there is no private state (*i.e.*, it is unit), so all state is shared with the other side for linking. In particular, linking with λ_{C} means getting access to the λ_{C} heap σ_{C}.

5.2 Wrapping OCaml to the C ABI

We can now finally define the formal model of the OCaml runtime. The OCaml runtime is a module, which works similar to the "lifting module" of Section 5.1.2. It does not only turn λ_{ML} into a module, but also makes it compatible with the C linkage model. Specifically, when execution an λ_{ML} expression e, it wraps external calls made by that expression, so that they are compatible with the C linkage model. To do so, it has private state, which stores *e.g.*, the block-level heap and the garbage collection state. The linker takes care of actually transferring control to another module (usually $\uparrow\lambda_{\mathsf{C}}$), while keeping the private state of this module unchanged. Since this wraps the contained λ_{ML} expression into something compatible with the C linkage module, we also call it the *wrapping module*. The wrapping module also provides support for primitives, which are implemented as functions of the wrapper. Since λ_{ML} does not have the concept of a program (*i.e.*, the only valid λ_{ML} program is $\varnothing$), there are no λ_{ML} functions that λ_{C} can call. Instead, λ_{C} uses external calls to invoke FFI primitives. These primitives are implemented in this wrapper, and thus have access to the private state of the wrapper. Formally, their implementation is by adding rules for them to the operational semantics of the wrapper. Almost all primitives are further discussed in Section 5.3, except for one, main. This primitive can be used initially to start execution of the "main expression" e_{main}. It is the equivalent of the *main function*, and the whole wrapper is parameterized by this special expression. It can also be thought of as a model of the `caml_startup` function.

We formally denote the wrapping module as $\lceil e_{\mathsf{main}} \rceil_{\mathrm{FFI}}$. The whole system is then $\lceil e_{\mathsf{main}} \rceil_{\mathrm{FFI}} \oplus \uparrow\lambda_{\mathsf{C}}$, that is, it links λ_{C} with the runtime $\lceil e_{\mathsf{main}} \rceil_{\mathrm{FFI}}$. Execution is defined to start with the main primitive, which starts executing e_{main}. This expression then reduces, and occasionally switches between OCaml and C using external calls, and

$$
\begin{aligned}
\mathsf{Val} \ni v &::= (n : \mathbb{Z}) \mid (\gamma : \mathsf{Loc}) \\
\mathsf{Mutability} \ni m &::= \mathsf{Mut} \mid \mathsf{Imm} \\
\mathsf{Tag} \ni t &::= 0 \mid 1 \\
\mathsf{Block} \ni blk &::= \mathsf{B}(m, t, \vec{v}) \mid \mathsf{C}(\mathsf{rec}\, f\, x.\, e) \mid \mathsf{F}(\,\mathsf{Some}\, w \mid \mathsf{None}) \\
\mathsf{BlockStore} \ni \zeta &\triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Block} \\
\mathsf{tagNumber} &: \mathsf{Block} \to \mathbb{Z} \\
\mathsf{mutability} &: \mathsf{Block} \to \mathsf{Mutability}
\end{aligned}
$$

TagNumberDefault
$$\mathsf{tagNumber}(\mathsf{B}(t, m, \vec{v})) = t$$

MutabilityDefault
$$\mathsf{mutability}(\mathsf{B}(t, m, \vec{v})) = m$$

TagNumberCallback
$$\mathsf{tagNumber}(\mathsf{C}(\mathsf{rec}\, f\, x.\, e)) = 247$$

MutabilityCallback
$$\mathsf{mutability}(\mathsf{C}(\mathsf{rec}\, f\, x.\, e)) = \mathsf{Imm}$$

TagNumberCustom
$$\mathsf{tagNumber}(\mathsf{F}(\_)) = 255$$

MutabilityCustom
$$\mathsf{tagNumber}(\mathsf{F}(\_)) = \mathsf{Mut}$$

Figure 5.4: Formal model of the block-level heap.

the callback primitive. We now formalize the entirety of Chapter 3, starting with the block-level heap. Next, we continue with the representation relation describing how λ_{ML} values are encoded. Afterwards, we discuss our model of the garbage collector. Finally, we plug all of this together to give an operational semantics to state changes, which we expand to construct the final operational semantics.

The Block-Level Heap

To begin developing a formal model of the OCaml runtime, we formalize the block-level heap, starting with block-level values. All these definitions can be found in Figure 5.4. The basic notion is that of a block-level value Val, which is either an integer or a pointer to another block-level value. The block-level heap then stores three kinds of blocks: The first are normal blocks, which have a tag, a length (implicitly), and a list of block-level values. Unlike before, we now make the mutability of a block operational. There are two special kinds of blocks, modelling the two special kinds of blocks described in Section 3.4: One models callbacks, by simply storing an λ_{ML} callback value, and the other models custom blocks, by simply embedding a λ_{C} value (or None to indicate an uninitialized block). While regular blocks B carry an explicit mutability and and explicit tag, these are implicit for the other kinds of blocks. Closure blocks C always have tag 247, and are never mutable. Custom blocks F are always mutable, and always have tag 255. The function tagNumber can be used to get the numeric tag of a block. For this, we identify the tags $t : \mathsf{Tag}$ with the numbers $0, 1 : \mathbb{Z}$. Similarly, the function $\mathsf{mutability}$ defines the implicit mutability of blocks.

Linking OCaml Values to Block-Level Values

Next, we formally capture the connection between high-level λ_{ML} and block-level values. To do so, we define an inductive relation IsVal between both. Additionally, the relation is parameterized by ζ, since the representation of some high-level values is defined as pointer pointing to a specific block. The definition is in Figure 5.5. While this is straightforward for numbers, units, booleans, pairs, and sums, the case of OCaml locations ℓ is more interesting. For these, we must find a specific block that encodes the contents of the location. Additionally, we do not just want any block, but we want that block to be permanently linked to that location, so that, even in future, only this block and no other block represents this location. To ensure this, we introduce a new map χ : LocMap, which describes additional "ghost" information for each block. Specifically, it describes a block as linked to ML location, or as being private. We call this map the *visibility* map. It also includes a third possible mode, which is used for custom blocks, linking them to foreign values. As shown by the rule IsValForeign, foreign values $\textcircled{\iota}$ are backed by a specific block. Usually, this block is a custom block, but the definitions do not require it. This map χ must be injective: No two block-level locations may be assigned to the same OCaml location. Similarly, no two block-level locations may have the same foreign id. It is allowed for two OCaml locations to be private.

We next define how to embed the entire OCaml heap into the block-level store. This is achieved by partitioning the block-level store into two disjoint parts: $\zeta = \zeta_{\mathsf{rest}} \dot\cup \zeta_{\mathrm{ML}}$. One part, ζ_{rest}, has no particular constraints, except for being disjoint from the other part. The other part, ζ_{ML}, contains exactly the blocks that, according to χ, back a heap element. Formally, we require that IsStoreBlocks$(\chi, \zeta_{\mathrm{ML}}, \sigma_{\mathrm{ML}})$. This also requires χ to be large enough, so that it assigns a block-level location γ to each OCaml location ℓ. Further, we require that IsStore$(\chi, \zeta, \sigma_{\mathrm{ML}})$ holds on the entire block-level heap ζ (the union of both parts). Taken together, both predicates describe that the block-level heap contains blocks matching the content of each OCaml reference (or array), which also means that there are blocks for all high-level values encoded as immutable blocks. Note that our definitions make no specific requirements about the identity of immutable blocks. For example, if we have two OCaml references, which both contain the same pair $\langle 1,2 \rangle$, these pairs might be stored in different blocks. A more clever implementation of the OCaml runtime, that uses hash-consing, could store both of these in the same block (which is legal, since that block is immutable). Our definitions allow both. Since we consider this an implementation detail of the runtime, we do not specify this, so that our formal model remains correct even when more optimizations are added to the OCaml runtime/compiler. The definition IsSameDom also warrants a closer look: It defines the locations in σ_{ML} that are part of

$$\mathsf{LocKind} \ni k ::= \mathsf{Pub}\ \ell \mid \mathsf{Priv} \mid \mathsf{Fgn}\ \iota$$

$$\mathsf{LocMap} \ni \chi \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{LocKind} \text{ injective in } \mathsf{Pub}, \mathsf{Fgn}$$

$$\textsc{IsValInt}\quad \mathsf{IsVal}(\chi, \zeta, n, n) \qquad \textsc{IsValTrue}\quad \mathsf{IsVal}(\chi, \zeta, \mathsf{true}, 1) \qquad \textsc{IsValFalse}\quad \mathsf{IsVal}(\chi, \zeta, \mathsf{false}, 0) \qquad \textsc{IsValUnit}\quad \mathsf{IsVal}(\chi, \zeta, \langle\rangle, 0)$$

$$\textsc{IsValLoc}\ \frac{\chi[\gamma] = \mathsf{Pub}\ \ell}{\mathsf{IsVal}(\chi, \zeta, \gamma, \ell)} \qquad \textsc{IsValForeign}\ \frac{\chi[\gamma] = \mathsf{Fgn}\ \iota}{\mathsf{IsVal}(\chi, \zeta, \gamma, \textcircled{\iota})} \qquad \textsc{IsValClosure}\ \frac{\zeta[\gamma] = \mathsf{C}(\mathsf{rec}\ f\ x.\ e)}{\mathsf{IsVal}(\chi, \zeta, \gamma, \mathsf{rec}\ f\ x.\ e)}$$

$$\textsc{IsValPair}\ \frac{\zeta[\gamma] = \mathsf{B}(\mathsf{Imm},0,[v, v']) \qquad \mathsf{IsVal}(\chi, \zeta, v, V) \qquad \mathsf{IsVal}(\chi, \zeta, v', V')}{\mathsf{IsVal}(\chi, \zeta, \gamma, \langle V, V' \rangle)}$$

$$\textsc{IsValInl}\ \frac{\zeta[\gamma] = \mathsf{B}(\mathsf{Imm},0,[v]) \qquad \mathsf{IsVal}(\chi, \zeta, v, V)}{\mathsf{IsVal}(\chi, \zeta, \gamma, \mathsf{inl}\ V)} \qquad \textsc{IsValInr}\ \frac{\zeta[\gamma] = \mathsf{B}(\mathsf{Imm},1,[v]) \qquad \mathsf{IsVal}(\chi, \zeta, v, V)}{\mathsf{IsVal}(\chi, \zeta, \gamma, \mathsf{inr}\ V)}$$

$$\textsc{IsHeapBlock}\ \frac{\mathsf{IsVal}(\chi, \zeta, \vec{v}, \vec{V})}{\mathsf{IsHeapBlock}(\chi, \zeta, \mathsf{B}(\mathsf{Mut},0,\vec{v}), \vec{V})}$$

$$\begin{aligned}
\mathsf{IsPrivate}(\chi, \zeta) &\triangleq \forall \gamma \in \mathrm{dom}\ \zeta.\ \chi[\gamma] = \mathsf{Priv} \\
\mathsf{IsPublic}(\chi, \sigma_{\mathrm{ML}}) &\triangleq \forall \ell \in \mathrm{dom}\ \sigma_{\mathrm{ML}}.\ \exists \gamma.\ \chi[\gamma] = \mathsf{Pub}\ \ell \\
\mathsf{IsSameDom}(\chi, \zeta, \sigma_{\mathrm{ML}}) &\triangleq \forall \gamma.\ \gamma \in \mathrm{dom}\ \zeta \iff \exists \ell\ \vec{V}.\ \chi[\gamma] = \mathsf{Pub}\ \ell \land \sigma_{\mathrm{ML}}[\ell] = \vec{V} \\
\mathsf{IsStoreBlocks}(\chi, \zeta, \sigma_{\mathrm{ML}}) &\triangleq \mathsf{IsPublic}(\chi, \sigma_{\mathrm{ML}}) \land \mathsf{IsSameDom}(\chi, \zeta, \sigma_{\mathrm{ML}}) \\
\mathsf{IsStore}(\chi, \zeta, \sigma_{\mathrm{ML}}) &\triangleq \forall \ell\ \vec{V}\ \gamma\ \mathsf{blk}.\ \sigma_{\mathrm{ML}}[\ell] = \vec{V} \to \chi[\gamma] = \mathsf{Pub}\ \ell \to \zeta[\gamma] = \mathsf{blk} \to \\
&\qquad \mathsf{IsHeapBlock}(\chi, \zeta, \mathsf{blk}, \vec{V})
\end{aligned}$$

Figure 5.5: Definitions linking λ_{ML} and block-level concepts.

ζ_{ML}, namely all locations that store a proper array. Notably, this excludes locations that store $\lightning$, the one special value allowed for heap cells in $\lambda_{\mathtt{ML}}$. This dummy value is used to mark the contents of these locations as unavailable for $\lambda_{\mathtt{ML}}$. Similarly, IsStore imposes no requirement on these locations.

Garbage Collection

There are two key ingredients to our formal model of garbage collection. The first is that in $\lambda_{\mathtt{ML}}$, as well as on the block-level heap, garbage collection is invisible. We think this assumption is justified since the garbage collector works very hard to ensure that it only deallocates unreachable blocks, and that when it moves a block, all references to it are updated atomically. More informally, the block-level heap can be thought of as logical memory, following Hur and Dreyer [16], where garbage collection is invisible. In glue code, garbage collection becomes visible. The second key idea is then that the glue code can only operate on the block-level heap over "arms-length" operations, where the definition of *arms-length* incorporates the effects of the garbage collector. This also means that we do not directly embed the block-level memory into C, but instead give an axiomatic treatment of accesses to it using the FFI primitives. In the actual FFI, there are uses where raw access to the runtime memory is needed, which we discuss in Section 3.5. We now proceed to define an encoding of block-level values to $\lambda_{\mathtt{C}}$ values, similarly to how IsVal encoded $\lambda_{\mathtt{ML}}$ values into block-level values. The relation $\sim_{\mathrm{C}}^{\theta}$, defined in Figure 5.6 does precisely this: When $v \sim_{\mathrm{C}}^{\theta} w$, then v is encoded as the $\lambda_{\mathtt{C}}$ value w. The index (rather superscript) parameter, the address map θ, plays a role similar to LocMap played for IsVal: it describes which $\lambda_{\mathtt{C}}$ addresses encode which block-level locations. Representing a block-level location is then straightforward. Representing an integer could also be, but we additionally introduce a function $\mathrm{codeInt} : \mathbb{Z} \to \mathbb{Z}$, which describes how integers are encoded. The whole development is then parametric over this function, which is only required to be injective. This function captures the "lowest bit" encoding present in the actual OCaml runtime, where block-level integers are distinguished from block-level pointers by having their lowest bit set (and being shifted one bit to the left). We mentioned already that the block-level memory is not actually serialized into the C memory. Instead, the C locations associated with block-level locations by $\sim_{\mathrm{ML}} \theta$ merely describe which arguments must be passed to formalization of macros like `Store_field` to access the proper block. These addresses are better thought of as *abstract names*, instead of concrete values. This also means that, if θ changes, the blocks referred to by an address also change. Indeed, this is how we implement garbage collection. The address map $\theta : \mathrm{AddrMap}$ is better thought of as the current state of the garbage collector. It assigns "real" locations to the "logical" block-level heap, as discussed in by Hur and Dreyer [16]. The formal equivalent of the garbage collector choosing tho deallocate a block is then that this block's location is removed from the address map θ. Such a block can then no longer be referenced from C,

$$\mathsf{AddrMap} \ni \theta \triangleq \mathsf{Loc} \xrightharpoonup{\text{fin}} \mathsf{Addr} \text{ which is injective}$$

$$\mathsf{RootMap} \ni rm \triangleq \mathsf{Addr} \xrightharpoonup{\text{fin}} \mathsf{Val}$$

$$rs \ni \mathsf{RootSet} \triangleq \mathcal{P}_{\text{fin}}\,(\mathsf{Addr})$$

$$\textsc{ReprInt}\quad \frac{m = \mathsf{codeInt}\; n}{n \sim^{\theta}_{C} m} \qquad \textsc{ReprLoc}\quad \frac{\theta[\gamma] = a}{\gamma \sim^{\theta}_{C} a} \qquad \textsc{ElemOfBlock}\quad \frac{\gamma = \vec{v}_i}{\gamma \in \mathsf{B}(t, m, \vec{v})} \qquad \textsc{ReprInj}\quad \frac{v_1 \sim^{\theta}_{C} w_1 \qquad v_2 \sim^{\theta}_{C} w_2}{v_1 = v_2 \iff w_1 = w_2}$$

$$\mathsf{GcClosed}(\zeta, \theta) \triangleq \forall \gamma\, blk\, \gamma'.\, \gamma \in \mathsf{dom}\ \theta \to \zeta[\gamma]{=}blk \to \gamma' \in blk \to \gamma' \in \mathsf{dom}\ \theta$$

$$\mathsf{GcRooted}(rm, \theta) \triangleq \forall a \gamma.\, rm[a] = \gamma \to \gamma \in \mathsf{dom}\ \theta$$

$$\mathsf{ReprRoots}(rm, \theta, \sigma_C) \triangleq \mathsf{dom}\ rm {=} \mathsf{dom}\ \sigma_C \wedge \forall a v w.\, rm[a]{=}v \to \sigma_C[a]{=}w \to v \sim^{\theta}_{C} w$$

$$\mathsf{WithRoots}(\theta, rm, \sigma_{rest}, \sigma_C) \triangleq \exists \sigma_{root}.\, \mathsf{ReprRoots}(rm, \theta, \sigma_{root}) \wedge \sigma_C = \sigma_{rest} \dot{\cup} \sigma_{root}$$

$$\textsc{GcStateSwitch}\quad \frac{\mathsf{GcClosed}(\zeta, \theta) \qquad \mathsf{GcRooted}(rm, \theta) \qquad \mathsf{WithRoots}(\theta, rm, \sigma_{rest}, \sigma_C) \qquad rs = \mathsf{dom}\ rm}{\mathsf{GcStateSwitch}(\zeta, \theta, rm, \sigma_{rest}, rs, \sigma_C)}$$

Figure 5.6: Definitions for garbage collection, and for roots.

which, from the C programmer's perspective, is no different from it not existing at all. Similarly, if the garbage collector decides to move a block, this is modelled formally by changing the value that block's location is mapped to by the address map θ.

The predicate GcClosed describes one of the formal requirements we have for the garbage collector, namely that it only deallocates unreachable blocks. The condition is formulated inversely, namely that if the garbage collector still keeps a block blk alive, all other blocks referenced by that block (written as $\gamma \in blk$, see ElemOfBlock) must also still be alive. We say that this property describes that the address map is *closed under reachability*. We saw earlier how we could root a value to ensure that this value is kept alive by the garbage collector. We also simplified our rooting mechanism so that all roots need to be explicitly (un)registered, using primitives discussed in Section 5.3. Due to this simplification, we can simply capture roots formally using a rm : RootMap, which stores the block-level representation of a root (remember that a root must be a block-level value, at least during a GC run). Given a C location a, that location is a root precisely when it is assigned a value by the roots map: $a \in \mathsf{dom}\ rm$. The condition GcRooted then describes the next formal requirement of the GC: That all currently rooted values are considered reachable, and therefore present in θ. These two conditions, GcRooted and GcClosed, are the only two conditions we require the GC state to satisfy. This again leaves open how the GC is actually implemented, so that code verified against our model is compatible with a larger variety of potential GC implementations. An additional requirement about the roots map is that it agrees with the content of the C memory at rooted locations. To describe this, we split the C memory into two disjoint parts: $\sigma_{\mathsf{C}} = \sigma_{\mathsf{Crest}} \dot{\cup} \sigma_{\mathsf{Croot}}$. We then require that $\mathsf{ReprRoots}(rm, \theta, \sigma_{\mathsf{Croot}})$, which is just the straightforward lifting of $\sim^{\theta}_{\mathsf{C}}$ to maps. This is formalized by the WithRoots predicate, which describes how to construct the full C state from the C state without roots, plus the roots stored in block-level form. If the user were to store a C value in a root that does not encode a block-level value, the proposition WithRoots becomes false. By later requiring that this proposition needs to be proven by the angel, we can make it *undefined behavior* for roots to store improper values. This UB only happens during garbage collection runs. In-between those, roots are allowed to store anything. Next to the roots map, we also introduce the roots set $rs : \mathsf{RootSet}$, which is sometimes used when we only describe which sets are roots, instead of the full root contents. When we need the full contents, we require that $rs = \mathsf{dom}\ rm$. Specifically, the we store the roots set when the full λ_{C} heap σ_{C} is present. When the roots map is used, we only store the non-rooted part of the λ_{C} heap σ_{Crest}. When given an address map θ, these two kinds of representations are equivalent. However, the one with rm does not become invalidated that happens when the garbage collector runs (and θ changes).

The relation GcStateSwitch, defined by GcStateSwitch, puts it all together. It requires that θ is well-formed, and that two versions of the λ_{C} state–the one using the roots map, and the one using just the roots set–agree.

The Wrapper State

We now have all the definitions in place to define the full wrapper state, shown in Figure 5.7. The wrapper state has two sides: the C side, and the OCaml side. The C side is the public state, since outgoing calls are made using the C linkage model. The OCaml side is purely internal, and never exposed to the outside world. It is the state that is actually present when λ_{ML} code is executing. Whenever this code makes an external call to C, the wrapper switches to the C side, which is also the side used to define the FFI primitives. The difference between both sides is that the OCaml side operates on a proper λ_{ML} heap σ_{ML}. In contrast, while working in C, we operate on the block-level state, with the entire λ_{ML} heap σ_{ML} serialized into its block-level representation. When switching sides, it is required that both sides represent the same data. The visibility map χ is always present, and works the same on both sides. The block-level heap ζ is also present on both sides, but it has a slightly different meaning on both: When we are on the OCaml side, it includes the full λ_{ML} heap. On the OCaml side, it does not: To not duplicate information, we remove all block-level blocks that are also part of the λ_{ML} state σ_{ML} from ζ, so that the information about their content is only stored in one place. What the OCaml side also lacks is the address map θ. Since the garbage collector is only observable using the OCaml

$$
\begin{aligned}
\mathsf{CSideState} \ni \rho_C &\triangleq \mathit{LocMap} \times \mathit{BlockStore} \times \mathit{AddrMap} \times \mathit{RootSet} \\
\mathsf{MLSideState} \ni \rho_{ML} &\triangleq \mathit{LocMap} \times \mathit{BlockStore} \times \mathit{RootMap} \times \Sigma_C \\
\Sigma \ni \rho &::= \mathsf{CState}(\rho_C, \sigma_C) \mid \mathsf{MLState}(\rho_{ML}, \sigma_{ML}) \\
\Sigma_{\mathsf{pub}} &\triangleq \Sigma_C \\
\Sigma_{\mathsf{priv}} &\triangleq \mathsf{CSideState}
\end{aligned}
$$

SWrapperSplit
$$\mathit{Split}(\mathsf{CState}(\rho_C, \sigma_C), \sigma_C, \rho_C)$$

FreezeRefl
$$\mathsf{blk} \overset{\mathsf{freeze}}{\rightsquigarrow} \mathsf{blk}$$

FreezeMut
$$\mathsf{B}(\mathsf{Mut},\mathsf{t},\vec{v}) \overset{\mathsf{freeze}}{\rightsquigarrow} \mathsf{B}(\mathsf{Imm},\mathsf{t},\vec{v})$$

ExposeRefl
$$\mathsf{k} \overset{\mathsf{expose}}{\rightsquigarrow} \mathsf{k}$$

ExposePublic
$$\mathsf{Priv} \overset{\mathsf{expose}}{\rightsquigarrow} \mathsf{Pub}\,\ell$$

MlToCDemonic
$$\frac{\begin{array}{ccc} & \chi \subseteq \chi' \qquad \mathit{IsStoreBlocks}(\chi', \sigma_{ML}, \zeta_{ML}) & \\ \mathit{IsPrivate}(\chi', \zeta_{\mathsf{newimm}}) & \zeta' = \zeta \mathbin{\dot\cup} \zeta_{ML} \mathbin{\dot\cup} \zeta_{\mathsf{newimm}} & \mathit{IsStore}(\chi', \zeta', \sigma_{ML}) \\ \mathit{IsVal}(\chi', \zeta', \vec{v}, \vec{V}) & \mathit{GcStateSwitch}(\zeta', \theta, rm, \sigma_C, rs, \sigma'_C) & \vec{v} \sim^{\theta}_{C} \vec{w} \end{array}}{(\vec{V}, (\chi, \zeta, rm, \sigma_C), \sigma_{ML}) \mathrel{\hat{\hookrightarrow}}_{\mathrm{M2C}} (\vec{w}, (\chi', \zeta', \theta, rs), \sigma'_C)}$$

MlToC
$$\frac{\begin{array}{c} \mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi \\ \forall \gamma \ell.\ \chi[\gamma] = \mathsf{Pub}\,\ell \to \ell \in \mathsf{dom}\ \zeta \to \sigma_{ML}[\ell] = \lightning \qquad \mathsf{dom}\ \sigma_C \mathbin{\#\#} \mathsf{dom}\ rm \\ \forall \vec{w}\ \rho_C\ \sigma'_C.\ (\vec{V}, (\chi, \zeta, rm, \sigma'_C), \sigma_{ML}) \hookrightarrow (\vec{w}, \rho_C, \sigma'_C) \to X(\vec{w}, \rho_C, \sigma_C) \end{array}}{(\vec{V}, (\chi, \zeta, rm, \sigma_C), \sigma_{ML}) \hookrightarrow_{\mathrm{M2C}} X}$$

CToMlAngelic
$$\frac{\begin{array}{ccc} & \chi \overset{\mathsf{expose}}{\rightsquigarrow} \chi' & \\ \zeta \overset{\mathsf{freeze}}{\rightsquigarrow} \zeta' \mathbin{\dot\cup} \zeta_{ML} & \mathit{IsStoreBlocks}(\chi', \sigma_{ML}, \zeta_{ML}) & \mathit{IsStore}(\chi', \zeta' \mathbin{\dot\cup} \zeta_{ML}, \sigma_{ML}) \\ \mathit{IsVal}(\chi', \zeta' \mathbin{\dot\cup} \zeta_{ML}, \vec{v}, \vec{V}) \qquad \mathsf{dom}\ rm = rs & \mathit{WithRoots}(\theta, rm, \sigma_C, \sigma'_C) & \vec{v} \sim^{\theta}_{C} \vec{w} \end{array}}{(\vec{w}, (\chi, \zeta, \theta, rs), \sigma) \mathrel{\hat{\hookrightarrow}}_{\mathrm{C2M}} (\vec{V}, (\chi', \zeta', rm, \sigma'_C), \sigma_{ML})}$$

Figure 5.7: Definition of the wrapper state, and of the rules for switching.

FFI, the address map is not needed when OCaml code is running. Thus, it is only materialized when switching to the C side. This map is materialized anew each time such a switch happens, which formalizes that the garbage collector can run every time OCaml runs. The final difference is in the handling of roots. While on the C side, we only store a roots set rs, since the contents of each roots is fully defined by what is stored in the λ_{C} heap σ_{C}. While on the OCaml side, the roots are stored as block-level values. But to again not store the same information in different places, the locations storing roots are deleted from σ_{C}. The relation between all those states is formally defined by the operations for switching from the OCaml side to the C side, and vice-versa. A graphical representation of the interaction of all these maps during such a switch can be found in Figure 5.8, see below for an explanation.

Switching to C

The rule MLToC defines how the OCaml state is switched to C, which in particular involves deserializing the OCaml heap to a block-level heap, as well as demonically choosing a GC state θ. Since this step involves mostly demonic choices, the rule MLToCDemonic defined what demonic choices are possible. First, the visibility map χ is extended, to allow for new allocations that happened while λ_{ML} code was running. Next, two new fragments of the block-level store are demonically chosen, namely ζ_{ML} and ζ_{newimm}. The former contains a block for each λ_{ML} array, and the latter contains some private blocks, which accounts for new allocations that happened in OCaml code. These two parts are then combined with the old fragment block-level store to form the new store, which must in particular be a faithful representation of σ, as defined by $\mathrm{IsStore}$. Next, a new address map θ is demonically chosen. This address map must be subject to the usual constraints of being closed under reachability, and containing at least all roots, as defined by $\mathrm{GcStateSwitch}$. Finally, the new C state σ'_{C} is created by adding the C representation of all roots. The rule does not just convert the state, but also takes additional OCaml values $\vec{V}$ and converts these to their new C level representatives $\vec{w}$. The choices of ζ' and θ are further constrained such that $\vec{w}$ faithfully represents the old values. This mechanism is used for the arguments (and return values) of function calls.

The full rule for lowering the state, MLToC, is mainly defined by reference to MLToCDemonic. Additionally, there are three side conditions, which are there to ensure that there is no *no behavior*, which is necessary to prove Theorem 5.2. All three of them are invariants, that continue to hold while an λ_{ML} expression is executing. Our program logic later validates that these actually are invariants, so that it can prove that there is no *undefined behavior*. Attentive readers can already observe that $\mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi$ remains invariant even when we switch the state to the C side. It continues to hold, by *e.g.*, requiring $\mathrm{IsPrivate}(\chi', \zeta_{\mathsf{newimm}})$ in MLToCDemonic. In fact, this is an invariant of the entire semantics, continuing to hold while C code is executing and interacting with the runtime via primitives. One side condition is a

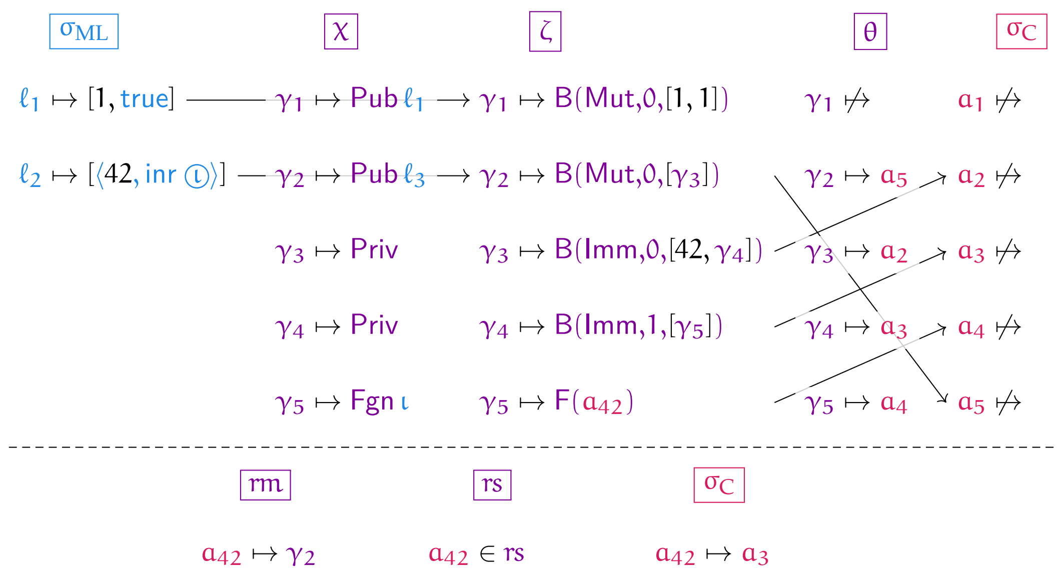

Figure 5.8: An example visualizing the interaction of the various maps.

bit more interesting: It expresses that the block-level heap ζ and the λ_{ML} state σ_{ML} do not store conflicting data for any block. In fact, if the block-level heap stores data at a location, then the OCaml heap cell for this location must be $\not\!\!\iota$, which explicitly marks the data for this cell as being stored at the block-level, and as being inaccessible to OCaml.

In Figure 5.8, we give an example of such an operation. We start with the σ_{ML} shown on the very left, where there are two arrays. Additionally, one of the locations is rooted. For the sake of explanation, let us also assume that these two locations were just created by λ_{ML} code. First, the visibility map χ is demonically extended to give block-level locations for our λ_{ML} arrays stored at ℓ_1 and ℓ_2. Additionally, the new private blocks at γ_3 and γ_4 are created. We presume the custom block at γ_5 already exists, since custom blocks can only created using the FFI. Following the parlance of MLToCDemonic, we have that $\text{dom}\ \zeta = \{\gamma_5\}$ (*i.e.*, the old block-level store already contained γ_5), $\text{dom}\ \zeta_{\text{ML}} = \{\gamma_1, \gamma_2\}$, and $\text{dom}\ \zeta_{\text{newimm}} = \{\gamma_3, \gamma_4\}$. These can all be combined to form the new store. Additionally, it is easy to check that this is a faithful representation. Next, a new garbage collection state is chosen demonically. Here, we display a situation where the block at γ_1 is no longer reachable. The block at γ_2, however, is reachable, since there is a root storing this value. Thus, this block and all blocks reachable from it have entries in the address map θ. This map completely changes every time the garbage collector runs. In χ, on the other hand, mappings remain stable throughout the execution. Further, to emphasize the point that we *do not* serialize the block-level heap ζ into the λ_C heap σ_C, we chose a heap σ_C where these addresses are not allocated. In the semantics, the choice of the address map θ is actually completely independent of which addresses are, and are not, allocated in the C heap λ_C. Finally, we have the root at a_{42}, which stores γ_2. When switching to C, this gets lowered into the C representation a_3 of the root's value γ_2, since $\gamma_2 \sim^{\theta}_{C} a_3$. All that is retained is the roots set rs, to keep track of the fact that a_{42} was registered as a root. As a high-level intuition, the arrows indicate which block locations are identified using χ and θ.

Switching to OCaml

The rule for moving back to the OCaml side is CToMlAngelic. It is slightly shorter, since it does not require choosing a new GC state. It is still interesting, since it chooses the new λ_{ML} heap *angelically*. Angelic choice is necessary here to properly conjure up a λ_{ML} heap σ_{ML} given just the data available while on the C side. To see the difficulty yourself, consider Figure 5.8, but imagine that the λ_{ML} heap σ_{ML} was not given, instead that you need to create on so that the picture still matches. Already for ℓ_1, this leads to issues: It is not clear whether a block-level 1 should become an λ_{ML} 1, or the value `true`. In fact, both would be equally valid choices. In most cases, the wanted value is the one that was stored there previously, but this is not always the case (*e.g.*, when using `Obj.magic`, this is explicitly not wanted). In either way, since the old version of the λ_{ML} heap σ_{ML} is not present at this point, this can not be used to restrict the choice in the operational semantics. Even if we changed the semantics accordingly, it would not help with choosing a λ_{ML} representative for freshly allocated locations. The correct approach, it turns out, is leaving this choice to the programmer when arguing correctness. This is precisely angelic non-determinism. This is also not a new observation: Already in 1989, Back [2] observed that dual non-determinism is needed for operationalizing such representation switches. Once the new λ_{ML} heap σ_{ML} has been chosen, all blocks associated with it are deleted from ζ. What remains in ζ' are all immutable blocks, all custom blocks, and the blocks for λ_{ML} locations that have been marked $\lightning$ in the new λ_{ML} heap σ_{ML}.

Apart from angelically choosing the new heap, the rule allows making two further choices, indicated by the relations $\overset{\text{freeze}}{\rightsquigarrow}$ and $\overset{\text{expose}}{\rightsquigarrow}$. The first relation, $\overset{\text{freeze}}{\rightsquigarrow}$, allows *freezing a block*. Specifically, it allows the programmer to turn a mutable block into an immutable one. We lift this relation to entire maps, so that $\zeta \overset{\text{freeze}}{\rightsquigarrow} \zeta' \mathbin{\dot\cup} \zeta_{ML}$ means that the domain of ζ and $\zeta' \mathbin{\dot\cup} \zeta_{ML}$ remain the same, only some previously mutable blocks are now immutable. This pattern of allowing the programmer to freeze blocks is necessary to properly support initializing blocks. When a new block is allocated using `caml_alloc`, it is mutable. It must be mutable, so that useful data can be stored in it. But if this block has been allocated to encode *e.g.*, a pair, it must eventually become immutable, since pairs are represented by immutable blocks. This is what block freezing accomplishes: When switching back to the OCaml side, some blocks can be frozen so that they may encode the block-level representation of pairs and sums. The other relation, $\overset{\text{expose}}{\rightsquigarrow}$, allows *exposing* new blocks. This is again needed to properly initialize some blocks. When a new block is created, it is not only mutable, but also marked as Priv in χ. This means that this block can not be used encode an λ_{ML} array/reference, since the blocks backing such arrays are identified by an entry $\mathsf{Pub}\,\ell$ in θ. Thus, $\overset{\text{expose}}{\rightsquigarrow}$ similarly allows making some private blocks public. Note that the target side of $\overset{\text{expose}}{\rightsquigarrow}$ must still be injective, so that the chosen locations must be fresh.

The final requirement is that the roots, which are currently just represented by the roots set rs and the λ_{C} heap σ_{C}, store proper block-level values. This is enforced by requiring that a roots map rm exists, which agrees with the roots encoded in σ_{C}. Formally, this is achieved by requiring that $\mathsf{WithRoots}(\theta, rm, \sigma_{\text{C}}, \sigma'_{\text{C}})$. The resulting new λ_{C} heap σ'_{C} is a remnant state, from which all locations storing roots have been removed. This roots map must contain precisely all rooted locations, as required by $\operatorname{dom} rm = rs$. Just like $\hat{\looparrowright}_{\text{M2C}}$, the relation $\hat{\looparrowright}_{\text{C2M}}$ also allows converting a list of λ_{C} values $\vec{w}$ to λ_{ML} values, along with the general state switching. The conversion follows the pattern described above: The λ_{ML} values $\vec{V}$ are chosen angelically, subject to IsVal. The block-level values $\vec{v}$ that are the other argument of IsVal are uniquely determined by $\vec{v} \sim^{\theta}_{\text{C}} \vec{w}$.

The Full Semantics

We are now able to write down the full operational semantics of the wrapper. The relevant definitions are in Figure 5.9. The wrapper is a module $[e_{\mathsf{main}}]_{\text{FFI}}$, which is parameterized by an OCaml expression e_{main}. It also defines a program $[e_{\mathsf{main}}]_{\text{FFI}}$, which implements all the FFI primitives (which we discuss in Section 5.3). The wrapper combines some of the techniques already described in Section 5.1.2 and Appendix A. Specifically, it also embeds the λ_{ML} semantics, lifting them to be multi-language-based. Additionally, it ensures that the state can switch between the OCaml and C side around external calls. The rule SWRAPSTEP describes the normal case, where the wrapper is on the OCaml side and executing the OCaml code. When this code makes an external call, the wrapper intercepts it (rule SWRAPHOOKCALL), saves the context, switches the state to the C side, and steps to Call. This Call expression is recognized by the linker. Thus, once the wrapper arrives there, the linker takes over, switches execution to another module, and eventually substitutes back a result. Once this result has arrived, the rule SWRAPRETTOML translates the λ_{C} value to an λ_{ML} value, switches the state back to the OCaml side, and substitutes this λ_{ML} value into the saved context. When the λ_{ML} expression finishes, it is again serialized into a λ_{C} expression by SWRAPVALTOC, which is used when a callback (or the main expression) returns.

Finally, we also have a rule for our first primitive, $\mathsf{main}_{e_{\mathsf{main}}}$. Note that it is not this primitive that is parameterized by e_{main}–the whole development is generic

$$
\begin{aligned}
\mathsf{SExpr} \ni E &::= w \mid \lceil e \rceil \mid \mathsf{Call}\ \mathsf{fn}\ \vec{w} \mid \mathsf{RunPrim}\ \mathsf{prm}\ \vec{w} \\
\mathsf{Ctx} \ni K &::= \bullet \mid K \cdot K_{ML} \\
\mathsf{Expr} \ni e &\triangleq \mathsf{SExpr} \times \mathsf{Ctx} \\
\mathsf{Prim} \ni \mathsf{prm} &::= \mathsf{main}_{e_{\mathsf{main}}} \mid \mathsf{Val\_int} \mid \cdots \\
\mathsf{Func} &\triangleq \mathsf{Prim} \\
\mathsf{applyFunc}(\mathsf{prm}, \vec{w}) &\triangleq (\mathsf{RunPrim}\ \mathsf{prm}\ \vec{w}, \bullet)
\end{aligned}
$$

SWrapStep

$$
\frac{\neg \mathrm{IsVal}\ e \qquad \mathrm{reducible}(\varnothing, e, \sigma_{ML}) \qquad \forall e'\ \sigma'.\ (e, \sigma) \longrightarrow_{\mathtt{ML}} (e', \sigma') \Rightarrow X((\lceil e' \rceil, K), \mathsf{MLState}(\rho_{ML}, \sigma'))}{((\lceil e \rceil, K), \mathsf{MLState}(\rho_{ML}, \sigma_{ML})) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapHookCall

$$
\frac{\mathsf{fn} \notin \mathsf{dom}\ p \qquad (\vec{V}, \rho_{ML}, \sigma_{ML}) \looparrowright_{\mathrm{M2C}} Y_\rho \qquad \forall \vec{w}\ \rho_C\ \sigma_C.\ Y_\rho(\vec{w}, \rho_C, \sigma_C) \Rightarrow X((\mathsf{Call}\ \mathsf{fn}\ \vec{w}, K \cdot K_{ML}), \mathsf{CState}(\rho_C, \sigma_C))}{((\lceil K_{ML}[\mathsf{call}\ \mathsf{fn}\ \vec{V}] \rceil, K), \mathsf{MLState}(\rho_{ML}, \sigma_{ML})) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapValToC

$$
\frac{([V], \rho_{ML}, \sigma_{ML}) \looparrowright_{\mathrm{M2C}} Y_\rho \qquad \forall w\ \rho_C\ \sigma_C.\ Y_\rho([w], \rho_C, \sigma_C) \Rightarrow X((w, K), \mathsf{CState}(\rho_C, \sigma_C))}{((\lceil V \rceil, K), \mathsf{MLState}(\rho_{ML}, \sigma_{ML})) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapRetToML

$$
\frac{([w], \rho_C, \sigma) \mathrel{\hat{\looparrowright}}_{\mathrm{C2M}} ([V], \rho_{ML}, \sigma_{ML}) \qquad X((\lceil K_{ML}[V] \rceil, K), \mathsf{MLState}(\rho_{ML}, \sigma_{ML}))}{((w, K \cdot K_{ML}), \mathsf{CState}(\rho_C, \sigma_C)) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapToPrimitive

$$
\frac{p[\mathsf{fn}] = \mathsf{prm} \qquad X((\mathsf{RunPrim}\ \mathsf{prm}\ \vec{w}, K), \mathsf{CState}(\rho_C, \sigma_C))}{((\mathsf{Call}\ \mathsf{fn}\ \vec{w}, K), \mathsf{CState}(\rho_C, \sigma_C)) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapPrimMain

$$
\frac{X((\lceil e_{\mathsf{main}} \rceil, \bullet), \mathsf{MLState}((\chi \triangleq \varnothing, \zeta \triangleq \varnothing, \mathrm{rm} \triangleq \varnothing, \sigma_C), \varnothing))}{((\mathsf{RunPrim}\ \mathsf{main}_{e_{\mathsf{main}}}\ [], \bullet), \mathsf{CState}((\chi \triangleq \varnothing, \zeta \triangleq \varnothing, \theta \triangleq \varnothing, \mathrm{rs} \triangleq \varnothing), \sigma_C)) \twoheadrightarrow_{\mathtt{W}} X}
$$

SWrapTerminate

$$
\frac{}{((w, \bullet), \mathsf{CState}(\rho_C, \sigma_C)) \twoheadrightarrow_{\mathtt{W}} X}
$$

Figure 5.9: Operational Semantics of the Wrapper.

over this expression. This primitive expects that it is called "initially," so that the wrapper state is still empty. It then starts the execution of the e_{main}, as defined by SWrapPrimMain. We already mentioned that the FFI primitives are implemented as functions defined by the wrapper. Thus, the wrapper also has a "program" p, which stores all the primitives by their name. Specifically, we have $p = [e_{\mathsf{main}}]_{\mathrm{FFI}} = \{\texttt{"main"} := \mathsf{main}_{e_{\mathsf{main}}}, \texttt{"Int\_val"} := \mathsf{Int\_val}, \ldots\}$. Technically, this program is a parameter of the step relation $\twoheadrightarrow_{\mathtt{W}}$, but we omitted it here for presentation purposes. The rule SWrapToPrimitive dispatches incoming calls to the right primitive, by stepping to the special RunPrim state. With the rules defined in Figure 5.9, we have a working wrapper, which can translate an λ_{ML} heap to a block-level heap that is in theory accessible from λ_{C}. What is missing for making it actually accessible from λ_{C} are the primitives we outlined in Chapter 3, which are implemented functions provided by the wrapper. These are defined in Section 5.3.

Before we can continue by defining primitives, we need to prove that our wrapper never exhibits *no behavior*.

Theorem 5.2 (Absence Of *No Behavior* For The Wrapper) *Let* $(E, K) : \mathsf{Expr}$ *not be a value and* $\rho : \Sigma$. *If* $((E, K), \rho) \twoheadrightarrow_{\mathtt{W}} X$, *then* X *is nonempty.*

Proof By inversion on the step:

- The case of SWrapStep is similar to Theorem 5.1.
- The cases of SWrapHookCall and SWrapValToC follow from the fact that $\looparrowright_{\mathrm{M2C}}$ never exhibits *no behavior*.
- The cases of SWrapRetToML, SWrapToPrimitive, and SWrapPrimMain are trivial.

We now are left with proving that $\looparrowright_{\mathrm{M2C}}$ never exhibits *no behavior*, which means showing that the relation defined by $\hat{\looparrowright}_{\mathrm{M2C}}$ in MLToCDemonic is inhabited, where we get to assume the facts from MLToC. This is true, but the proof is involved. We outline the necessary steps here:

1. We prove that each λ_{ML} value can be serialized to a block-level value, by potentially extending the visibility map χ to χ' and by allocating new immutable blocks into ζ_{newimm}. In particular, ensure by construction that $\zeta_{\mathsf{newimm}} \mathrel{\#\#} \chi$.
2. Similarly, the argument vector V is serialized to block-level values $\vec{v}$.
3. We use this serialize the entire λ_{ML} heap to a block-level heap ζ_{ML}.
4. To construct the address map θ, we first construct its domain, for which we simply add all γ from $\mathsf{dom}\ \zeta'$, from all blocks in ζ', all roots in rm, and $\vec{v}$. This implements the simplest form of garbage collection: no-op garbage collection, where nothing is ever freed.

5. We then actually "inflate" the address map θ, by choosing a fresh C address a for each block-level location γ in its previously constructed domain.

6. We construct $\vec{w}$ using the just-constructed address map θ (which must contain all locations in $\vec{v}$).

7. We similarly re-add the C encodings of all roots from rm to σ_C, yielding σ'_C.

8. The roots set is constructed like so: $rs = \text{dom } rm$.

9. When constructed as such, showing χ', ζ', θ, $\vec{v}$, σ'_C and $\vec{w}$ validate all the required properties is almost always straightforward / by construction. However:

   - To prove $\zeta \mathrel{\#\#} \zeta_{\text{newimm}}$, we use that all new blocks are not in χ, and that $\text{dom } \zeta \subseteq \text{dom } \chi$. We could do without this requirement, but our operational semantics guarantee this is true anyway.

   - To prove $\zeta \mathrel{\#\#} \zeta_{ML}$, we need that $\forall \gamma \ell. \chi[\gamma] = \text{Pub } \ell \to \ell \in \text{dom } \zeta \to \sigma_{ML}[\ell] = ⚡$.

   - Finally, $\text{dom } \sigma_C \mathrel{\#\#} \text{dom } rm$ is needed to show $\sigma_{\text{rest}} \mathrel{\#\#} \sigma_{\text{root}}$ when proving WithRoots (see Figure 5.6). □

5.3 Defining Primitives

Defining the wrapper primitives is relatively straightforward. We distinguish three kinds of primitives, listed in no particular order. The first are the *simple* primitives. What makes them simple is that they never cause garbage collection, and also do not have any non-determinism (except for plain *undefined behavior*). Most primitives are in this category.

Unfortunately, the word *primitive* is now overloaded, since it is ambiguous whether a primitive is part of our formal model, or the primitive as it exists in the real world. We thus introduce the following terms: The *formal* primitive describes the primitive as we model it in our formal model. The *role model* primitive describes how the primitive works in the actual OCaml FFI, not in our formal model of it.

The next category are the *allocating* primitives. These primitives allocate new blocks, which means that they can cause a garbage collector run. The third kind of primitive are the *switching* primitives. These are the primitives that switch from executing C, to executing OCaml. We already know one such primitive: main. This class only contains one further primitive, which is callback.

5.3.1 Callbacks

We start with callback, since the other primitives share some common infrastructure. callback is the formal version of the role model `caml_callback`. To facilitate

$$\frac{\begin{array}{c}([w_{\mathsf{f}}, w_{\mathsf{a}}], \rho_{\mathrm{C}}, \sigma) \mathrel{\hat{\leftrightarrow}}_{\mathrm{C2M}} ([\mathsf{rec}\, f\, x.\, e, V_{\mathsf{a}}], \rho_{\mathrm{ML}}, \sigma_{\mathrm{ML}}) \\ X((\lceil(\mathsf{rec}\, f\, x.\, e)\; V_{\mathsf{a}}\rceil, K), \mathsf{MLState}(\rho_{\mathrm{ML}}, \sigma_{\mathrm{ML}}))\end{array}}{((\mathsf{RunPrim}\ \mathsf{callback}\ [w_{\mathsf{f}}, w_{\mathsf{a}}], K), \mathsf{CState}(\rho_{\mathrm{C}}, \sigma_{\mathrm{C}})) \twoheadrightarrow_{\mathtt{W}} X}\ \textsc{SWrapPrimCallback}$$

$$\frac{(\mathrm{prm}, \vec{w}, \rho_{\mathrm{C}}, \sigma_{\mathrm{C}}) \twoheadrightarrow_{\mathrm{prm}} Y \qquad \forall w'\, \rho'_{\mathrm{C}}\, \sigma'_{\mathrm{C}}.\ Y(w', \rho'_{\mathrm{C}}, \sigma'_{\mathrm{C}}) \Rightarrow X((w', K), \mathsf{CState}(\rho_{\mathrm{C}'}, \sigma'_{\mathrm{C}}))}{((\mathsf{RunPrim}\ \mathrm{prm}\ \vec{w}, K), \mathsf{CState}(\rho_{\mathrm{C}}, \sigma_{\mathrm{C}})) \twoheadrightarrow_{\mathtt{W}} X}\ \textsc{SWrapPrimOther}$$

Figure 5.10: Extensions to the Operation Semantics $\twoheadrightarrow_{\mathtt{W}}$ required to handle primitives.

callbacks, we add the step described by SWrapPrimCallback in Figure 5.10. This primitive switches the state to the OCaml side, so that afterwards, λ_{ML} code can execute. During this switch, the first argument of the primitive, w_{f}, is expected to encode a closure. The new λ_{ML} expression that afterwards starts executing is then exactly this closure, applied to the value V_{a} encoded by w_{a}. This expression then simply reduces inside the wrapper, and eventually returns using SWrapValToC. Of course, this expression can itself make external calls. Since we already defined the linker such that calls can repeatedly go back-and-forth, this causes no issues. Defining callbacks is surprisingly simple, given that introducing them makes the formal setup much more complicated, as otherwise calls from C to the wrapper would not cause mutual recursion (main is defined such that it can only be invoked once). In fact, this simplicity is only superficial. To make this definition this short, we had to introduce modules with their finely-tuned notions of contexts that do not define head redexes, and a linker that can recursively link code. Further, simply re-using the relation $\hat{\leftrightarrow}_{\mathrm{C2M}}$ does not make this definition simpler. We can already foreshadow the other primitives do not take much more effort to define. We have now reached the points where all ingredients necessary to define them are there, and just need to be put together properly.

5.3.2 Allocating Primitives

There are two allocating primitives: alloc and alloc_custom. Unsurprisingly, the definition for both are very similar, with the distinction that alloc_custom allocates a custom block, while alloc only allocates a regular block. To define these primitives, we do not simply extend the $\twoheadrightarrow_{\mathtt{W}}$ relation like we did for callback. Instead, we extend the relation $\twoheadrightarrow_{\mathrm{prm}}$, as introduced by SWrapPrimOther. The relation $\twoheadrightarrow_{\mathrm{prm}}$ allows us to define primitives more concisely. For one, we do not have to state that the step applies to the expression $(\mathsf{RunPrim}\ \mathrm{prm}\ \vec{w}, K)$, but can only focus on the relevant part, namely prm and $\vec{w}$. We also expect all primitives defined using this

$$\textsc{AllocCore}\quad \frac{\gamma \notin \mathrm{dom}\,\chi \qquad \gamma \sim^{\theta'}_{C} a \qquad \mathsf{GcStateSwitch}(\zeta, \theta', rm, \sigma_{\mathsf{rest}}, rs, \sigma'_C) \qquad \rho_C = (\chi[\gamma := k], \zeta[\gamma := blk], \theta', rs)}{\mathsf{AllocCore}(\chi, \zeta, rm, \sigma_{\mathsf{rest}}, a, \rho_C, \sigma'_C, k, blk)}$$

$$\textsc{SWrapPrimAlloc}\quad \frac{\begin{array}{c} n_t = t \qquad n_{\mathsf{size}} \geqslant 0 \\ \mathsf{GcStateSwitch}(\zeta, \theta, rm, \sigma_{\mathsf{rest}}, rs, \sigma_C) \\ \forall \theta' \sigma'_C a.\ \mathsf{AllocCore}(\chi, \zeta, rm, \sigma_{\mathsf{rest}}, a, \rho_C, \sigma'_C, \mathsf{Priv}, \mathsf{B}(t, \mathsf{Mut}, \overset{n_{\mathsf{size}}\ \text{many}}{\underbrace{[0, \ldots, 0]}})) \Rightarrow Y(a, \rho_C, \sigma'_C) \end{array}}{(\mathsf{alloc}, [n_{\mathsf{size}}, n_t], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

$$\textsc{SWrapPrimAllocCustom}\quad \frac{\begin{array}{c} \mathsf{GcStateSwitch}(\zeta, \theta, rm, \sigma_{\mathsf{rest}}, rs, \sigma_C) \\ \forall \theta'\ \sigma'_C\ a\ \iota.\ \mathsf{AllocCore}(\chi, \zeta, rm, \sigma_{\mathsf{rest}}, a, \rho_C, \sigma'_C, \mathsf{Fgn}\ \iota, \mathsf{F}(\mathsf{None})) \Rightarrow Y(a, \rho_C, \sigma'_C) \end{array}}{(\mathsf{alloc\_custom}, [], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

Figure 5.11: Operational Semantics of allocating primitives.

method to evaluate to a λ_C value w within one step. This assumption does not hold for callback and main, which is why they are defined separately. It does hold for all other primitives. There are two allocating primitives: alloc and alloc_custom. The primitive alloc is our formal variant of `caml_alloc`, which is used to allocate a new standard block. The primitive alloc_custom is the formal variant of `caml_alloc_custom`, and allocates custom blocks. The definitions of alloc and alloc_custom are found in Figure 5.11. Since both may cause the garbage collector to run, they first extract the block-level representation of the roots stored in σ_C to a roots map rm. Then, several demonic choices happen, which are described by AllocCore, using rule AllocCore. First, a new block-level location γ is demonically chosen, which must not yet exist in χ. Since our semantics guarantee that $\mathrm{dom}\ \zeta \subseteq \mathrm{dom}\ \chi$, this location is also fresh in ζ. Then, a new garbage collector state θ' is also chosen demonically. This must satisfy the usual condition GcStateSwitch, which also restores the just-extracted roots back into the new C state σ'_C. Additionally, it is also ensured that γ is live in θ'. Specifically, it must be accessible using a. Then, both χ and ζ are extended, but the specific values differ between these two. We note that since visibility maps $\chi : \mathsf{LocMap}$ are by definition injective, the new visibility map χ' must also be injective. For alloc, described in SWrapPrimAlloc, the new block has size n_{size}, and the tag described by n_t, which must actually encode a tag. We then get a new block, which is initially marked as private in χ, and that has tag n_t, size n_{size}, and stores only block-level zeros. Further, this block is mutable. This formally captures all aspects of the role model `caml_alloc` we discussed previously.

For alloc_custom, described in SWrapPrimAllocCustom, the new block is marked as belonging to the foreign value identifier ι. Due to the implicit injectivity requirements in visibility maps, this foreign value identifier is fresh. The custom block itself is $\mathsf{F}(\mathsf{None})$, which denotes that it is uninitialized. Before that block can be read, it must be initialized. We mentioned in Section 3.5 that our formal custom blocks, when compared to their role model, are much more simple. They are so simple, in fact, that allocating a custom block does not need any arguments.

In case it was not already clear from SWrapPrimOther, we explain how rules defined using $\longrightarrow\!\!\!\!\twoheadrightarrow_{\text{prm}}$ are to be read: The left side of that step relation is a pair containing the name of the primitive (here alloc), its arguments (here the two-element list $[n_{\mathsf{size}}, n_{\mathsf{t}}]$), and the current state. The current state is just ρ_C and σ_C, but since all primitives access parts of the state, ρ_C is usually decomposed into its constituent parts. Then, using the familiar way we define demonic choices, we describe the possible target configurations. These are the states that satisfy Y. This target configuration consists of the return value (w' for alloc), and the new state (which, for alloc, is actually defined by AllocCore).

Like before, one thing is missing before we can conclude our discussion of these primitives: We need to prove the absence of *no behavior*. Proving this re-uses some of the machinery from Theorem 5.2, in particular the construction of a new GC state θ'. Since both runtime locations γ and foreign value identifiers ι are countably infinite, it is obvious that there always are ones that are fresh. We already mentioned that simple primitives always determine a unique value. This also means that proving the absence of *no behavior* is trivial for them. We therefore omit proving it, leaving this to the reader.

5.3.3 Simple Primitives

The rules for the remaining primitives are to be found in Figure 5.12.

Converting Integers

Arguably, the simplest primitives are Val_int and Int_val, which are (unsurprisingly) the formal version of their role models **`Val_int`** and **`Int_val`**. Their formal semantics is described by SWrapPrimValInt and SWrapPrimIntVal. Val_int converts an integer n into a C value w that encodes the integer n, as defined by $n \sim_C^\theta w$. Int_val is the inverse operations. The formal model of **`Is_block`**, isblock, is not much harder. This returns 1 when its argument is a λ_{ML} value that represents a block (SWrapPrimIsBlockTrue), and 0 when it represents an integer (SWrapPrimIsBlockFalse). Unlike the role model **`Is_block`**, the formal primitive can only be used on values that actually store a block. If they are used on a λ_C value that used to represent a block, but no longer does, it has undefined behavior. In contrast, the role model **`Is_block`**

SWrapPrimValInt

$$\frac{n \sim^{\theta}_{C} w \qquad Y(w, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{Val\_int}, [n], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimIntVal

$$\frac{n \sim^{\theta}_{C} w \qquad Y(n, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{Int\_val}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimIsBlockTrue

$$\frac{\gamma \sim^{\theta}_{C} w \qquad Y(1, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{isblock}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimIsBlockFalse

$$\frac{n \sim^{\theta}_{C} w \qquad Y(0, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{isblock}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimTag

$$\frac{\gamma \sim^{\theta}_{C} w \qquad \zeta[\gamma] = \mathsf{blk} \qquad \mathsf{tagNumber}(\mathsf{blk}) = n \qquad Y(n, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{read\_tag}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimLength

$$\frac{\gamma \sim^{\theta}_{C} w \qquad \zeta[\gamma] = \mathsf{B}(t,m,\vec{v}) \qquad Y(|\vec{v}|, (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{length}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimField

$$\frac{\gamma \sim^{\theta}_{C} w \qquad \zeta[\gamma] = \mathsf{B}(t,m,\vec{v}) \qquad 0 \leqslant n < |\vec{v}| \qquad v_n \sim^{\theta}_{C} w' \qquad Y(w', (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{Field}, [w, n], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimStoreField

$$\frac{\begin{array}{c}\gamma \sim^{\theta}_{C} w \qquad v \sim^{\theta}_{C} w' \\ \zeta[\gamma] = \mathsf{B}(t,\mathsf{Mut},\vec{v}) \qquad 0 \leqslant n < |\vec{v}| \qquad Y(0, (\chi, \zeta[\gamma := \mathsf{B}(t,\mathsf{Mut},\vec{v}[i := v])], \theta, rs), \sigma_C)\end{array}}{(\mathsf{Store\_field}, [w, n, w'], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimReadCustom

$$\frac{\gamma \sim^{\theta}_{C} w \qquad \zeta[\gamma] = \mathsf{F}(w') \qquad Y(w', (\chi, \zeta, \theta, rs), \sigma_C)}{(\mathsf{read\_custom}, [w], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimWriteCustom

$$\frac{\gamma \sim^{\theta}_{C} w \qquad \zeta[\gamma] = \mathsf{F}(\_) \qquad Y(0, (\chi, \zeta[\gamma := \mathsf{F}(w')], \theta, rs), \sigma_C)}{(\mathsf{write\_custom}, [w, w'], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimRegisterRoot

$$\frac{a \notin rs \qquad Y(0, (\chi, \zeta, \theta, rs \cup \{a\}), \sigma_C)}{(\mathsf{registerroot}, [a], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

SWrapPrimUnregisterRoot

$$\frac{a \notin rs \qquad Y(0, (\chi, \zeta, \theta, rs \cup \{a\}), \sigma_C)}{(\mathsf{unregisterroot}, [a], (\chi, \zeta, \theta, rs), \sigma_C) \twoheadrightarrow_{\mathrm{prm}} Y}$$

Figure 5.12: Operational Semantics of simple primitives.

still returns `1` for arguments that did not survive the last garbage collection, since it only looks at the least significant bit, which is unaffected by what the pointer points to.

Tags and Length

The primitive `read_tag` is used to read the tag of a block. Its role model, the macro `Tag_val`, uses pointer arithmetic to access the tag of a block, and is thus also an lvalue, so that it can be used to change the tag of a block. Since we do not support changing the tag, our formal version `read_tag` simply returns the tag. To do so, as shown in SWrapPrimTag, the function tagNumber is used, which is defined in Figure 5.4. The actual numeric values returned by the formal primitive are the same ones that the role model primitive returns.

The primitive `length`, defined in SWrapPrimLength is the formal version of the role model primitive `Wosize_val`. For a regular block, this primitive returns the number of fields stored by this block. The formal primitive can not be used to get the size of a custom block, or of a callback block. The role model primitive does work on such blocks. For custom blocks, which in actual OCaml can have an arbitrary length, it returns this length (in multiples of `sizeof(value)`). For callbacks, it returns 2, since callbacks are encoded as a block of size 2. We choose not to model this behavior, especially since the size of a custom block would always be 1 in our model. Modelling this would not be hard: One would just need to define a function $\mathsf{blockLength} : \mathsf{Block} \to \mathbb{Z}$, similar to tagNumber.

Roots

While our state is on the C side, it only contains a roots *set* rs, that contains each C address a that is registered as a root. These locations are only required to actually store a λ_C value encoding a block-level value when a garbage collector run happens, *i.e.*, when the state is switched, or an allocation primitive is invoked. We have previously described that we simplified the formal model down to two primitives: `registerroot`, which roughly is a formal model of the role model of the primitive `caml_register_global_root`, and `unregisterroot`, which similarly models the role model for `caml_unregister_global_root`. The rule for `registerroot` is found in SWrapPrimRegisterRoot, the for `unregisterroot` is in SWrapPrimUnregisterRoot. Since primitives operate on the C side, they only interact with the roots set. Registering a root then means that the root is added to the set, and unregistering a root means that the root is removed from the set. Additionally, the rules make it *undefined behavior* to register a root that is already registered, or to unregister a root that is not actually a root. Again, this seems pretty simple, but only because the machinery that actually ensures roots remain consistent is already baked into the operational semantics, in particular the definitions around WithRoots from figure Figure 5.6.

Accessing and Modifying Regular Blocks

The primitive Field is the formal model of the role model primitive `Field`. In actual OCaml, `Field` is a macro that computes the proper offset for a `value` encoding a pointer. Thus, `Field` is an lvalue, which means that it can be stored to. Such stores are usually unsafe (see the discussion on uninitialized data in Section 3.5), and our formal model does not allow such direct stores. Instead Field is just a normal function that returns the requested value. To actually safely write to a block, the role model primitive `Store_field` exists. This is a macro that wraps a call to `caml_modify`, which is not a macro, but rather a real function implemented in the OCaml runtime. Our formal primitive Store_field models the behavior of the role model macro `Store_field`. The formal macros Field, defined by SWrapPrimField, and Store_field, defined by SWrapPrimStoreField, both take similar arguments. The first argument is a λ_C value w, which must encode a block-level location γ. This location denotes the block $\mathsf{blk} = \zeta[\gamma]$ these primitives operate on. It must be a regular block, storing the block-level values $\vec{v}$. For Store_field, it must also be mutable. The second argument n is an index, describing which field of the block is to be read/modified. This index must be in bounds, formally required by asserting $0 \leqslant n < |\vec{v}|$

The Field formal primitive then simply returns the nth field v_n, in encoded form. Note that the encoding w' is chosen angelically, but this is not an issue: This choice is unique, as θ is injective.

The Store_field formal primitive takes a third argument, the λ_C value w', which must also encode a block-level value v'. It then modifies the block by storing v' at index i. Strictly speaking, it modifies ζ, by storing the modified block at its location γ. Since this primitive is only executed for its side effects, it returns 0.

Accessing and Modifying Custom Blocks

The formal primitives for operating on custom blocks are interesting, since they lack proper role models. In actual OCaml, the macro `Data_custom_val` computes a pointer to the user-usable area of a custom block (which in actual OCaml can be larger than one machine word). Accessing and modifying this data can then be done by operating on this pointer using normal C loads and stores. Since our formal model draws a hard barrier between the λ_C heap σ_C and the runtime memory ρ_C, we can not replicate this behavior. Instead, we have two formal primitives, read_custom and write_custom, that read and write the contents of a custom block. The first argument of both primitives is a λ_C value w describing the block-level location γ at which the desired custom block is stored.

For read_custom, this is the only argument. The custom block $\mathsf{F}(w')$ must store λ_C value w', so it must in particular not be uninitialized. This value w' is then returned.

For write_custom, we do not care what is stored in the foreign block. In particular, F(_) is to be read so that it also allows storing to an uninitialized block. We then update the value in that block to w', which was passed as the second argument. More formally, we update ζ to store this updated block at its location γ.

Conclusion

We have now discussion the definition of all primitives, which completes our discussion of the operational semantics. We again note that the definition of the primitives only seems simple. These primitives are tightly coupled to the overall notion of the wrapper state. The list of primitives is also complete, in a sense, in that it covers all the features available in our definition of the wrapper. While some primitives still could be defined, like one to set the tag, we do not define these since these are not realistic. For this concrete example, we mentioned earlier that changing the tag is considered *undefined behavior*, at least by us.

Chapter 6

The Combined Program Logic

In this chapter, we build a separation logic to reason about programs written in OCaml and C, that is adequate for the operational semantics defined in Chapter 5.

Before we can start, we must find a weakest precondition for modules with dual non-determinism. This is accomplished in Section 6.1. The adequacy theorem is interesting, since it features co-inductive traces and Transfinite Iris [45]. We then, in Section 6.1.2 briefly discuss how linking programs works at the program logic level.

In Section 6.2, we gently introduce the program logic, by reasoning through some examples already explained in Chapter 3. We then discuss the formal construction in Section 6.3.

This concludes our discussion of the original paper. Section 6.4 then introduces the first material going beyond the paper–it generalized the View Reconciliation laws to work with fractional points-tos.

6.1 Weakest Preconditions for Linkable Modules

6.1.1 Program Logic and Adequacy

We begin by (again) changing our weakest precondition, so that it supports angelic and demonic non-determinism. We start by expressing the property we want to have, namely adequacy using the co-inductive traces of Section 5.1.

Theorem 6.1 (Adequacy for Modules) *Let* p *be a program,* σ *be a state such that* $\vdash \mathrm{SI}(\sigma)$*, and* $Q' : \Sigma \to \mathsf{Val} \to \mathbb{P}\mathsf{rop}$ *such that* $\forall v\, \sigma'.\, \mathrm{SI}(\sigma') * Q(v) \Rrightarrow\!\!* \; Q'(\sigma', v)$*. Then the following holds:*

$$\vdash \mathsf{wp}\; e \,@\, p, \bot \,\{Q\} \implies (e, \sigma) \twoheadrightarrow^{\mathsf{trace}} \{(v, \sigma') \mid Q'(\sigma', v)\}$$

Intuitively, this proof allows us to turn the proof of a weakest precondition into a process[1] that describes how the angelic choices are resolved. When combined

[1]Co-inductive predicates can be intuitively visualized as a generating process.

with a demon resolving the demonic choices, this process can be used to construct executions, and all the executions the demon can construct will only terminate in states satisfying the postcondition (or step forever). Our goal is now to construct a weakest precondition with that property. Without further ado, here it is:

Definition 6.2 (Weakest Precondition for Modules)

$$\mathsf{wp}\ e\,@\,p, \Psi\,\{Q\} \triangleq \forall \sigma.\, \mathrm{SI}(\sigma) \Rrightarrow\!\!\ast \begin{cases} \mathrm{SI}(\sigma) \ast Q(v) & e = \overline{v} \\[1ex] \begin{array}{l} \mathrm{fn} \notin \mathsf{dom}\ \ p \ast \\ \mathrm{SI}(\sigma) \ast \mathit{atBoundary} \ast \\ \dot{\mid\!\Rrightarrow} \exists Q'.\, \Psi\ \mathrm{fn}\ \vec{v}\ Q' \ast \\ \quad \triangleright \forall v'.\, Q'\ v' \mathrel{-\!\!\ast} \\ \qquad \mathit{atBoundary} \mathrel{-\!\!\ast} \\ \qquad \mathsf{wp}\ \mathsf{K}[v']\,@\,p, \Psi\,\{Q\} \end{array} & e = \mathsf{K}[\mathsf{call}\ \mathrm{fn}\ \vec{v}] \\[1ex] \begin{array}{l} \exists X.\, (e, \sigma) \longrightarrow\!\!\!\!\rightarrow X \ast \\ \forall e'\, \sigma'.\, (e', \sigma') \in X \mathrel{-\!\!\ast} \\ \quad \dot{\mid\!\Rrightarrow} \triangleright \dot{\mid\!\Rrightarrow} \mathrm{SI}(\sigma') \ast \mathsf{wp}\ e'\,@\,p, \Psi\,\{Q\} \end{array} & \textit{otw.} \end{cases}$$

Comparing this definition to the one for regular languages (Definition 4.1), we see that the first case is the same. The second case for external calls has changed by introducing the $\mathit{atBoundary}$ token. This token is a separation logic assertion, defined by each language, that describes whether the program is in a state where it is able to execute external calls. We further describe it in Section 6.1.2. The third and final case sees the most interesting changes. We now have the quantifier alternation required for encoding angelic and demonic choices. When proving a weakest precondition (verifying a program), we (the prover) must first provide an X such that $(e, \sigma) \longrightarrow\!\!\!\!\rightarrow X$. In other words, we need to resolve the angelic choice. Once chosen, we must then handle all configurations $(e', \sigma') \in X$, since the demon could force us to continue with either of them. Thus, we must prove the weakest precondition for all $(e', \sigma') \in X$. Also note that we no longer need to prove that our state is $\mathit{reducible}$. The reason for this is subtle, and related to a weirdness in the standard interpretation of regular operational semantics: In regular semantics, if the step relation had multiple possible targets, we interpreted this as a demonic choice. The case of this choice being empty, however, was not a demonic choice over the empty set (*i.e., no behavior*), but instead *undefined behavior*. This requires one to, in addition of handling every successor state, prove that such a successor state exists. With multi-relation-based semantics, this becomes much cleaner: Undefined behavior is an angelic choice over the empty set. Since the weakest precondition already forces us to resolve the angelic choice by choosing a suitable value, this implicitly enforces that the set of possible angelic choices is not empty, and thus that there is no UB. What the weakest precondition does not prevent is *no behavior*. In fact, if a configuration were to exhibit *no behavior*, its weakest precondition would just be true, since this amounts to proving a universal quantifier over an empty set. Luckily, we forbid our modules from having *no behavior*, so we do not need to worry about it further.

We next look at the proof of Theorem 6.1. This is where transfinite Iris is used, since we need to extract the existential quantifier $\exists X.\, (e, \sigma) \longrightarrow\!\!\!\!\!\rightarrow X$ from a proof of the weakest precondition. It is only thanks to transfinite Iris that we can prove a trace-based adequacy theorem. The one from before, based on (prefixes of) executions, was necessary because the proof required carefully inspecting the length of the execution, to choose an initial step-index that is large enough to cover the entire execution. This was necessary since without the existential property, the only way for extracting an existential proven within the logic was to specify the step-index a priori. In Transfinite Iris, the axiom of choice to instead is used (implicitly in the proof of the existential property) to find a witness that is valid for all step-indexes, allowing us to co-inductively extract an potentially infinite trace (by leveraging an transfinite step-index that is of greater infinity).

Weakest Precondition Lifting As final evidence that our new weakest precondition is properly defined, we can observe that it is *equivalent* to the old weakest precondition when instantiated with the lifting module $\uparrow\lambda$:

Theorem 6.3 (Agreement of Weakest Preconditions and Lifting) *Let λ be a language, with $e : \mathsf{Expr}_\lambda$, $p : \mathsf{Prog}_\lambda$ $\Psi : \mathsf{Proto}_\lambda$ and $Q : \mathsf{Val}_\lambda \to iProp$. Then the weakest preconditions on λ and $\uparrow\lambda$ agree.*

$$\underbrace{\mathsf{wp}_\lambda\ e \mathbin{@} p, \Psi\,\{Q\}}_{\mathsf{wp}\ \textit{for a language}} \mathrel{-\!\!*\!*} \underbrace{\mathsf{wp}_{\uparrow\lambda}\ e \mathbin{@} p, \Psi\,\{Q\}}_{\mathsf{wp}\ \textit{for a module}}$$

We can also use this to actually prove the adequacy theorem of Theorem 4.2.

Proof (of Theorem 4.2)
Given $\vdash \mathsf{wp}_\lambda\ e \mathbin{@} p, \bot\,\{Q\}$ as well e', σ' such that $(e, \sigma) \longrightarrow^{p*} (e', \sigma')$, we must show $Q'(\sigma', e')$. We have $\vdash \mathsf{wp}_{\uparrow\lambda}\ e \mathbin{@} p, \bot\,\{Q\}$ by Theorem 6.3. Using Theorem 6.1, we get a co-inductive trace that only terminates in values satisfying Q'. We proceed by induction on the execution $(e, \sigma) \longrightarrow^{p*} (e', \sigma')$, and feed each step into the co-inductive trace, until either reaches a value, in which case we are done. If the inductive trace $\longrightarrow^{p*}$ terminates first, without reaching a value, than the co-inductive trace guarantees that we are still safe. □

This lemma is again one of the lemmas that enable language-locality. By being able to lift verification carried out using the regular weakest preconditions to the one using modules, we allow these results to be used by the linking operator, defined in the next subsection.

6.1.2 Weakest Preconditions and Linking

In Section 4.1, we defined intra-language lifting and proved the following theorem (Theorem 4.4) for it:

$$\textsc{WP-Link-Intra}\quad \frac{(\Psi_2 \sqcup \Psi_{\text{axiom}}) \vdash p_1 : \Psi_1 \qquad (\Psi_1 \sqcup \Psi_{\text{axiom}}) \vdash p_2 : \Psi_2 \qquad p_1 \mathrel{\#\#} p_2}{\Psi_{\text{axiom}} \vdash p_1 \cup p_2 : \Psi_1 \sqcup \Psi_2}$$

The theorem for correctness of the linking operator, which we only define in Appendix A, looks very similar:

Theorem 6.4 (Correctness of The Linking Operator) *Let λ_1, λ_2 be linkable modules agreeing on the linkage model. Then their programs can then be linked into programs of $\lambda_1 \oplus \lambda_2$:*

$$\textsc{WP-Link-Modules}\quad \frac{\begin{array}{c} p_1 \mathrel{\#\#} p_2 \qquad \Psi_{\text{axiom}} \mathrel{\#\#} (\mathsf{dom}\ p_1 \cup \mathsf{dom}\ p_2) \\ (\Psi_2 \sqcup \Psi_{\text{axiom}}) \vdash_{\lambda_1} p_1 : \Psi_1 \qquad (\Psi_1 \sqcup \Psi_{\text{axiom}}) \vdash_{\lambda_2} p_2 : \Psi_2 \end{array}}{\Psi_{\text{axiom}} \vdash_{\lambda_1 \oplus \lambda_2} p_1 \cup p_2 : \Psi_1 \sqcup \Psi_2}$$

The theorem refers to *linkable* modules. A module is linkable when its public, private, and overall state can be split and merged at the places the linking operator requires it to, as discussed in Section 5.1.3 As shown in Figure A.2, a merge happens whenever the state switches from Boundary to StateL or StateR (compare SLinkHandleCallL, SLinkReturnL). The state is split when changing in the opposite direction, as described by SLinkToExtCallL and SLinkToValL. Remember that the linking operator required its modules to have these three kinds of state. In order to verify the linking operator, we also need three kinds of state interpretations. We have the three state interpretations for the three kinds of state. Additionally, we have the boundary token $\mathsf{atBoundary}$, which we already included in Definition 6.2, the definition of the weakest precondition for modules. To be linkable, these state interpretations have to satisfy the three laws SIJoin, SISplit, and SIAtBoundary, shown in Figure 6.1. The first two describe that the state interpretations behave according to Split, the pure relation describing when public and private states can be split and joined. Specifically, they specify that the requirements on top imply (via the magic wand) the conclusion, but after performing an update, following the syntactic sugar $\Rrightarrow\!\!\ast$. Specifically, SIJoin requires that joining is always possible. Splitting is more complicated, with SISplit merely requiring that if Split tells us that a split it possible, this is also possible in the program logic. We only know that a split is possible when we are *at the boundary*, as described by $\mathsf{atBoundary}$. The weakest precondition

$$\begin{aligned}
\mathrm{SI} &: \Sigma \to \mathit{iProp}\\
\mathrm{SI}_{\mathsf{pub}} &: \Sigma_{\mathsf{pub}} \to \mathit{iProp}\\
\mathrm{SI}_{\mathsf{priv}} &: \Sigma_{\mathsf{priv}} \to \mathit{iProp}\\
\mathsf{atBoundary} &: \mathit{iProp}
\end{aligned}$$

$$\textsc{SIJoin}\quad \frac{\mathrm{SI}_{\mathsf{priv}}(\sigma_{\mathsf{priv}}) \qquad \mathrm{SI}_{\mathsf{pub}}(\sigma_{\mathsf{pub}})}{\exists \sigma.\, \mathsf{Split}(\sigma, \sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv}}) * \mathrm{SI}(\sigma)}\,{*}
\qquad\qquad
\textsc{SISplit}\quad \frac{\mathrm{SI}(\sigma) \qquad \mathsf{Split}(\sigma, \sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv}})}{\mathrm{SI}_{\mathsf{priv}}(\sigma) * \mathrm{SI}_{\mathsf{pub}}(\sigma)}\,{*}$$

$$\textsc{SIAtBoundary}\quad \frac{\mathsf{atBoundary} \qquad \mathrm{SI}(\sigma)}{\exists \sigma_{\mathsf{pub}}\ \sigma_{\mathsf{priv}}.\, \mathsf{Split}(\sigma, \sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv}})}\,{*}$$

Figure 6.1: Program Logic definitions for linking.

on modules requires that this token is available around external calls, but makes no requirements otherwise. We see later that this is used by the wrapper module, which is at a boundary only when its state is on the C side. For the lifting module $\uparrow\lambda$, this boundary token is always true. The linking module itself is at a boundary when it is the Boundary state, and both its contained languages are at the boundary. (This is only relevant when linking more than two modules.)

The proof of Theorem 6.4 is now essentially the same as that of Theorem 4.4. The proof is more complicated since there is more stuff: The boundary tokens $\mathsf{atBoundary}$ of both languages need to be pushed around, and the sequence of administrative states outlined in Figure A.2 must be followed. But at the core, the proof still follows the execution of one side, until that side makes an external call, which is then resolved properly. With this, we have all machinery in place to actually construct a program logic for the wrapper.

6.2 Verifying Glue Code By Example

We have now, after nearly 91 pages, finally arrived at the section where we truly discuss the logical foundations of language interoperability between OCaml and C. Before, this has sometimes also been referred to as a program logic for the wrapper. It turns out that, when using our logic to verify a multi-language program, almost no time is spend working "within" the wrapper. Instead, one has to verify glue code, which is written in regular C, with a large amount of FFI primitives sprinkled in. These FFI primitives are implemented as external calls from C to the wrapper, where they are implemented by operating on the private wrapper state. Our goal is to verify glue code, and we thus have to reason about the FFI primitives as external calls. To do so, we define a protocol Ψ_{FFI}, that gives the program logic rules for all the FFI primitives. More specifically, we define the reasoning rules for each of the primitives (except for main, which is special), and then combine them into a large protocol: $\Psi_{\mathsf{FFI}} \triangleq \Psi_{\mathsf{Int\_val}} \sqcup \Psi_{\mathsf{Val\_int}} \sqcup \cdots$. We now first discuss how these specifications look like, what constructs there are in our program logic, and which laws we expect them to satisfy. The actual implementation of this program logic is only discussed in Section 6.3.

6.2.1 A First Example

Let's start by discussing the `plus1` program we saw in Chapter 3, reproduced here in formal syntax.

$$
\begin{aligned}
&e_{\mathsf{plus1}} \triangleq \\
&\quad \mathsf{let}\ n\ =\ \mathsf{call}\ \text{caml\_plus1}\ [41]\ \mathsf{in} \\
&\quad \mathsf{assert}(n = 42)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\text{caml\_plus1}(v) \triangleq \\
&\quad \mathsf{let}\ n\ =\ \mathsf{Int\_val}(v)\ \mathsf{in} \\
&\quad \mathsf{let}\ m\ =\ n + 1\ \mathsf{in} \\
&\quad \mathsf{Val\_int}(m)
\end{aligned}
$$

We use the syntactic sugar $\mathsf{Val\_int}(n)$ for $\mathsf{call}\ \mathsf{Val\_int}\ [n]$, so that the formal syntax looks more similar to actual C. The function already uses three external calls, one in λ_{ML}, and two in λ_{C}. As a first step towards verifying the programs, we have to give specifications to the external calls. The external call to caml_plus1 is easy to specify, it has the following protocol:

$$\Psi_{\mathsf{plus1ML}} \triangleq \forall n.\ \langle \top \rangle\ \text{caml\_plus1}\ [n]\ \langle m.\ m = n + 1 \rangle$$

With this specification for external calls, we are able to prove that the OCaml program executes successfully and in particular that the assert is satisfied. Of course, this verification still has a hole: We need to prove that the C implementation satisfies the OCaml specification Ψ_{plus1} given above. But what does it even mean for our C code to satisfy such a specification? The answer is that our C code has to satisfy a *translated specification*, where we are given C values representing block-level values that themselves represent the original OCaml values, in a way that roughly matches the operational semantics rules outlined by MLToC. We come back to formally pinning down the precise specification we want caml_plus1 to satisfy later. Let us for now ignore the ML-to-block-level encoding and assume that we are only given a value w that encodes a block-level integer n. In the last chapter, specifically Figure 5.6, we already defined $n \sim_{\mathrm{C}}^{\theta} w$, which formerly expresses that w encodes n. Before we can discuss potential specifications for caml_plus1, we need to specify the behavior of the primitives used in it. We start with Int_val. Its specification is

$$\begin{aligned}
\Psi_{\mathsf{Int\_val}} &\triangleq \forall w\, n, \theta.\ \langle n \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle\ \mathsf{Int\_val}\ [w]\ \langle m.\ m = n * \mathsf{GC}(\theta)\rangle \\
\Psi_{\mathsf{Val\_int}} &\triangleq \forall n, \theta.\ \langle \mathsf{GC}(\theta)\rangle\ \mathsf{Int\_val}\ [n]\ \langle w.\ n \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle \\
\Psi_{\mathsf{isblock}} &\triangleq \forall w\, v\, \theta.\ \langle v \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle\ \mathsf{Int\_val}\ [w]\ \langle m.\ \mathrm{IsBlock}(v) = m * \mathsf{GC}(\theta)\rangle \\
\Psi_{\mathsf{registerroot}} &\triangleq \forall a\, w\, v\, \theta.\ \langle a \mapsto_{C} w * v \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle\ \mathsf{registerroot}\ [a] \\
&\qquad\qquad \langle 0.\ a \mapsto_{\mathsf{root}} v * \mathsf{GC}(\theta)\rangle \\
\Psi_{\mathsf{unregisterroot}} &\triangleq \forall a\, v\, \theta.\ \langle a \mapsto_{\mathsf{root}} v * \mathsf{GC}(\theta)\rangle\ \mathsf{unregisterroot}\ [a] \\
&\qquad\qquad \langle 0.\ \exists w.\ a \mapsto_{C} w * v \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle
\end{aligned}$$

$$\textsc{IsBlockLoc}\quad \mathrm{IsBlock}(\gamma) = 1 \qquad\qquad \textsc{IsBlockInt}\quad \mathrm{IsBlock}(n) = 0$$

$$\textsc{CWP-Load-Root}\quad \frac{a \mapsto^{d}_{\mathsf{root}} v \qquad \mathsf{GC}(\theta) \qquad \forall w.\ a \mapsto^{d}_{\mathsf{root}} v \mathrel{-\!\!*} v \sim^{\theta}_{C} w \mathrel{-\!\!*} \mathsf{GC}(\theta) \mathrel{-\!\!*} \mathsf{wp}\ w \mathbin{@} p, \Psi\, \{Q\}}{\mathsf{wp}\ {*a} \mathbin{@} p, \Psi\, \{Q\}}\,*$$

$$\textsc{CWP-Store-Root}\quad \frac{a \mapsto_{\mathsf{root}} - \qquad v \sim^{\theta}_{C} w \qquad \mathsf{GC}(\theta) \qquad a \mapsto_{\mathsf{root}} v \mathrel{-\!\!*} \mathsf{GC}(\theta) \mathrel{-\!\!*} \mathsf{wp}\ 0 \mathbin{@} p, \Psi\, \{Q\}}{\mathsf{wp}\ {*a} \leftarrow w \mathbin{@} p, \Psi\, \{Q\}}\,*$$

Figure 6.2: Specifications for primitives not operating on blocks.

stated as a C specification, and any proof about the correctness of caml_plus1 needs to use it.

$$\Psi_{\mathsf{Int\_val}} \triangleq \forall w\, n.\ \langle n \sim^{\theta}_{C} w\rangle\ \mathsf{Int\_val}\ [w]\ \langle m.\ m = n\rangle$$

This specification is "almost" correct. Only one detail is missing: What is θ? Remember that this is supposed to indicate the current state of the garbage collector. When it changes, the garbage collector has run, and all `value`s pointing to blocks become unusable, if we did not root them. But how do we know what the current state of the garbage collector is? The answer is that we introduce a new separation logic token for it–the *GC* token $\mathsf{GC}(\theta)$. This token, when present, tells us that θ is the current state of the garbage collector. As we see later, this token can also be understood as denoting the permission to access the runtime memory in general. But for now, we can simply include it in our specification of Int_val to ensure that θ is indeed the correct version. The specification is very similar to the one given to Val_int, which can both be found in Figure 6.2. Unlike $v \sim^{\theta}_{C} w$, which is a pure proposition and therefore persistent, the GC token $\mathsf{GC}(\theta)$ is not persistent. In fact, it is exclusive. Thus, the specification of Val_int must give it back at the end. With this, we can prove following C specification for caml_plus1:

$$\forall w\, n, \theta.\ \langle n \sim^{\theta}_{C} w * \mathsf{GC}(\theta)\rangle\ \text{caml\_plus1}\ [w]\ \langle w'.\ (n+1) \sim^{\theta}_{C} w' * \mathsf{GC}(\theta)\rangle$$

Unfortunately, this is not yet sufficient to establish that the λ_C implementation of caml_plus1 matches the λ_{ML} specification. It only contains part of the translation. Currently, we simply assume that the block-level value we start with is a block-level integer, but we only know that our function is called with a λ_{ML} integer. What we implicitly assumed is that OCaml integers are represented as block-level integers. Of course, we know this is true, but we want to establish this formally. To do so, we need a new relation $V \sim_{ML} v$ describing that v is the block-level representative of the λ_{ML} value V. Before, we could simply re-use $v \sim^{\theta}_{C} w$ from Figure 5.6 to define a similar encoding relation between block-level and λ_C values. We could also now reuse IsVal from Figure 5.5 to track when a block-level value v represents a λ_{ML} value V. However, this would us require to also always have available the total visibility map χ, and the total block-level heap ζ. But this is counter to the very foundation of Separation Logic, which was introduced to allow us to *locally* reason about the content of heaps, without having (or even being able) to care about what is stored in the areas we do not modify. Note that this kind of local reasoning is not possible for the garbage collector state θ. This map changes when the garbage collector runs, and if that happens, it changes completely, not just locally. Thus, it does not make sense to own only a fragment of this heap. Instead, the GC token $\mathsf{GC}(\theta)$ expresses full ownership of the entire address map θ, which is needed every time the garbage collector is run. For the visibility map χ and the block-level heap ζ, it is indeed the case that these only change locally (ignoring, for now, the fact that half the block-level heap ζ is removed when the state switches sides). Thus, we can develop a more standard Separation Logic theory for these, where ownership is indeed split into per-location tokens. The Separation Logic theory for these uses the resources shown in Figure 6.3. We leave the definition of some resources open until Section 6.3. For these, we only give an intuitive meaning describing which permission they are supposed to describe. The others, which are better understood as syntactic sugar, are defined using these axiomatic resources. While the figure already shows many resources, we introduce all of them gradually. We first focus on the OCaml-to-block-level representation relation $V \sim_{ML} v$. Intuitively, this is a separation logic counterpart of $\mathrm{IsVal}(\chi, \zeta, V, v)$, and thus denotes that the block-level value v represents the λ_{ML} value V. As such, it closely mirrors the definition of IsVal from Figure 5.5. It is, however, a separation logic relation. It is not indexed by χ and ζ, but instead uses separation logic to reason about their content locally. Its definition is not opaque since it defines how, precisely, λ_{ML} values are encoded. Knowing this is obviously necessary to formally validate glue code, and can not be substituted by some axioms. Since $V \sim_{ML} v$ is only defined using persistent propositions, it is itself persistent. As the block-level-to-C relation

$$
\begin{aligned}
\mathsf{GC}(\theta) &\triangleq \text{general runtime access; current GC state is } \theta\text{; exclusive}\\
v \sim_{C}^{\theta} w &\triangleq w \text{ encodes } v \text{ at GC state } \theta\text{, defined in Figure 5.6; pure}\\
V \sim_{\mathrm{ML}} v &\triangleq v \text{ represents } V\text{, defined below; persistent}\\
\mathsf{isLoc}(\ell, \gamma) &\triangleq \gamma \text{ represents location } \ell\text{, \textit{i.e.}, } \chi[\gamma] = \mathsf{Pub}\,\ell\text{; persistent}\\
\mathsf{isForeign}(\iota, \gamma) &\triangleq \gamma \text{ represents foreign identifier } \iota\text{, \textit{i.e.}, } \chi[\gamma] = \mathsf{Fgn}\,\iota\text{; persistent}\\
\mathsf{isPriv}^{d}(\gamma) &\triangleq \gamma \text{ is private, \textit{i.e.}, } \chi[\gamma] = \mathsf{Priv}\text{; fractional with } d : \mathbb{Q}_{\boxdot}\\
\gamma \mapsto_{\mathrm{FFI}}^{d} \mathsf{blk} &\triangleq \gamma \text{ points to block } \mathsf{blk}\text{, \textit{i.e.}, } \zeta[\gamma] = \mathsf{blk}\text{; fractional with } d : \mathbb{Q}_{\boxdot}\\
\gamma \mapsto_{\mathsf{clos}} \mathsf{rec}\, f\, x.\, e &\triangleq \gamma \mapsto_{\mathrm{FFI}}^{\square} \mathsf{C}(\mathsf{rec}\, f\, x.\, e)\\
\gamma \mapsto_{\mathsf{cstm}}^{d} w &\triangleq \gamma \mapsto_{\mathrm{FFI}}^{d} \mathsf{F}(w) \ast \exists \iota.\, \mathsf{isForeign}(\iota, \gamma)\\
\gamma \mapsto_{\mathsf{blk}[t|\mathsf{mut}]}^{d} \vec{v} &\triangleq \exists \ell.\, \gamma \mapsto_{\mathrm{FFI}}^{d} \mathsf{B}(\mathsf{Mut}, t, \vec{v}) \ast \mathsf{isLoc}(\ell, \gamma)\\
\gamma \mapsto_{\mathsf{blk}[t|\mathsf{imm}]} \vec{v} &\triangleq \gamma \mapsto_{\mathrm{FFI}}^{\square} \mathsf{B}(\mathsf{Imm}, t, \vec{v})\\
\gamma \mapsto_{\mathsf{blk}[t|\mathsf{fresh}]}^{d} \vec{v} &\triangleq \gamma \mapsto_{\mathrm{FFI}}^{d} \mathsf{B}(\mathsf{Mut}, t, \vec{v}) \ast \mathsf{isPriv}^{1}(\gamma)\\
a \mapsto_{\mathsf{root}}^{d} v &\triangleq a \text{ is a root storing } v\text{; fractional with } d : \mathbb{Q}_{\boxdot}\\
\mathsf{atInit} &\triangleq \text{permission to invoke } \mathsf{main}\text{; exclusive}
\end{aligned}
$$

$$
\textsc{SimInt}\quad n \sim_{\mathrm{ML}} n
\qquad\qquad
\textsc{SimTrue}\quad \mathsf{true} \sim_{\mathrm{ML}} 1
\qquad\qquad
\textsc{SimFalse}\quad \mathsf{false} \sim_{\mathrm{ML}} 0
\qquad\qquad
\textsc{SimUnit}\quad \langle\rangle \sim_{\mathrm{ML}} 0
$$

$$
\textsc{SimLoc}\quad \frac{\mathsf{isLoc}(\ell, \gamma)}{\gamma \sim_{\mathrm{ML}} \ell}\ast
\qquad\qquad
\textsc{SimForeign}\quad \frac{\mathsf{isForeign}(\iota, \gamma)}{\gamma \sim_{\mathrm{ML}} \text{ⓘ}}\ast
\qquad\qquad
\textsc{SimClosure}\quad \frac{\gamma \mapsto_{\mathsf{clos}} \mathsf{rec}\, f\, x.\, e}{\gamma \sim_{\mathrm{ML}} \mathsf{rec}\, f\, x.\, e}\ast
$$

$$
\textsc{SimPair}\quad \frac{\gamma \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v, v'] \qquad v \sim_{\mathrm{ML}} V \qquad v' \sim_{\mathrm{ML}} V'}{\gamma \sim_{\mathrm{ML}} \langle V, V' \rangle}\ast
$$

$$
\textsc{SimInl}\quad \frac{\gamma \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v] \qquad v \sim_{\mathrm{ML}} V}{\gamma \sim_{\mathrm{ML}} \mathsf{inl}\ V}\ast
\qquad\qquad
\textsc{SimInr}\quad \frac{\gamma \mapsto_{\mathsf{blk}[1|\mathsf{imm}]} [v] \qquad v \sim_{\mathrm{ML}} V}{\gamma \sim_{\mathrm{ML}} \mathsf{inr}\ V}\ast
$$

Figure 6.3: The Separation Logic theory for reasoning about glue code.

is also persistent (it is in fact pure), this means that glue code can freely copy λ_C values that represent λ_{ML} values, without running into ownership issues.

We now come back to our example function, caml_plus1. It is now possible for us to prove a specification for it that is sufficient to allow us to claim that it can be safely linked with λ_{ML}. Since actually combining everything into a state where we can prove adequacy is a bit involved, we leave this open until Section 6.2.5. Using SimInt and the specifications for Val_int and Int_val introduced above, we can verify the following specification for caml_plus1:

$$\begin{aligned}\Psi_{\mathsf{plus1C}} \triangleq \forall n\, v\, w\, \theta.\ &\langle n \sim_{ML} v * v \sim_C^{\theta} w * \mathsf{GC}(\theta)\rangle\ \text{caml\_plus1}\ [w]\\ &\langle w'.\ \exists m\, v'.\ m = n + 1 * m \sim_{ML} v' * v' \sim_C^{\theta} w' * \mathsf{GC}(\theta)\rangle\end{aligned}$$

Compare this to the specification we assumed in λ_{ML}:

$$\Psi_{\mathsf{plus1ML}} \triangleq \forall n.\ \langle \top \rangle\ \text{caml\_plus1}\ [n]\ \langle m.\ m = n + 1\rangle$$

We have now reached the point where we can conclude that the λ_C specification Ψ_{plus1} is indeed a faithful translation of the λ_{ML} specification Ψ_{plus1}. In fact, the specification above is already too strong: It also establishes that the garbage collector never runs. In general, it is sufficient to just establish $\exists \theta'.\ \mathsf{GC}(\theta')$, which denotes that the garbage collector may run, changing θ. For verifying the remaining examples, we just assert that for correctness, it suffices to prove the λ_C functions correct according to a specification that is similar to the λ_{ML} specification in the way shown above. For reference, the correctness proof of both programs, in the form of a Hoare outline, can be found in Figure 6.4. To make these outlines more compact, we follow the Iris Proof Mode [21] in distinguishing between spatial and persistent resources. We treat persistent resources as if they were pure, since both of these remain during the entire execution of the program. Specifically, instead of repeating these persistent resources each time, we "remember" them by writing them to the right of a spacial context. We only write them down once, understanding that such propositions are persistent, and then reference them later without being required to add them to our "spacial context" again. Non-persistent resources remain unchanged, we repeat these every time to indicate that they are still there.

Working With Blocks

The program we just verified is unfortunately not very exciting. It only manipulates a few numbers, and does not use most of the features provided by our wrapper. We now discuss more examples, in order to introduce the program logic rules for the remaining primitives. To do so, we verify that the following C program, caml_swap_pair, swaps the two components of a pair. Formally, we want it to satisfy the following specification:

$$\Psi_{\mathsf{swap\_pairML}} \triangleq \forall V_1\, V_2.\ \langle \top \rangle\ \text{caml\_swap\_pair}\ [\langle V_1, V_2 \rangle]\ \langle V'.\ V' = \langle V_2, V_1 \rangle\rangle$$

```
{$n \sim_{ML} v * \mathsf{GC}(\theta)$}                                    $v \sim_C^{\theta} \mathrm{v}$
{$v = n * \mathsf{GC}(\theta)$}                                         $n \sim_C^{\theta} \mathrm{v}$
   let $n$ = Int_val($v$) in
{$\mathsf{GC}(\theta)$}                                                 $n = n$
   let $m$ = $n + 1$ in
{$\mathsf{GC}(\theta)$}                                                 $m = n + 1$
   Val_int($m$)
{$w'. (n+1) \sim_C^{\theta'} w' * \mathsf{GC}(\theta)$}
{$w'. (n+1) \sim_C^{\theta'} w' * (n+1) \sim_{ML} (n+1) * \mathsf{GC}(\theta)$}
{$w'. \exists v'\, v' \sim_C^{\theta'} w' * (n+1) \sim_{ML} v' * \mathsf{GC}(\theta)$}
```

```
{$\top$}
   let $n$ = call caml_plus1 [41] in
{$n = 41 + 1$}
{$n = 42$}
   assert($n = 42$)
{$\_. \top$}
```

Figure 6.4: Hoare outlines for caml_plus1, and for its λ_{ML} client.

$$\Psi_{\text{swap\_pairC}} \triangleq \forall V_1\ V_2\ v\ w\ \theta.\ \langle\langle V_1, V_2\rangle \sim_{\mathrm{ML}} v * v \sim_{\mathrm{C}}^{\theta} w * \mathsf{GC}(\theta)\rangle\ \text{caml\_swap\_pair}\ [w]$$
$$\langle w'.\ \exists v'\ \theta'.\ \langle V_2, V_1\rangle \sim_{\mathrm{ML}} v' * v' \sim_{\mathrm{C}}^{\theta'} w' * \mathsf{GC}(\theta')\rangle$$

```
caml_swap_pair(v) ≜
  let r = malloc(1) in
  *r ← v;
  registerroot(r);
  let np = alloc(2, 0) in
  let v = *r in
  let cl = Field(v, 0) in
  let cr = Field(v, 1) in
  Store_field(np, 1, cl);
  Store_field(np, 0, cr);
  unregisterroot(r);
  free(r, 1);
  np
```

Figure 6.5: Specification for, and implementation of caml_swap_pair.

As defined there, *swapping* a pair means creating a new pair that has both components swapped. Also note that this function takes one argument, which is a pair, and not of two arguments. A simple λ_{ML} implementation of this function is $\mathsf{rec}\_\,x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x\rangle$. This λ_{ML} implementation already shows us the different tasks we need to accomplish in our λ_{C} program: reading the components of a pair, creating a new pair, and ensuring the new one stores the right components. Additionally, the λ_{C} program needs to live in harmony with the garbage collector, which is automatic in λ_{ML}. The λ_{C} implementation of this function closely follows the `swap_pair` program seen in Section 3.3. It is shown in Figure 6.5, along with the desired specification. A full Hoare outline of the correctness proof is shown in Figure 6.6.

To verify this program, we start by noticing that we receive a block-level value v that represents a λ_{ML} pair: $\langle V_1, V_2\rangle \sim_{\mathrm{ML}} v$. Due to SimPair, we know that $v = \gamma$, that $\gamma \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v_1, v_2]$, and that v_1 and v_2 represent V_1 and V_2. What is new here is the block-level points-to $\gamma \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v_1, v_2]$. This block-level points-to is not unlike the points-tos we have seen so far, except that it describes the contents of the block-level heap ζ. This specific points-to is just a specific version of the general block-level points-to $\gamma \mapsto_{\mathrm{FFI}}^{d} \mathsf{blk}$, which simply takes a block as argument.

$$\begin{array}{ll}
\{\langle V_1, V_2 \rangle \sim_{\mathrm{ML}} v_{\mathsf{pair}} * \mathsf{GC}(\theta)\} & v_{\mathsf{pair}} \sim_{\mathrm{C}}^{\theta} \mathrm{v} \\
\{v_{\mathsf{pair}} = \gamma_{\mathsf{pair}} * \gamma_{\mathsf{pair}} \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v_1, v_2] * & \gamma_{\mathsf{pair}} \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v_1, v_2] \\
\quad V_1 \sim_{\mathrm{ML}} v_1 * V_2 \sim_{\mathrm{ML}} v_2 * \mathsf{GC}(\theta)\} & V_1 \sim_{\mathrm{ML}} v_1 * V_2 \sim_{\mathrm{ML}} v_2 \\
\{\mathsf{GC}(\theta)\} & \gamma_{\mathsf{pair}} \sim_{\mathrm{C}}^{\theta} \mathrm{v} \\
\quad \mathsf{let}\ \mathrm{r} = \mathsf{malloc}(1)\ \mathsf{in} & \\
\{\mathsf{GC}(\theta) * \mathrm{r} \mapsto_{\mathrm{C}} \star\} & \\
\quad *\mathrm{r} \leftarrow \mathrm{v}; & \\
\{\mathsf{GC}(\theta) * \mathrm{r} \mapsto_{\mathrm{C}} \mathrm{v}\} & \\
\quad \mathsf{registerroot}(\mathrm{r}); & \\
\{\mathsf{GC}(\theta) * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}}\} & \\
\quad \mathsf{let}\ \mathrm{np} = \mathsf{alloc}(2, 0)\ \mathsf{in} & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [0, 0]\} & \gamma_{\mathrm{np}} \sim_{\mathrm{C}}^{\theta'} \mathrm{np} \\
\quad \mathsf{let}\ \mathrm{v} = *\mathrm{r}\ \mathsf{in} & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [0, 0]\} & \gamma_{\mathsf{pair}} \sim_{\mathrm{C}}^{\theta'} \mathrm{v} \\
\quad \mathsf{let}\ \mathrm{cl} = \mathsf{Field}(\mathrm{v}, 0)\ \mathsf{in} & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [0, 0]\} & v_1 \sim_{\mathrm{C}}^{\theta'} \mathrm{cl} \\
\quad \mathsf{let}\ \mathrm{cr} = \mathsf{Field}(\mathrm{v}, 1)\ \mathsf{in} & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [0, 0]\} & v_2 \sim_{\mathrm{C}}^{\theta'} \mathrm{cr} \\
\quad \mathsf{Store\_field}(\mathrm{np}, 1, \mathrm{cl}); & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [0, v_2]\} & \\
\quad \mathsf{Store\_field}(\mathrm{np}, 0, \mathrm{cr}); & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathsf{root}} \gamma_{\mathsf{pair}} * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [v_1, v_2]\} & \\
\quad \mathsf{unregisterroot}(\mathrm{r}); & \\
\{\mathsf{GC}(\theta') * \mathrm{r} \mapsto_{\mathrm{C}} w * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [v_1, v_2]\} & v_{\mathsf{pair}} \sim_{\mathrm{C}}^{\theta'} w \\
\quad \mathsf{free}(\mathrm{r}, 1); & \\
\{\mathsf{GC}(\theta') * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [v_1, v_2]\} & \\
\{\mathsf{GC}(\theta') * \gamma_{\mathrm{np}} \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [v_1, v_2] * V_1 \sim_{\mathrm{ML}} v_1 * V_2 \sim_{\mathrm{ML}} v_2\} & \\
\{\mathsf{GC}(\theta') * \langle V_2, V_1 \rangle \sim_{\mathrm{ML}} \gamma_{\mathrm{np}}\} & \\
\quad \mathrm{np} & \\
\{\mathrm{np}.\ \exists v_{\mathrm{np}}\ \theta'.\ \mathsf{GC}(\theta') * \langle V_2, V_1 \rangle \sim_{\mathrm{ML}} \gamma_{\mathrm{np}} * \gamma_{\mathrm{np}} \sim_{\mathrm{C}}^{\theta'} \mathrm{np}\} &
\end{array}$$

Figure 6.6: Hoare outline for caml_swap_pair.

$$
\begin{aligned}
\Psi_{\mathsf{read\_tag}} &\triangleq \forall w\,\gamma\,\theta\,\mathsf{blk}\,\mathrm{d}.\ \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{blk}\rangle\ \mathsf{read\_tag}\ [w] \\
&\qquad\qquad\qquad\quad \langle \mathrm{n}.\ \mathsf{tagNumber}(\mathsf{blk}) = \mathrm{n} * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{blk}\rangle \\
\Psi_{\mathsf{length}} &\triangleq \forall w\,\gamma\,\theta\,\mathrm{t}\,\mathrm{m}\,\vec{v}\,\mathrm{d}.\ \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{B}(\mathrm{m},\mathrm{t},\vec{v})\rangle\ \mathsf{length}\ [w] \\
&\qquad\qquad\qquad\qquad \langle \mathrm{n}.\ |\vec{v}| = \mathrm{n} * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{B}(\mathrm{m},\mathrm{t},\vec{v})\rangle \\
\Psi_{\mathsf{Field}} &\triangleq \forall w\,\gamma\,\theta\,\mathrm{t}\,\mathrm{m}\,\vec{v}\,\mathrm{d}\,\mathrm{n}. \\
&\quad \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{B}(\mathrm{m},\mathrm{t},\vec{v}) * 0 \leqslant \mathrm{n} < |\vec{v}|\rangle \\
&\quad \mathsf{Field}\ [w, \mathrm{n}] \\
&\quad \langle w'.\ v_{\mathrm{n}} \sim^{\theta}_{\mathsf{C}} w' * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathrm{FFI}} \mathsf{B}(\mathrm{m},\mathrm{t},\vec{v})\rangle \\
\Psi_{\mathsf{Store\_field}} &\triangleq \forall w\,\gamma\,\theta\,\mathrm{t}\,\vec{v}\,\mathrm{n}\,w'\,v'. \\
&\quad \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{FFI}} \mathsf{B}(\mathsf{Mut},\mathrm{t},\vec{v}) * 0 \leqslant \mathrm{n} < |\vec{v}| * v' \sim^{\theta}_{\mathsf{C}} w'\rangle \\
&\quad \mathsf{Store\_field}\ [w, \mathrm{n}, w'] \\
&\quad \langle 0.\ \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{FFI}} \mathsf{B}(\mathsf{Mut},\mathrm{t},\vec{v}[\mathrm{n} := v'])\rangle \\
\Psi_{\mathsf{alloc}} &\triangleq \forall \mathrm{n}_{\mathrm{t}}\,\mathrm{n}_{\mathsf{sz}}\,\mathrm{t}. \\
&\quad \langle \mathsf{GC}(\theta) * \mathrm{n}_{\mathrm{t}} = \mathrm{t} * \mathrm{t} \in \{0, 1\} * \mathrm{n}_{\mathsf{sz}} \geqslant 0\rangle \\
&\quad \mathsf{alloc}\ [\mathrm{n}_{\mathsf{sz}}, \mathrm{n}_{\mathrm{t}}] \\
&\quad \langle w.\ \exists \gamma\,\theta'.\ \gamma \sim^{\theta'}_{\mathsf{C}} w * \mathsf{GC}(\theta') * \gamma \mapsto_{\mathsf{blk}[\mathrm{t}|\mathsf{fresh}]} \underbrace{[0, \ldots, 0]}_{\mathrm{n}_{\mathsf{sz}}\text{ many}}\rangle
\end{aligned}
$$

Figure 6.7: Specifications for primitives operating on regular blocks.

In particular, the specific points-to is already specialized to be persistent. This is because pairs are represented by immutable blocks, and we know these will never change. Additionally, blocks from the block-level heap ζ are never deleted,[2] so they can remain forever. The garbage collector does not modify the block-level heap ζ directly, it only makes certain blocks inaccessible from C.

These block-level points-tos are then used to give specifications to the primitives operating on blocks, which can be found in Figure 6.7. The specification of alloc is the most interesting of these. What makes it more interesting is that alloc causes a garbage collector run. Thus, afterwards, the GC state θ has changed into θ', which reflects in the changed GC token $\mathsf{GC}(\theta')$. Also, we get a new block-level location γ. Both it and θ' are chosen demonically, which just means that our program has to be correct for all suitable choices. Finally, this primitive is specified using the *fresh block-level points-to*. We see why this is called "fresh" shortly. It additionally includes the token $\mathsf{isPriv}(\gamma)$. This token denotes that in the current visibility map χ, the location

[2] Again, ignoring that half that heap is deleted when switching the state to the OCaml side.

γ is still private. Similarly, there are $\mathsf{isLoc}(\ell, \gamma)$, for when γ represents the location γ, and $\mathsf{isForeign}(\iota, \gamma)$ for foreign blocks. In fact, these are best understood as *visibility map points-tos*, since they describe the data currently stored in the visibility map at these addresses. We also come back to these two other visibility map points-to variants shortly.

Roots

Instead, notice that, since alloc causes a new allocation, we had to root the input value r representing the input pair, so that we can still use it once we have allocated the new block. First, we think about what would have happened had we not rooted this value. Then, when trying to use it as the argument of Field, we would need to pass in a λ_C value w such that $w \sim_C^{\theta'} \gamma_r$, where γ_r be the block-level location initially encoded by r. In particular, the encoding relation $\sim_C^{\theta'}$ is parameterized by θ' (notice the prime!), since after allocation, we now have $\mathsf{GC}(\theta')$. However, all we know about γ_r is that $w \sim_C^{\theta} \gamma_r$ in the old GC state, since we got this initially, as part of the fact that our input value represents an λ_{ML} pair. This is now useless, since the current GC state is $\theta' \neq \theta$. So, if we had not used roots, we would be stuck here. Fortunately, we were wise enough to register our input value as a root. Note that to do so, we must allocate a λ_C memory cell, since roots are registered by-reference, and local variables are not part of the formal memory. To now actually reason about roots, we introduce the *root points-to*. The root points-to $a \mapsto_{\mathsf{root}} v$ indicates that the location at a is a root, and that this root stores the block-level value v. The specification for $\mathsf{registerroot}$ and $\mathsf{unregisterroot}$, already introduced in Figure 6.2, now amount to swapping between C and root points-tos. Intuitively, we give up the ownership over a memory location when registering a root, so that the location is now owned by the runtime. Indeed, every time the garbage collector runs, the roots are modified by the runtime, which requires this (full) ownership. The root points-to can then be thought of as "quasi-ownership," which tells us that we can get the ownership back when we unregister a root. But notice how caml_swap_pair actually modifies the content of roots: It uses ordinary λ_C loads and stores! The fact that regular C memory accesses still work at rooted locations is formalized by CWP-Load-Root and CWP-Store-Root. We note that these are rules of the λ_C program logic, that are applicable when just reasoning about λ_C. This is possible because, as we see in Section 6.3, the GC token $\mathsf{GC}(\theta)$ stores the C points-to that is consumed by $\mathsf{registerroot}$, so that it can temporarily be used to justify memory accesses, like those indicated by these rules.

We also note that the rules for $\mathsf{registerroot}$ are stricter than what would be possible with our operational semantics. Specifically, the operational semantics require that the roots encode block-level values *only* when a GC run happens. Our program logic, instead, requires that they store such values *all the time*, in particular already when the root is registered using $\mathsf{registerroot}$. By outright banning such a temporary breakage of encoding invariants, we make the program logic easier. Now, it becomes unnecessary to track which roots are currently not storing valid encodings of block-level values.

Freezing Block-Level Points-Tos

Armed with our understanding of block-level points-tos, we can reason through the uses of Field and Store_field. (We encourage the reader to follow along through the Hoare outline in Figure 6.6.) These are very similar to standard load and store rules. The only difference between these and other classical memory rules (*e.g.*, CWP-Store) are that here, both the block-level location, and the block-level values that are read/written, are passed as λ_C values encoding these block-level values. We are also able to reason through how the root is registered, used, and later deallocated. Once we are done with this, our program has terminated. There, we have the following separation logic resources at hand (including some persistent resources):

$$\mathsf{GC}(\theta') * \gamma_{\text{np}} \mapsto_{\mathsf{blk}[0|\mathsf{fresh}]} [v_2, v_1] * V_1 \sim_{\text{ML}} v_1 * V_2 \sim_{\text{ML}} v_2$$

To establish that we indeed return a pair, we need a value v such that $\langle V_2, V_1 \rangle \sim_{\text{ML}} v$. By SimPair, this value needs to point to a block, tag 0, with the components of the pair. Our newly allocation location γ_{np} indeed does so. However, one thing is wrong: The mutability. Pairs are backed by *immutable* points-tos, while our points-to is still mutable and fully owned. It had to be mutable, so that we could initialize it. But now that we are done with the initialization, we want to mark it as immutable. For this, we can use the law UpdateFreeze, shown in Figure 6.8. This law is a so-called *update law*: It allows us to perform a ghost update, by which we can turn our fresh points-to into an immutable one. Additionally, the GC token must be present during the update, since it gates the access to runtime memory in general. In the operational semantics, this freezing step is justified by FreezeMut. But there, this freezing step can only be performed when the state actually switches back to OCaml. In the program logic, we are able to do these freezing operations *in advance*–but this makes the implementation (discussed in Section 6.3) more complicated.

This freezing operation is why we have a special *fresh* block-level points-to. A regular block usually has one of two roles: It either backs some immutable data, like a pair, or it backs a location, which requires it to be mutable (as we see shortly). When a block is associated with a location in the visibility map χ, it remains so forever. This means that it can not be made immutable again, since this would violated IsStoreBlocks. A fresh block-level points-to denotes that this points is *not* (yet) associated with a location. It can thus be made mutable without breaking any invariants of the wrapper. We say that such a points-to has not yet been *shared* with OCaml, since it was not yet made accessible from λ_{ML} code. Formally, this is guaranteed by including $\mathsf{isPriv}(\gamma)$ in the definition of a fresh points-to, which excludes the reference from being associated with a location. Further note that the update laws shown in Figure 6.8 are usable when just reasoning in the λ_C program

$$\textsc{UpdateFreeze}\quad \dfrac{\mathsf{GC}(\theta) \qquad \gamma \mapsto_{\mathsf{blk}[t|\mathsf{fresh}]} \vec{v}}{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathsf{blk}[t|\mathsf{imm}]} \vec{v}} \qquad\qquad \textsc{UpdateExpose}\quad \dfrac{\mathsf{GC}(\theta) \qquad \gamma \mapsto_{\mathsf{blk}[t|\mathsf{fresh}]} \vec{v}}{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathsf{blk}[t|\mathsf{mut}]} \vec{v}}$$

$$\textsc{UpdateMlToBlock}\quad \dfrac{\mathsf{GC}(\theta) \qquad \ell \mapsto_{\mathrm{ML}} \vec{V}}{\exists \gamma\, \vec{v}.\, \mathsf{GC}(\theta) * \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}}$$

$$\textsc{UpdateBlockToMl}\quad \dfrac{\mathsf{GC}(\theta) \qquad \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v}}{\exists \ell.\, \mathsf{GC}(\theta) * \ell \mapsto_{\mathrm{ML}} \vec{V} * \mathsf{isLoc}(\ell, \gamma)}$$

$$\textsc{ConfrontMlBlock}\quad \dfrac{\mathsf{GC}(\theta) \qquad \mathsf{isLoc}(\ell, \gamma) \qquad \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \ell \mapsto_{\mathrm{ML}} \vec{V}}{\bot}$$

$$\textsc{IsLocInjective}\quad \dfrac{\mathsf{isLoc}(\ell_1, \gamma_1) \qquad \mathsf{isLoc}(\ell_2, \gamma_2) \qquad \mathsf{GC}(\theta)}{\ell_1 = \ell_2 \iff \gamma_1 = \gamma_2}$$

$$\textsc{IsFgnInjective}\quad \dfrac{\mathsf{isForeign}(\iota_1, \gamma_1) \qquad \mathsf{isForeign}(\iota_2, \gamma_2) \qquad \mathsf{GC}(\theta)}{\iota_1 = \iota_2 \iff \gamma_1 = \gamma_2}$$

Figure 6.8: Update laws for manipulating block-level points-tos.

$$\Psi_{\text{plus1\_refC}} \triangleq \forall n\, \ell\, v\, w\, \theta.\ \langle \ell \mapsto_{\text{ML}} [n] * \ell \sim_{\text{ML}} v * v \sim_{\text{C}}^{\theta} w * \mathsf{GC}(\theta) \rangle\ \text{caml\_plus1\_ref}\ [w]$$
$$\langle w'.\ \exists \theta'.\ \ell \mapsto_{\text{ML}} [n+1] * \langle\rangle \sim_{\text{ML}} v' * v' \sim_{\text{C}}^{\theta'} w' * \mathsf{GC}(\theta') \rangle$$
$$\Psi_{\text{plus1\_refML}} \triangleq \forall n\, \ell.\ \langle \ell \mapsto_{\text{ML}} [n] \rangle\ \text{caml\_plus1\_ref}\ [\ell]$$
$$\langle \langle\rangle.\ \ell \mapsto_{\text{ML}} [n+1] \rangle$$

```
caml_plus1_ref(v) ≜
  let nc  =  Field(v, 0) in
  let n  =  Int_val(nc) in
  let m  =  n + 1 in
  let mc  =  Val_int(m) in
  Store_field(v, 0, mc);
  Val_int(0)
```

Figure 6.9: Specification for, and implementation of caml_plus1_ref.

logic. They do not need access to the wrapper state interpretation. Instead, the GC token $\mathsf{GC}(\theta)$ actually contains the invariants and separation logic resources required to allow this update. This fact is again part of what we understand by language-local reasoning: λ_{C} code, even if it uses FFI primitives, should remain verifiable within the existing λ_{C} program logic. By performing this update, we can then establish that $\gamma_{\text{np}} \mapsto_{\text{blk}[0|\text{imm}]} [v_2, v_1]$, which allows us to conclude that $\langle V_2, V_1 \rangle \sim_{\text{ML}} \gamma_{\text{np}}$. Once this is done, the remaining obligations are almost trivial.

6.2.2 Working With References

We next look at a program that works with a λ_{ML} reference. Specifically, we generalize our caml_plus1 program to take a reference to an integer, and to then increment this reference. Again, this is just a formalization of a role model program we saw in Section 3.1. The function caml_plus1_ref, along with its desired λ_{ML} and derived λ_{C} specification, are shown in Figure 6.9.

The middle part of this program just manipulates some block-level points-tos using the primitives we already discussed. But before we can start so, we notice a problem. From the precondition, we get that the λ_{ML} points-to $\ell \mapsto_{\text{ML}} [n]$, which tells us that ℓ points to a one-entry array (a reference). But to use Field and Store_field, we need a *block-level* points-to. How do we get such a points-to? Intuitively, we know that an λ_{ML} array is represented as a block. We just need to formalize this. We do so by adding more update laws, namely the laws UpdateMlToBlock and UpdateBlockToMl from Figure 6.8. The first law, UpdateMlToBlock, allows us to turn an λ_{ML} points-to into

$$
\begin{array}{ll}
\{\ell \mapsto_{\mathrm{ML}} [n] * \ell \sim_{\mathrm{ML}} v_{\mathsf{loc}} * \mathsf{GC}(\theta)\} & v_{\mathsf{loc}} \sim_{\mathrm{C}}^{\theta} v \\
\{\ell \mapsto_{\mathrm{ML}} [n] * v_{\mathsf{loc}} = \gamma_{\mathsf{loc}} * \mathsf{GC}(\theta)\} & v_{\mathsf{loc}} \sim_{\mathrm{C}}^{\theta} v * \ell \sim_{\mathrm{ML}} \gamma_{\mathsf{loc}} \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [V] * \mathsf{GC}(\theta)\} & V \sim_{\mathrm{ML}} n \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \mathsf{GC}(\theta)\} & \\
\quad \mathsf{let}\ nc\ =\ \mathsf{Field}(v, 0)\ \mathsf{in} & \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \mathsf{GC}(\theta)\} & n \sim_{\mathrm{C}}^{\theta} nc \\
\quad \mathsf{let}\ n\ =\ \mathsf{Int\_val}(nc)\ \mathsf{in} & \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \mathsf{GC}(\theta)\} & n = n \\
\quad \mathsf{let}\ m\ =\ n + 1\ \mathsf{in} & \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \mathsf{GC}(\theta)\} & m = n + 1 \\
\quad \mathsf{let}\ mc\ =\ \mathsf{Val\_int}(m)\ \mathsf{in} & \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \mathsf{GC}(\theta)\} & (n+1) \sim_{\mathrm{C}}^{\theta} nc \\
\quad \mathsf{Store\_field}(v, 0, mc); & \\
\{\gamma_{\mathsf{loc}} \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n+1] * \mathsf{GC}(\theta)\} & (n+1) \sim_{\mathrm{C}}^{\theta} nc \\
\{\ell' \mapsto_{\mathrm{ML}} [n+1] * \mathsf{GC}(\theta)\} & \ell' \sim_{\mathrm{ML}} \gamma_{\mathsf{loc}} \\
\{\ell \mapsto_{\mathrm{ML}} [n+1] * \mathsf{GC}(\theta)\} & \\
\quad \mathsf{Val\_int}(0) & \\
\{w.\ \ell \mapsto_{\mathrm{ML}} [n+1] * \langle\rangle \sim_{\mathrm{ML}} 0 * 0 \sim_{\mathrm{C}}^{\theta} w * \mathsf{GC}(\theta)\} & \\
\{w.\ \exists v.\ \theta'.\ \ell \mapsto_{\mathrm{ML}} [n+1] * \langle\rangle \sim_{\mathrm{ML}} v * v \sim_{\mathrm{C}}^{\theta'} w * \mathsf{GC}(\theta')\} &
\end{array}
$$

Figure 6.10: Hoare outline for caml_plus1_ref.

a block-level points-to. The contents of the new block-level points-to must represent the contents of the original block. We also get the fact that the new block-level location γ represents the original λ_{ML} location ℓ (compare with SimLoc). Later in the proof, we need to convert back. For this, we use the rule UpdateBlockToMl. This rule is a bit more powerful than we need it here: It allows any mutable block-level points-to to be turned into a λ_{ML}, even if this block was originally allocated in glue code. We discuss this further down. In our proof in Figure 6.10, we just use this law to update the block-level points-to back to an λ_{ML} one. If we inspect the rule carefully, we see that we get an λ_{ML} points-to for an existentially quantified location ℓ', which may or may not be the original location ℓ. To prove that it is the original location, we use the fact that no two λ_{ML} locations are represented by the same block-level location. Specifically, we use IsLocInjective on $\ell \sim_{\mathrm{ML}} \gamma_{\mathsf{loc}}$ and $\ell' \sim_{\mathrm{ML}} \gamma_{\mathsf{loc}}$, which unfold (using SimLoc) to isLoc. This allows us to conclude that $\ell = \ell'$. Apart from this, the correctness proof of caml_plus1_ref is standard. We again encourage the reader to read through the Hoare outline of Figure 6.10.

View Reconciliation The two rules, UpdateMlToBlock and its inverse UpdateBlockToMl, might seem obvious. They do however represent a not entirely obvious solution to a potentially serious problem. To understand this problem, remember that when switching to the C side, the entire λ_{ML} heap σ_{ML} is serialized into the block-level heap ζ. While on the C side, there is no physical λ_{ML} state. It is only restored when the state switches back to the OCaml side. One potential solution would be to just convert all λ_{ML} points-tos to block-level points-tos when switching the state to C, and to convert them all back when switching back. This solution has but one problem: It *breaks the frame rule*. To see why, consider how this would work as a user of this logic. To prove that an external call is correct, the user would not only need to fulfill the specification of this call, but also give up *all* λ_{ML} points-tos. This includes points-tos that are currently framed out, which means that framing points-tos around external calls would need to be forbidden.

Instead, our solution works without breaking any laws of separation logic: We allow these points-tos to be converted gradually and locally. Locally means that each points-to can be converted on its own, independent of the other points-tos. Gradually means that this conversion can happen at any time (while in glue code), instead of all-at-once. In fact, gradual conversions entails that a λ_{ML} points-to just remains an λ_{ML} points-to throughout the glue code. This is the case for most λ_{ML} points-tos, in particular for all those that are currently framed out. Currently, the rules are such that we can only ever convert fully owned points-tos. This ensures that both points-tos are exclusive with each other, *i.e.*, we can never have both around at the same time. This is formalized by the rule ConfrontMlBlock. In Section 6.4, we strengthen our theory to allow having fractional points-tos of both around, as long as the combined sum of these fractions across both kinds is $\leqslant 1$.

As a final point, the rule UpdateBlockToMl even allows setting the new λ_{ML} values in the array. This means that going from λ_{ML} to λ_{C} and back can change a points-to from one storing booleans (represented by 0 or 1) to one storing integers (still represented by 0 or 1), even if not accessed by the C code. In OCaml, there even is a function `Obj.magic : 'a -> 'b`, which unsafely transmutes any value into any value, that is just implemented[3] as the identity function. This function is an example of unsafe transmutes (allowing one to write code violating type safety), and our logic is able reason about such unsafe transmutes, so that we can show that some of these are correct under the right conditions. But note that our model of the OCaml runtime is not perfect, and there might be compiler invariants about such unsafe transmutes that we are not aware of.

Before we continue, we want to look at how these rules can inform our intuitive mental model of reasoning about glue code. In Ocaml-C glue code, we have to think about OCaml values in both their high-level, OCaml representation, and their block-level representation. In fact, we often have to switch between both of them. Since the underlying state does not change, this change is just a change of view. Intuitively, we mentally "flip a switch," to now think about a particular block as storing block-level instead of λ_{ML} values. These update rules tell us what conditions we need to check when switching back and forth. We therefore call these rules *view reconciliation* rules, since they allow us to reconcile the OCaml view with the block-level view. We also speak of exchanging, or of trading in points-tos.

Creating New References We mentioned earlier that we can create a new λ_{ML} reference in glue code, by simply allocating a block using alloc. This block is fresh, and in order to back a reference, it needs to be turned into a mutable block-level points-to $\gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v}$. This is possible using yet another update law, namely UpdateExpose. Like all other update laws, this law is also a fact that already holds in the λ_{C} program logic, since the GC token $\mathsf{GC}(\theta)$ is able to implement this switching. Note that a mutable points-to already is associated with a location ℓ, by definition. Thus, UpdateExpose already *exposes* this location to λ_{ML}, by converting from $\mathsf{isPriv}(\gamma)$ to $\mathsf{isLoc}(\ell, \gamma)$. In the operational semantics, this update step (compare ExposePublic) only happens during state switches, but we again strengthen the program logic by allowing these switches to happen *in advance*. This is similar to how we were able to make a block immutable *in advance* with the rule UpdateFreeze. We also would like to remark that when turning a block-level points-to into an λ_{ML} points-to using UpdateBlockToMl, the choice of the λ_{ML} values $\vec{v}$ that end up stored in that block is made by the user (who verifies the program). In particular, if we have $\gamma \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [0]$, this could be turned into a λ_{ML} points-to that stores $[0]$, or one that

[3]Actually, the OCaml compiler sometimes performs optimizations with the knowledge that this implemented by the identify function. Instead, consider this paragraph to instead apply to this function wrapped inside the opaque identity: `fun x => Sys.opaque_identity(Obj.magic x)`

$$
\begin{aligned}
\Psi^{\Psi}_{\mathsf{callback}} \triangleq\ & \forall f\, x\, e\, \gamma\, V\, v\, \theta\, Q. \\
& \langle \gamma \sim^{\theta}_{C} w * \gamma \mapsto_{\mathsf{clos}} \mathsf{rec}\, f\, x.\, e * v \sim^{\theta}_{C} w' * V \sim_{\mathrm{ML}} v * \mathsf{GC}(\theta) * \\
& \quad \triangleright \mathsf{wp}\ (\mathsf{rec}\, f\, x.\, e) V \mathbin{@} \emptyset, \Psi\, \{Q\} \rangle \\
& \mathsf{callback}\ [w, w'] \\
& \langle w_r.\, \exists V_r\, v_r\, \theta'.\, \mathsf{GC}(\theta') * V_r \sim_{\mathrm{ML}} v_r * v_r \sim^{\theta'}_{C} w_r * Q(V_r) \rangle
\end{aligned}
$$

Figure 6.11: The specification for callback.

stores [true], or one that stores [⟨⟩]. This crucially relies on the fact that state switching is *angelic*, so that during the correctness proof, we can choose which value should be used here.

The Other Primitives We have not yet discussed the specifications for read_tag, length, and isblock. The one for isblock uses IsBlock, defined in Figure 6.2, to determine whether 0 or 1 should be returned. The other two, which return the tag number of the length of a block, are very simple. We remark that length only works on regular blocks, whereas read_tag supports all kinds of blocks, so that it can be used to discriminate between different kinds of blocks, *e.g.*, between callbacks and foreign blocks.

6.2.3 Callbacks

For an interesting but simple example using callbacks, we can consider a λ_C function that, when passed a closure, just invokes this closure with argument ⟨⟩. We then prove that this can be used to build a diverging program. But in order to diverge, this program will repeatedly switch between OCaml and C. This also then demonstrates that such recursive method calls can be nested arbitrarily deep in our model. The specification for callback is shown in Figure 6.11. Besides the usual encoding of arguments as λ_C values, this specification tells us that invoking a closure simply executes the λ_{ML} code that is usually executed when invoking a closure. This is accomplished by including the normal λ_{ML} weakest precondition in the specification. The program that uses callbacks to diverge consists of two parts: A λ_{ML} part, and a λ_C part. These are as follows, with the λ_{ML} part on the left:

```
e_diverge ≜                                          caml_diverge(F) ≜
  let F = (rec F x. call caml_diverge F) in            let u = Val_int(0) in
  F(⟨⟩)                                                callback(F, u)
```

The main challenge when verifying this program is verifying that the closure defined inside e_{diverge}, when applied to ⟨⟩, does indeed diverge. To prove this, we use the following specification for the external call to caml_diverge:

$\Psi^{\Psi}_{\text{diverge}} \triangleq \forall \mathsf{f}\ \mathsf{x}\ e\ Q.\ \langle \triangleright\, \mathsf{wp}\ (\mathsf{rec}\ \mathsf{f}\ \mathsf{x}.\ e)(\langle\rangle)\ @\ \varnothing, \Psi\ \{Q\}\rangle\ \text{caml\_diverge}\ [\mathsf{rec}\ \mathsf{f}\ \mathsf{x}.\ e]\ \langle V'.\ Q(V')\rangle$

This specification just expresses that caml_diverge emulates the behavior of its input argument applied to $\langle\rangle$.

Both $\Psi^{\Psi}_{\text{diverge}}$ and $\Psi^{\Psi}_{\text{callback}}$ are parameterized by another λ_{ML} protocol Ψ, which describes the external calls available to callbacks made from C. The reason this is necessary is intricate: Basically, we have two layers of external calls. On the first layer, we have the external calls made by λ_{ML}, that follow the specification $\Psi^{\Psi}_{\text{diverge}}$. Since these external calls use callbacks, execution switches back to the OCaml side at the next higher level. Here, the external calls that are specified to be available are Ψ. In other words, each time we add a new frame to the cross-language call stack (without returning in-between) by invoking a callback, we unwrap this specification once. This is insufficient to prove correctness of our diverging program that switches back-and-forth between λ_{ML} and λ_{C}. In order to verify it, we want to have the same specification on all layers. Formally, we want our "infinitely unfolding" specification $\Psi_{\text{div\_fix}}$ to be a fixed point of the above specification:

$$\Psi_{\text{div\_fix}} = \Psi^{\Psi_{\text{div\_fix}}}_{\text{diverge}}$$

Luckily, we can accomplish this. To do so, we use the magic of step-indexing. For this, it is crucial that the specification Ψ_{diverge} guards the weakest precondition part behind a later. This ensures that $\Psi \mapsto \Psi^{\Psi}_{\text{diverge}}$ is *contractive*. We can thus tie a fixed point specification that satisfies the above law. In order to show that caml_diverge, written in λ_{C}, actually implements this specification, it is necessary that the specification of callbacks also guards its use of the weakest precondition behind a later. Once we have shown that the λ_{C} function satisfies the overall specification, verifying that the closure diverges becomes a case study in *Löb induction*. By Löb induction, we can assume that we have already proven that our closure diverges *at the next step-index*. Since the usage of this closure (inside the weakest precondition) is guarded by a later, everything works out.

6.2.4 Foreign Blocks and Semantic Types

The rules for primitives manipulating custom blocks are shown in Figure 6.12. By now, the rules should not be surprising. They involve custom block points-tos, which are also yet another specification of the block-level points-tos we have seen. Specifically, they carry with them the fact that the custom block backs a certain foreign value identifier, indicated by $\text{isForeign}(\iota, \gamma)$. This proposition is persistent, and similar to $\text{isLoc}(\ell, \gamma)$, it also has an injectivity property, as defined by IsFgnInjective.

In Section 3.4, we created a simple buffer library, that uses custom blocks to implement a very basic form of buffers. These buffers were so simple, that all they stored

$$\begin{aligned}
\Psi_{\mathsf{read\_custom}} &\triangleq \forall w\,\gamma\,\theta\,w'\,\mathrm{d}.\ \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathsf{cstm}} w' \rangle\ \mathsf{read\_custom}\ [w] \\
&\qquad\qquad \langle w'.\ \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto^{\mathrm{d}}_{\mathsf{cstm}} w' \rangle \\
\Psi_{\mathsf{write\_custom}} &\triangleq \forall w\,\gamma\,\theta\,w'.\ \langle \gamma \sim^{\theta}_{\mathsf{C}} w * \mathsf{GC}(\theta) * \gamma \mapsto_{\mathsf{cstm}} \_ \rangle\ \mathsf{write\_custom}\ [w, w'] \\
&\qquad\qquad \langle 0.\ \mathsf{GC}(\theta) * \gamma \mapsto_{\mathsf{cstm}} w' \rangle \\
\Psi_{\mathsf{alloc\_custom}} &\triangleq \langle \mathsf{GC}(\theta) \rangle\ \mathsf{alloc\_custom}\ [] \\
&\quad \langle w.\ \exists \gamma\,\theta'.\ \gamma \sim^{\theta'}_{\mathsf{C}} w * \mathsf{GC}(\theta') * \gamma \mapsto_{\mathsf{cstm}} \mathsf{None} \rangle
\end{aligned}$$

Figure 6.12: Specifications for primitives operating on custom blocks.

was one λ_C integer. To reason about these buffers in λ_{ML}, we would create a predicate $\mathrm{IsBuffer}(V, n)$, which describes that V is a buffer value, which stores the integer n. Then, a specification of a method that modifies the buffer might be given as follows:

$$\forall V\,n.\ \langle \mathrm{IsBuffer}(V, \_) \rangle\ \mathrm{caml\_buf\_store}\ [V, n]\ \langle 0.\ \mathrm{IsBuffer} \rangle (V, n)$$

If our implementation were to follow the example of Section 3.4, the definition of IsBuffer would have to be as follows:

$$\mathrm{IsBuffer}(V, n) \triangleq \exists \iota\,\gamma.\ V = \textcircled{\iota} * \mathsf{isForeign}(\iota, \gamma) * \gamma \mapsto_{\mathsf{cstm}} n$$

What is more interesting is that this can also be used with the logical relation of Section 4.5. As an example, we can encapsulate a closure that calls caml_buf_store behind an existential type, so that the following λ_{ML} expression can be shown semantically well-typed at type $\exists \tau.\ \tau \to \mathsf{int} \to \mathsf{unit}$:

$$\mathsf{pack}\ \lambda b\ n.\ \mathsf{call\ caml\_buf\_store}\ [b, n]$$

To do so, we need to pick a suitable type, with which we can instantiate the existential quantifier. This type is the following type, which is a persistent predicate on values:

$$\mathrm{BufTypeInterp}(V) \triangleq \exists \iota.\ V = \textcircled{\iota} * \boxed{\exists n.\ \mathrm{IsBuffer}(V, n)}^{\mathcal{N}_{\mathtt{buf}}.\iota}$$

By using the logical relation, we can not only prove that a certain function computes the proper values, but also that it can be safely encapsulated in λ_{ML} by giving it a λ_{ML} type, which might be existential. In real-world OCaml, which lacks first-order existential types, this would usually be accomplished by using a module. Such a module can simply be *defined* like this:

```
struct
    type buf
    external buf_store : buf -> int -> unit = "caml_buf_store"
    ...
end
```

Our formal model of OCaml is closer to SystemF, it in particular lacks OCaml's module system. While encoding OCaml's modules in SystemF is possible [39], we do not do so, since it is unrelated to the OCaml FFI.

6.2.5 Combining Verified Programs

For all the examples shown so far, we just stopped after proving that our λ_C programs satisfy specifications similar enough ot those assumed for the λ_{ML} program. Now, we discuss what is needed to close the gap towards a final adequacy theorem. For this, we go back to the initial example, caml_plus1. We proved the following C specification:

$$\begin{aligned}\Psi_{\mathsf{plus1}} \triangleq \forall n, v, w\, \theta.\ &\langle n \sim_{ML} v * v \sim_C^{\theta} w * \mathsf{GC}(\theta)\rangle\ \mathrm{caml\_plus1}\ [w] \\ &\langle w'.\ \exists m\, v'.\ m = n+1 * m \sim_{ML} v' * v' \sim_C^{\theta} w' * \mathsf{GC}(\theta)\rangle\end{aligned}$$

Compare this to the specification we assumed in λ_{ML}:

$$\Psi_{\mathsf{plus1}} \triangleq \forall n.\ \langle \top \rangle\ \mathrm{caml\_plus1}\ [n]\ \langle m.\ m = n+1 \rangle$$

We argued that these specifications are similar enough. In fact, we even argued that the original specification is stronger than required, since it also encodes that the garbage collector never runs. We now formally define the translation that tells us what specification, exactly, our λ_C program must occupy for us to use it from λ_{ML}. Formally, this translation is defined as a translation of protocols. Given an λ_{ML} protocol Ψ about functions with λ_{ML} values as arguments (and as return values), it creates a λ_C protocol $[\Psi]_{FFI}$ that defines the specification a λ_C program needs to have in order for us to claim that it implements that λ_{ML} specification. The formal definition of the translation $[\Psi]_{FFI}$ is shown in Figure 6.13. Intuitively, this protocol says that we get λ_C values that encode (via $\sim_{ML}$ and $\sim_C^{\theta}$) the original λ_{ML} values, and that similarly the λ_C return value must encode a λ_{ML} that satisfies the original postcondition. Further, we get to initially assume the GC token $\mathsf{GC}(\theta)$, which we have to return in the end. But we do not have to return $\mathsf{GC}(\theta)$ specifically–the address map may change, so that $\exists \theta'.\ \mathsf{GC}(\theta')$ is sufficient. As mentioned, our specification Ψ_{plus1} is thus even stronger than required. Formally, we have that Ψ_{plus1} is entailed by the translated specification: $[\Psi_{\mathsf{plus1}}]_{FFI} \sqsubseteq \Psi_{\mathsf{plus1}}$ (remember that the notion of

$$\begin{aligned}
[\Psi]_{\mathrm{FFI}}(\mathsf{fn}, \vec{w}, Q_{\mathrm{C}}) &\triangleq \exists \vec{V}\, \vec{v}\, \theta\, Q_{\mathrm{ML}}.\, \vec{V} \sim_{\mathrm{ML}} \vec{v} * \vec{v} \sim_{\mathrm{C}}^{\theta} \vec{w} * \mathsf{GC}(\theta) * \\
&\qquad \Psi(\mathsf{fn}, \vec{V}, Q_{\mathrm{ML}}) * \\
&\qquad \forall V'\, v'\, \theta'\, w'.\, V' \sim_{\mathrm{ML}} v' * v' \sim_{\mathrm{C}}^{\theta'} w' * \mathsf{GC}(\theta') \twoheadrightarrow\!\!* \\
&\qquad\quad Q_{\mathrm{ML}}(V') \twoheadrightarrow\!\!* Q_{\mathrm{C}}(w) \\
\Psi^{P,Q}_{\mathsf{main}_e}(\mathsf{fn}, \vec{w}, Q_{\mathrm{C}}) &\triangleq \langle P * \mathsf{atInit} \rangle\ \texttt{"main"}\ [] \\
&\qquad \langle w'.\, \exists \vec{V}\, \vec{v}\, \theta.\, w'.\, V' \sim_{\mathrm{ML}} v' * v' \sim_{\mathrm{C}}^{\theta'} w' * \mathsf{GC}(\theta') * Q(V') \rangle \\
\Psi^{\Psi}_{\mathsf{FFI}} &\triangleq \Psi_{\mathsf{Int\_val}} \sqcup \Psi_{\mathsf{Val\_int}} \sqcup \cdots \sqcup \Psi^{\Psi}_{\mathsf{callback}} \qquad \text{all except } \mathsf{main} \\
[e]_{\mathrm{FFI}} &\triangleq \{\texttt{"main"} := \mathsf{main}_e, \texttt{"Int\_val"} := \mathsf{Int\_val}, \ldots\}
\end{aligned}$$

$$\textsc{CombinedCorrect}\quad \frac{\{P\}\, e\ @\, \varnothing, \Psi\, \{V.\, Q(V)\}_{\mathrm{ML}} \qquad \Psi^{\Psi_{\mathrm{cb}}}_{\mathsf{FFI}} \vdash p : [\Psi]_{\mathrm{FFI}} \qquad \mathsf{dom}\ p \ \#\#\ \mathsf{dom}\ [e]_{\mathrm{FFI}}}{\perp \vdash [e]_{\mathrm{FFI}} \oplus p : \Psi^{P,Q}_{\mathsf{main}_e}}$$

Figure 6.13: Reasoning rules for combining OCaml and C.

strength is “flipped” for protocols, see Section 4.1). Note that if we define a program $p_{\mathsf{plus1}} = \{\texttt{"caml\_plus1"} := \mathsf{caml\_plus1}\}$, we can even establish a program triple:

$$\Psi^{\Psi}_{\mathsf{FFI}} \vdash p_{\mathsf{plus1}} : \Psi_{\mathsf{plus1}}$$

The λ_{ML} interface Ψ is the interface specifying the behavior of closures. Since our example never invokes closures, the interface we put there does not matter. To avoid cluttering the following presentation, we omit it there. As a reminder, we have also shown that our λ_{ML} expression always executes without failure:

$$\{\top\}\, e_{\mathsf{plus1}}\ @\, \varnothing, \Psi_{\mathsf{plus1}}\, \{V.\, \top\}_{\mathrm{ML}}$$

If we now embed e_{plus1} in our wrapper, and link this with the lifted C program, we gain a multi-language program. We can then use the rule CombinedCorrect to establish that this combined program is correct:

$$\dfrac{\{\top\}\, e_{\mathsf{plus1}}\ @\, \varnothing, \Psi_{\mathsf{plus1}}\, \{V.\, \top\}_{\mathrm{ML}} \qquad \dfrac{\Psi_{\mathsf{FFI}} \vdash p_{\mathsf{plus1}} : \Psi_{\mathsf{plus1}} \qquad [\Psi_{\mathsf{plus1}}]_{\mathrm{FFI}} \sqsubseteq \Psi_{\mathsf{plus1}}}{\Psi_{\mathsf{FFI}} \vdash p_{\mathsf{plus1}} : [\Psi_{\mathsf{plus1}}]_{\mathrm{FFI}}}\ \textsc{JudgmentWeaken}}{\perp \vdash [e_{\mathsf{plus1}}]_{\mathrm{FFI}} \oplus p_{\mathsf{plus1}} : \Psi^{\top,\top}_{\mathsf{main}_{e_{\mathsf{plus1}}}}}\ \textsc{CombinedCorrect}$$

The result of this rule, $\perp \vdash [e_{\mathsf{plus1}}]_{\mathrm{FFI}} \oplus p_{\mathsf{plus1}} : \Psi^{\top,\top}_{\mathsf{main}_{e_{\mathsf{plus1}}}}$, tells us that the overall program is correct. The program it applies to, $[e_{\mathsf{plus1}}]_{\mathrm{FFI}} \oplus p_{\mathsf{plus1}}$, is the result of linking. Specifically, we linked our C program p_{plus1}, which also is a program of $\uparrow\lambda_{\mathrm{C}}$, with the wrapper program $[e_{\mathsf{plus1}}]_{\mathrm{FFI}}$, which contains all the primitives, and specifically contains the main primitive set up such that it starts executing e_{plus1}.

The rule then tells us two things. First, this resulting linked program does not make any external calls. Second, this program satisfies the specification $\Psi^{\top,\top}_{\mathsf{main}_{e_{\mathsf{plus1}}}}$. This specification, defined in Figure 6.13, tells us that the main function can safely execute. It is parameterized by a precondition P, and a postcondition Q, the latter describing the returned λ_{ML} values. In our case, $P \triangleq \top$, since our program has no preconditions. We also set $Q(V) \triangleq \top$, since we do not care about the result value, all we are interested is proving our program is safe. This specification then tells us that invoking the function `"main"` without arguments is almost safe, since in particular $P = \top$. It also tells us that the return value in λ_{C} must encode an λ_{ML} value in the usual sense, but we ignore this for now. Only one precondition remains, namely atInit. This separation logic token ensures that main can only be invoked once, and that the state is still empty, as required by SWrapPrimMain. As it turns out, we get this token initially when starting in the empty state. Assuming that we have this atInit token initially, we can now prove that the weakest precondition of invoking `"main"` in this linked program:

$$\mathsf{wp}\;((\mathsf{BeforeCall}\;\texttt{"main"}\;[]), \bullet) \mathbin{@} \varnothing, [e_{\mathsf{plus1}}]_{\mathrm{FFI}} \oplus p_{\mathsf{plus1}}\;\{\_.\,\top\}$$

This is a weakest precondition not assuming any external calls, thus we can apply adequacy, in the form of Theorem 6.1. When doing so, we need to pick the initial state to be the empty state,[4] so that atInit is actually true. Thus, our entire linked program is safe to execute when starting in the default, initially-empty, state. In other words, we have achieved the central aim of this thesis.

Wait, What?! As it turns out, we have not yet done so. What is missing is proving that the program logic we outlined here, in particular rule CombinedCorrect, is sound. But besides this proof, which we give in Section 6.3, we are truly done. To do so, we followed the following recipe:

1. Develop the individual languages, with external calls. (Chapter 4)
2. Develop a linker that can link ABI-compatible languages, allowing external calls in one language to resolve into the other. (Appendix A)
3. Develop a wrapper, that allows executing λ_{ML} code, but wraps its external calls to be ABI-compatible with λ_{C}. (Section 5.2)
4. Extend the wrapper with primitives, so that λ_{C} can interact with its internal state. (Section 5.3)
5. Verify the λ_{ML} program, by assuming a specification for external calls.
6. Verify the λ_{C} program, by assuming a specification for FFI primitives.

[4]Technically, we strengthen our adequacy theorem a bit, so that we can initially assume P if $P * \mathrm{SI}(\sigma)$ is satisfied by the initial state.

7. Show that the wrapper implements the assumed FFI primitive specification. (Section 6.3)
8. Show that the wrapper executes λ_{ML} faithfully, including a proper translation of external call specifications. Section 6.3
9. Show that the linker links this together into a useful result. (Section 6.1.2).

The last three steps are currently combined in the rule CombinedCorrect, which is heavily specialized to the result we wanted to prove here.

6.3 Formal Implementation

We now need to prove that the program logic we used in the previous section actually exists. To do so, we need to come up with a state interpretation of the wrapper state. Additionally, we need definitions of the GC token, and of all the other ghost state we used. These all need to match, they need to validate all the update rules, they need to prove the specifications we assumed about the primitives, and they must validate the CombinedCorrect rule. We saw hinted at the fact the GC token $\mathsf{GC}(\theta)$ is much more complicated than it seems. It does not only establish that θ is the current state of the garbage collector, but also

- stores the λ_{C} points-tos passed in when registering a root.
- allows trading λ_{ML} and block-level points-tos, as described by UpdateMlToBlock and UpdateBlockToMl.
- allows making mutable blocks immutable (UpdateFreeze).
- allows making private locations public (UpdateExpose).

But let us start simple. If all we want is to connect the address map θ in the GC token $\mathsf{GC}(\theta)$ to the actual address map present in the state of the wrapper, we could define both the state interpretation and the GC token like this:

$$\begin{aligned}\mathsf{SI}_{\mathsf{priv}}(\chi, \zeta, \theta, \mathit{rs}) &\triangleq \gamma_\theta \mapsto_{½} \theta * \cdots \\ \mathsf{GC}(\theta) &\triangleq \gamma_\theta \mapsto_{½} \theta\end{aligned}$$

This connects the GC token and the state interpretation using a ghost variable. Unfortunately, we now run into a naming conflict: Ghost variables, which associate a value with a ghost name, are denoted by γ. Block-level locations are also denoted by γ, but are colored differently. Back to the token: It uses a specific ghost variable, namely γ_θ, to connect both parts. Thus, when we know both are present (like while verifying that a primitive has a specification), we can use the agreement properties to find that both contain the same map. Also, we can combine both, so that the ghost variable is fully owned (as $1 = ½ + ½$). This allows us to change the address map, if needed. What we have defined above is by no means the complete definition. It is, however, already a small part of it.

Four Different State Interpretations

Before we continue defining the state interpretation, we should note what "the state interpretation" even is. We constructed our wrapper with state that can have two different sides: The OCaml side, for executing λ_{ML} code, and the C side, for executing λ_{C} code. The C side state is further splittable into the public (linkage-model-defining) and the private (internal) state. Since the wrapper is ABI-compatible with λ_{C}, the public state was defined as λ_{C} state. The public state interpretation is similarly forced to simply be the λ_{C} state interpretation. For the other components of the overall wrapper state, we best reconsider Figure 5.7, reproduced here:

$$
\begin{aligned}
\mathsf{CSideState} \ni \rho_{C} &\triangleq \mathit{LocMap} \times \mathit{BlockStore} \times \mathit{AddrMap} \times \mathit{RootSet} \\
\mathsf{MLSideState} \ni \rho_{ML} &\triangleq \mathit{LocMap} \times \mathit{BlockStore} \times \mathit{RootMap} \times \Sigma_{C} \\
\Sigma \ni \rho &::= \mathsf{CState}(\rho_{C}, \sigma_{C}) \mid \mathsf{MLState}(\rho_{ML}, \sigma_{ML}) \\
\Sigma_{\mathsf{pub}} &\triangleq \Sigma_{C} \\
\Sigma_{\mathsf{priv}} &\triangleq \mathsf{CSideState}
\end{aligned}
$$

We thus need a state interpretation for each side, *i.e.*, for each variant of σ. To increase modularity, we construct these from different parts, as follows:

$$
\begin{aligned}
\mathsf{SI}(\mathsf{CState}(\rho_{C}, \sigma_{C})) &\triangleq \mathsf{SI}_{\mathsf{priv}}(\rho_{C}) * \mathsf{SI}_{\mathsf{pub}}(\sigma_{C}) \\
\mathsf{SI}(\mathsf{MLState}(\rho_{ML}, \sigma_{ML})) &\triangleq \mathsf{SI}_{\mathsf{MLState}}(\rho_{ML}) * \mathsf{SI}(\sigma_{ML}) \\
\mathsf{SI}_{\mathsf{pub}}(\sigma_{C}) &\triangleq \mathsf{SI}(\sigma_{C})
\end{aligned}
$$

As we can see, the public state interpretation is simply that of λ_{C}. The private state interpretation still needs to be defined. By defining the overall state interpretation for the C side as their separated conjunction, we ensure that this state interpretation can always be split into public and private parts, which is required by the linking operator (compare SIJoin, SISplit, and SIAtBoundary). The state interpretation for the OCaml side similarly uses the λ_{ML} state interpretation SI for the λ_{ML} heap σ_{ML}. The other part, $\mathsf{SI}_{\mathsf{MLState}}(\rho_{ML})$, also still needs to be defined. We now go on to define the state interpretation for the C side, that is, $\mathsf{SI}_{\mathsf{priv}}$. This is the state interpretation that must be connected to the GC token $\mathsf{GC}(\theta)$. The other side of the state, which is used to execute λ_{ML} code, does not have a GC token.

The GC Token And The OCaml Side This point needs a bit more explanation. The GC token is needed in λ_{C} to reason about glue code. With the OCaml-C FFI, all the glue code is written in C. In OCaml, the fact that external calls are implemented in another language is (almost) invisible. We therefore also do not need special ghost state to reason about $\lambda_{\mathtt{ML}}$ code in the wrapper, since it should behave like regular OCaml code. Additionally, the garbage collector is invisible in $\lambda_{\mathtt{ML}}$, and does not need to be reasoned about at all. Finally, a GC token is also not necessary for $\lambda_{\mathtt{ML}}$. As part of overall correctness, we later need to prove that a $\lambda_{\mathtt{ML}}$ expression within the wrapper reduces just like it normally would. But to do so, we have access to the full wrapper state interpretation (of the OCaml side), so no special token is needed.

Implementing Roots

We ignore the OCaml side of the state interpretation for now, instead focusing on the C side, and its interaction with the GC token. Our first step is to add the needed support for roots. Remember that while in the C side, the wrapper only stores a roots set rs. In our program logic, we do however have a root points-to, which suggests that there is a roots map rm around somewhere, backing that points-to. This roots map exists in the GC token. Additionally, the GC token tracks that the entire garbage collector state is correct, in that in particular all roots are reachable, and the block-level heap is closed under reachability. To do so, the GC token also needs to know the current block-level heap. This leads to the following definitions:

$$
\begin{aligned}
\mathsf{SI}_{\mathsf{priv}}(\chi, \zeta, \theta, \mathit{rs}) &\triangleq \gamma_\theta \overset{👻}{\mapsto}_{1/2} \theta * \gamma_\zeta \overset{👻}{\mapsto}_{1/2} \zeta * \gamma_{\mathit{rs}} \overset{👻}{\mapsto}_{1/2} \mathit{rs} * \cdots \\
\mathsf{GC}(\theta) &\triangleq \exists \zeta\, \mathit{rs}.\, \gamma_\theta \overset{👻}{\mapsto}_{1/2} \theta * \gamma_\zeta \overset{👻}{\mapsto}_{1/2} \zeta * \gamma_{\mathit{rs}} \overset{👻}{\mapsto}_{1/2} \mathit{rs} * \mathsf{GCroots}(\theta, \zeta, \mathit{rs}) * \cdots \\
\mathsf{GCroots}(\theta, \zeta, \mathit{rs}) &\triangleq \exists \mathit{rm}.\, \boxed{\bullet \mathit{rm}}^{\gamma_{\mathit{rm}}} * \\
&\qquad \mathsf{dom}\ \mathit{rm} = \mathit{rs} * \mathsf{GcClosed}(\zeta, \theta) * \mathsf{GcRooted}(\mathit{rm}, \theta) * \\
&\qquad \mathop{\Huge *}_{a \mapsto v \in \mathit{rm}} \exists w.\, a \mapsto_{\mathsf{C}} w * v \sim^{\theta}_{\mathsf{C}} w \\
a \mapsto^{d}_{\mathsf{root}} v &\triangleq \boxed{\circ_d \{a := v\}}^{\gamma_{\mathit{rm}}}
\end{aligned}
$$

The first thing we notice is that we gained a few more ghost names, so that the GC token know the current block-level heap ζ and the current roots set rs. The actual part related to roots is then handled in $\mathsf{GCroots}$. In there, we first have the authoritative part that back root points-tos. The other four conditions, together, can be understood as a separation logic encoding of GcStateSwitch (see GcStateSwitch). Together, they encode the invariant that the heap is valid. This is used to *e.g.*, prove the specification of alloc, which requires that the current heap is correct as specified by GcStateSwitch. Since the GC token stores all the λ_{C} points-tos of rooted locations, the roots can be changed when a garbage collector run happens. More importantly, this also ensures that we are able to switch back to $\lambda_{\mathtt{ML}}$. With these definitions, we are able to show that we satisfy the specification of registerroot. Additionally, since the GC token contains the C points-tos of rooted locations, we can still write to those using normal C stores, as required by CWP-Store-Root and CWP-Load-Root. This does not need the wrapper state interpretation $\mathsf{SI}_{\mathsf{priv}}$ to be present.

More Points-Tos

In order to implement the ghost theory of block-level points-tos, we can simply again use Iris' standard ghost theory for these. However, we add a twist:

$$\mathsf{BlockLevelHeapAuth}(\zeta) \triangleq \boxed{\bullet \zeta}^{\gamma_\zeta} * \mathop{\huge *}_{\gamma \mapsto \mathsf{blk} \in \zeta} \mathsf{mutability}(\mathsf{blk}) = \mathsf{Imm} \mathrel{-\!\!*} \gamma \mapsto^{\boxdot}_{\mathsf{FFI}} \mathsf{blk}$$

$$\gamma \mapsto^{d}_{\mathsf{FFI}} \mathsf{blk} \triangleq \boxed{\circ_d\{\gamma := \mathsf{blk}\}}^{\gamma_\zeta}$$

Besides the authoritative fragment, we also store a persistent points-to for each immutable block. This allows us to prove the following law:

$$\mathsf{BlockLevelHeapAuth}(\zeta) \mathrel{-\!\!*} \zeta[\gamma] = \mathsf{blk} \mathrel{-\!\!*}\mathrel{-\!\!*} \mathsf{mutability}(\mathsf{blk}) = \mathsf{Imm} \mathrel{-\!\!*} \gamma \mapsto^{\boxdot}_{\mathsf{FFI}} \mathsf{blk}$$

Intuitively, this means that we can always conjure up a persistent points-to for immutable blocks. This also means that no-one can ever have a fully owned points-to for immutable blocks. In fact, such a points-to is never necessary. For example, a closure points-to $\gamma \mapsto_{\mathsf{clos}} \mathsf{rec}\, f\, x.\, e$, which describes an immutable block, is already immutable by definition. We do a similar construction for the visibility map points-tos:

$$\mathsf{VisibilityAuth}(\chi) \triangleq \boxed{\bullet \chi}^{\gamma_\chi} * \mathop{\huge *}_{\gamma \mapsto k} k \neq \mathsf{Priv} \mathrel{-\!\!*} \boxed{\circ_{\boxdot}\{\gamma := k\}}^{\gamma_\chi}$$

$$\mathsf{isPriv}^{d}(\gamma) \triangleq \boxed{\circ_d\{\gamma := \mathsf{Priv}\}}^{\gamma_\chi}$$

$$\mathsf{isLoc}(\ell, \gamma) \triangleq \boxed{\circ_{\boxdot}\{\gamma := \mathsf{Pub}\ \ell\}}^{\gamma_\chi}$$

$$\mathsf{isForeign}(\iota, \gamma) \triangleq \boxed{\circ_{\boxdot}\{\gamma := \mathsf{Fgn}\ \iota\}}^{\gamma_\chi}$$

The reason that we *cache* the persistent points-tos already with the authoritative part is that it allows us to prove the following laws about IsVal and $\sim_{\mathrm{ML}}$:

Theorem 6.5 (Correctness of $\sim_{\mathrm{ML}}$) *$V \sim_{\mathrm{ML}} v$ is the separation logic counterpart of $\mathsf{IsVal}(\chi, \zeta, \gamma, V)$. Formally, we this means we have the following inferences:*

$$\textsc{IsValToSL}\quad \frac{\mathsf{VisibilityAuth}(\chi) \qquad \mathsf{BlockLevelHeapAuth}(\zeta) \qquad \mathsf{IsVal}(\chi, \zeta, v, V)}{V \sim_{\mathrm{ML}} v} *$$

$$\textsc{IsValOfSL}\quad \frac{\mathsf{VisibilityAuth}(\chi) \qquad \mathsf{BlockLevelHeapAuth}(\zeta) \qquad V \sim_{\mathrm{ML}} v}{\mathsf{IsVal}(\chi, \zeta, v, V)} *$$

Proof By induction on V. For IsValToSL, we need the ability to conjure up persistent points-tos, which is possible due to the above construction. □

We can now incorporate both of these into our GC token:

$$\begin{aligned}
\mathsf{SI}_{\mathsf{priv}}(\chi, \zeta, \theta, rs) &\triangleq \gamma_\theta \mapsto_{1/2} \theta * \gamma_\chi \mapsto_{1/2} \chi * \gamma_\zeta \mapsto_{1/2} \zeta * \gamma_{rs} \mapsto_{1/2} rs * \cdots \\
\mathsf{GC}(\theta) &\triangleq \exists \chi\, \zeta\, rs.\, \gamma_\theta \mapsto_{1/2} \theta * \gamma_\chi \mapsto_{1/2} \chi * \gamma_\zeta \mapsto_{1/2} \zeta * \gamma_{rs} \mapsto_{1/2} rs * \\
&\qquad \mathsf{GCroots}(\theta, \zeta, rs) * \\
&\qquad \mathsf{GCblocks}(\chi, \zeta) * \cdots \\
\mathsf{GCblocks}(\chi, \zeta) &\triangleq \mathsf{VisibilityAuth}(\chi) * \mathsf{BlockLevelHeapAuth}(\zeta) * \mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi * \cdots
\end{aligned}$$

Our state interpretation is now almost complete. As we can see, it only consists of ghost variables. Instead, the actual "state interpretation," the one that *e.g.*, backs our various points-tos, is our GC token. Since the GC token is always required when we interact with the wrapper, moving out the state interpretation is not an issue. In fact, it is a strength, since it enables language-local reasoning. We ensure the update laws are usable when just working in the λ_C program logic by simply giving λ_C access to the state interpretation. While reasoning in λ_C, we are then able to update some of these resources, as long as that update is compatible with the actual value of all the ghost variables. In our GC token, we also track that $\mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi$. This is an invariant of our operational semantics, which is also required by some rules, most notably MlToC. We track this invariant in our GC token, instead of in the state interpretation, since this allows using this fact when proving some of the update laws.

Virtual λ_{ML} Heaps

We now get to the most interesting part of the GC token. This is the implementation of the view reconciliation laws, which allowed gradual and local converting of λ_{ML} points-tos to block-level ones. To implement this, we use a virtual λ_{ML} heap. We mentioned that when switching from the OCaml side to the C side, the λ_{ML} heap is completely serialized into the block-level heap. But if we want our λ_{ML} points-tos to live on, we need to back them with *something*. That something is the virtual λ_{ML} heap. When switching sides from OCaml to C, the physical λ_{ML} heap that has just disappeared lives on as the virtual heap. And when switching back, it is precisely this virtual heap that becomes the new physical heap. This means that this virtual heap satisfies all of the invariants required when switching sides, like that all its contents are high-level representatives of what is stored in the block-level heap. But since this heap is virtual, it can (within these constraints) be freely modified.

To now implement our update rules UpdateMlToBlock, we use the special λ_{ML} heap cell $\lightning$: This value denotes that the heap cell is *not available* in λ_{ML}, but only in the block-level part. When using that rule to convert a $\lambda_{\mathtt{ML}}$ points-to into a block-level points-to, the virtual heap is updated to instead store $\lightning$. The rule for switching back, UpdateBlockToMl, forces the user verifying the code to pick a new $\lambda_{\mathtt{ML}}$ array that should be stored back into the virtual heap to replace $\lightning$. This works precisely because our semantics make this choice angelically non-deterministic. Since our operational semantics already "support" the special value $\lightning$, no further tracking is needed. It is explicitly supported to first switch sides to C, then exchange a $\lambda_{\mathtt{ML}}$ points-to to a block-level one, and then switch sides back to OCaml without exchanging the points-to back to an $\lambda_{\mathtt{ML}}$ one. Of course, this means that this location is inaccessible in $\lambda_{\mathtt{ML}}$, since block-level points-tos can not be used in $\lambda_{\mathtt{ML}}$.

To further implement this switching, we need a *per-location invariant*, which describes the possible states each pair of connected locations $\mathsf{isLoc}(\ell, \gamma)$ can be in:

$$\begin{aligned}\mathsf{PerLocInvariant}(\zeta, \sigma_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{V}\, \vec{v}.\ & (\ell \mapsto_{\mathrm{ML}} \lightning * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut}, 0, \vec{v})) \\ & \vee \left(\sigma_{\mathrm{ML}}[\ell] = \vec{V} * \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v}\right) \\ & \vee (\sigma_{\mathrm{ML}}[\ell] = \lightning * \gamma \notin \zeta) \\ & \vee (\ell \notin \sigma_{\mathrm{ML}} * \gamma \notin \zeta)\end{aligned}$$

The last two disjuncts are phony, and are only here due to imprecisions in our formal model of $\lambda_{\mathtt{ML}}$.[5] The first two disjuncts are actually interesting, since they describe the two states that the view reconciliation laws switch between. In the first state, the block-level points-to is handed out to the user, and we store the $\lambda_{\mathtt{ML}}$ points-to, storing $\lightning$. In the second state, the $\lambda_{\mathtt{ML}}$ points-to is handed out. When it is traded for a block-level points-to, this block-level points-to must represent the contents of the $\lambda_{\mathtt{ML}}$ points-to. Hence, we require $\vec{V} \sim_{\mathrm{ML}} \vec{v}$. This per-location invariant is now added to the GC token, as part of $\mathsf{GCblocks}$:

$$\begin{aligned}\mathsf{GCblocks}(\chi, \zeta) \triangleq \exists \sigma_{\mathrm{ML}}.\ & \mathsf{VisibilityAuth}(\chi) * \mathsf{BlockLevelHeapAuth}(\zeta) * \mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi * \\ & \mathsf{SI}(\sigma_{\mathrm{ML}}) * (\forall \ell \in \mathsf{dom}\ \sigma_{\mathrm{ML}}.\, \exists \gamma.\, \chi[\gamma] = \mathsf{Pub}\ \ell) * \\ & \mathop{\Huge *}_{\gamma \mapsto \mathsf{Pub}\ \ell \in \chi} \mathsf{PerLocInvariant}(\zeta, \sigma_{\mathrm{ML}}, \gamma, \ell)\end{aligned}$$

While it does not look like it, this is a separation logic encoding of $\mathsf{IsStoreBlocks}$ and $\mathsf{IsStore}$, as indicated by the following theorem:

[5] The reason for them is that the visibility map χ is allowed to contain mappings for locations ℓ that are not actually allocated. This is because in $\lambda_{\mathtt{ML}}$, one can store location literals on the heap that are not the result of an allocation. This missing distinction between runtime and source values is also present in many other languages modelled in Iris, and arises partially because we model untyped languages.

Theorem 6.6 *Let* $\mathsf{GCblocks}(\chi, \zeta)$ *be given and let* σ_{ML} *be the one existentially quantified in* $\mathsf{GCblocks}$*. The following pure fact then holds:*

$$\exists \zeta'\ \zeta_{\mathrm{ML}}.\ \zeta = \zeta' \mathbin{\dot\cup} \zeta_{\mathrm{ML}} \wedge \mathrm{IsStoreBlocks}(\chi, \sigma_{\mathrm{ML}}, \zeta_{\mathrm{ML}}) \wedge \mathrm{IsStore}(\chi, \zeta, \sigma_{\mathrm{ML}})$$

Proof By induction on χ, with the rest quantified. However, the induction is not on every occurrence of χ. Additionally, a helper block-level heap ζ_{plus} is needed at some points. The full details can be found in Coq. □

Future Worlds

Our GC token is now almost complete. The only thing missing is support for freezing a block (UpdateFreeze), and for exposing a block (UpdateExpose). These two relations correspond to the two relations $\overset{\mathsf{freeze}}{\rightsquigarrow}$ and $\overset{\mathsf{expose}}{\rightsquigarrow}$ of the operational semantics (see Figure 5.5). In our program logic, we were able to already freeze or expose blocks before switching the side back to OCaml. To implement this, we use the same trick as before: virtual heaps. Specifically, we have the virtual block-level heap ζ_{future}, and the virtual visibility map χ_{future}. We call these "future" by analogy to Kripke/future world semantics. Indeed, the two relations $\overset{\mathsf{freeze}}{\rightsquigarrow}$ and $\overset{\mathsf{expose}}{\rightsquigarrow}$ can be understood as defining future world relations. To now merge support for these operations into our GC token, we change it like this:

$$\begin{aligned}\mathsf{GC}(\theta) \triangleq \exists \chi\ \zeta\ rs.\ & \gamma_\theta \overset{\circ}{\mapsto}_{1/2} \theta * \gamma_\chi \overset{\circ}{\mapsto}_{1/2} \chi * \gamma_\zeta \overset{\circ}{\mapsto}_{1/2} \zeta * \gamma_{rs} \overset{\circ}{\mapsto}_{1/2} rs * \\ & \zeta \overset{\mathsf{freeze}}{\rightsquigarrow} \zeta' * \chi \overset{\mathsf{expose}}{\rightsquigarrow} \chi' \\ & \mathsf{GCroots}(\theta, \zeta', rs) * \\ & \mathsf{GCblocks}(\chi', \zeta') * \cdots\end{aligned}$$

This means that all previously defined separation logic concepts, including the points-tos, now operate on the future block-level heap and visibility map. This makes the verification of our primitives a bit harder. For example, for alloc, we now only know that the block in the future block-level heap is mutable. This is sufficient to conclude that the block in the actual block-level heap is mutable, but only because $\overset{\mathsf{freeze}}{\rightsquigarrow}$ is defined to only allow making mutable blocks immutable, not the other way around.

The State Interpretation For The OCaml Side

We now have a state interpretation for the C side of the wrapper state. This must now be augmented with a state interpretation for the OCaml side. This state obligation must in particular allow one to switch sides, as described by MlToC and CToMlAngelic. For this to work, the state interpretation for the OCaml side needs to store the resources of both the GC token and the C side state interpretation. The main difficulty in defining this interpretation is that while on the OCaml side, both the block-level heap ζ and the $\lambda_{\mathtt{C}}$ heap σ_{C} are "missing something". Specifically, the roots are removed from σ_{C} and all blocks that describe $\lambda_{\mathtt{ML}}$ locations are removed from ζ. However, we don't want to (or can not actually, for $\lambda_{\mathtt{C}}$) remove these parts in the ghost theory. Instead, we again use virtual heaps, so that these parts can remain. The full definition is as follows:

$$\begin{aligned}
\mathsf{SI}_{\mathsf{MLState}}(\chi, \zeta, \mathit{rm}, \sigma_{\mathrm{C}}) \triangleq{}& \gamma_\theta \overset{\text{👻}}{\mapsto} \varnothing * \gamma_\chi \overset{\text{👻}}{\mapsto} \chi * \gamma_\zeta \overset{\text{👻}}{\mapsto} \zeta * \gamma_{\mathit{rm}} \overset{\text{👻}}{\mapsto} \mathsf{dom}\ \mathit{rs}\ * \\
& \mathsf{MLblocks}(\chi, \zeta) * \\
& \mathsf{SI}(\sigma_{\mathrm{C}} \mathbin{\dot\cup} ((\lambda\_.\,\dagger)\ \langle\$\rangle\ \mathit{rm})) * \boxed{\bullet \mathit{rm}}^{\gamma_{\mathit{rm}}} * \mathop{\Huge *}_{a \mapsto \_ \in \mathit{rm}} \exists w.\ a \mapsto_{\mathrm{C}} \dagger \\
\mathsf{MLblocks}(\chi, \zeta) \triangleq{}& \exists \zeta_{\mathrm{ML}}.\ \mathsf{VisibilityAuth}(\chi) * \mathsf{BlockLevelHeapAuth}(\zeta \mathbin{\dot\cup} \zeta_{\mathrm{ML}}) * \\
& \mathsf{dom}\ \zeta \subseteq \mathsf{dom}\ \chi * \\
& (\forall \gamma \in \mathsf{dom}\ \zeta_{\mathrm{ML}}.\ \exists \ell.\ \chi[\gamma] = \mathsf{Pub}\ \ell) * \\
& \mathop{\Huge *}_{\gamma \mapsto \mathsf{Pub}\ \ell \in \chi} \mathsf{PerLocInvML}(\zeta, \zeta_{\mathrm{ML}}, \gamma, \ell) \\
\mathsf{PerLocInvML}(\zeta, \zeta_{\mathrm{ML}}, \gamma, \ell) \triangleq{}& \exists \vec{v}.\ (\ell \mapsto_{\mathrm{ML}} \lightning * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut}, 0, \vec{v})) \\
& \vee (\gamma \notin \zeta * \gamma \mapsto_{\mathrm{FFI}} \mathsf{B}(\mathsf{Mut}, 0, \vec{v})) \\
& \vee (\gamma \notin \zeta * \gamma \notin \zeta_{\mathrm{ML}})
\end{aligned}$$

To define a virtual $\lambda_{\mathtt{C}}$ heap, we simply take the roots map (which contains all addresses deleted from the original heap), and convert it into a $\lambda_{\mathtt{C}}$ heap where every location is set to the special value $\dagger$ denoting deallocated locations. The specific value does not matter, we could as well have picked 0. For the block-level heap, we have ζ_{ML} as the virtual heap. This must be public, disjoint from the actually existing ζ, and satisfy a similar per-location invariant. This invariant has three cases. The first describes the case that we did a view reconciliation update to switch a $\lambda_{\mathtt{ML}}$ location to its block-level representative, but did not switch back. The second describes that we are a virtual location, in which case the value we are storing does not matter. The third case is again phony. This state invariant for the OCaml side satisfies all the properties we want it to satisfy. In particular, it allows us to switch states as defined by MLToC or CToMLAngelic: Each state interpretation can show that there is no undefined behavior, and that we can again re-establish the state interpretation of the other side. For this theorem, the GC token is understood as being part of the state interpretation (so that it is absorbed into the OCaml side state interpretation, and comes back when switching to C).

Theorem 6.7 (State Switching Correctness)

$$\textsc{SwitchCorrectMlToC}\qquad \frac{\mathsf{SI}(\mathsf{MLState}(\rho_{\mathrm{ML}},\sigma_{\mathrm{ML}})) \qquad (\vec{V},\rho_{\mathrm{ML}},\sigma_{\mathrm{ML}})\ \hat{\looparrowright}_{\mathrm{M2C}}\ (\vec{w},\rho_{\mathrm{C}},\sigma'_{\mathrm{C}})}{\mathsf{atBoundary} * \mathsf{SI}(\mathsf{CState}(\rho_{\mathrm{C}},\sigma_{\mathrm{C}})) * \exists \theta\, \vec{v}.\, \mathsf{GC}(\theta) * \vec{V} \sim_{\mathrm{ML}} \vec{v} * \vec{v} \sim^{\theta}_{\mathrm{C}} \vec{w}}\Rrightarrow^{*}$$

$$\textsc{SwitchCorrectCToMl}\qquad \frac{\mathsf{atBoundary} \qquad \mathsf{SI}(\mathsf{CState}(\rho_{\mathrm{C}},\sigma_{\mathrm{C}})) \qquad \mathsf{GC}(\theta) \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v} \qquad \vec{v} \sim^{\theta}_{\mathrm{C}} \vec{w}}{\exists \rho_{\mathrm{C}}\, \sigma_{\mathrm{ML}}.\, \mathsf{SI}(\mathsf{MLState}(\rho_{\mathrm{ML}},\sigma_{\mathrm{ML}})) * \dot{\Rrightarrow}(\vec{w},\rho_{\mathrm{C}},\sigma'_{\mathrm{C}})\ \hat{\looparrowright}_{\mathrm{C2M}}\ (\vec{V},\rho_{\mathrm{ML}},\sigma_{\mathrm{ML}})}^{*}$$

The proof of this is not straightforward. It proceeds by induction on χ, but several inductions are necessary, and in addition, induction is not on all occurrences of χ. For more details, we refer to the Coq mechanization. The second rule is interesting, since the update is delayed. This is because our weakest precondition only allows an update after the demonic choices have been made, but this rule already determines angelic choices.

6.3.1 Boundary and Initialization

Two things are still missing: The boundary token atBoundary, and the token atInit describing that main is safe to call. We earlier described how the C side of the state is our boundary state. Defining the boundary token is then easy: It is a ghost variable, that is true whenever we are on the C side, and false otherwise:

$$\begin{aligned}
\mathsf{atBoundary} &\triangleq \gamma_{\mathrm{boundary}} \overset{\circ}{\mapsto}_{1/2} \mathsf{true}\\
\mathsf{SI}_{\mathsf{MLState}}(\chi,\zeta,rm,\sigma_{\mathrm{C}}) &\triangleq \cdots * \gamma_{\mathrm{boundary}} \overset{\circ}{\mapsto}_{1/2} \mathsf{false}\\
\mathsf{SI}_{\mathsf{priv}}(\chi,\zeta,\theta,rs) &\triangleq \cdots * \gamma_{\mathrm{boundary}} \overset{\circ}{\mapsto}_{1} \mathsf{false}
\end{aligned}$$

Finally, we had the token atInit describing that we could call main. This describes that our wrapper state is currently completely empty. It is defined as follows:

$$\begin{aligned}
&\mathsf{atInit} \triangleq \gamma_{\mathrm{init}} \overset{\circ}{\mapsto}_{1/2} \mathsf{true}\\
&\quad \mathsf{SI}_{\mathsf{priv}}(\chi,\zeta,\theta,rs) \triangleq \cdots * \exists i.\, \gamma_{\mathrm{init}} \overset{\circ}{\mapsto}_{1/2} i\, *\\
&\qquad\qquad\qquad (i = \mathsf{true} \twoheadrightarrow \mathsf{GC}(\varnothing) * \chi = \varnothing * \zeta = \varnothing * rs = \varnothing)\\
&\mathsf{SI}_{\mathsf{MLState}}(\chi,\zeta,rm,\sigma_{\mathrm{C}}) \triangleq \cdots * \gamma_{\mathrm{init}} \overset{\circ}{\mapsto}_{1} \mathsf{false}
\end{aligned}$$

Thus, if we have the atInit token while in C (which is where we are when calling main), we know the state is empty. Additionally, we do not need to pass a GC token, since it is initially part of the state interpretation.

Establishing Correctness

To now prove that all of these constructions are correct for our operational semantics, we need to prove two things. The first is showing that all primitives satisfy the specifications we described in Section 6.2. The second is showing that our wrapper faithfully executes λ_{ML} code. Formally, we show this simulation theorem:

Theorem 6.8 (Wrapper Simulation Correctness)

$$\textsc{WrapperSimulation}\quad \frac{\mathsf{wp}\ e\ @\ \varnothing, \Psi\ \{V.\ Q(V)\}}{\mathsf{wp}\ (\lceil e \rceil, \bullet)\ @\ [e_{\mathsf{main}}]_{\text{FFI}}, [\Psi]_{\text{FFI}}\ \{w.\ \exists V\, v\, \theta.\ \mathsf{GC}(\theta) * v \sim^{\theta}_{\mathrm{C}} w * V \sim_{\text{ML}} v * Q(V)\}}$$

Proof By simulation/Löb induction on the given weakest precondition. When arriving at a value, the state needs to be switched to C. For external calls, the state also needs to be switched, and switched back when returning. □

This is needed to prove the specification of callback and main. All the other primitives can even be proven correct without this. Once one has done so, one can prove that the wrapper implements all the primitives:

Theorem 6.9 (Wrapper Primitive Correctness)

$$\textsc{WrapperCorrect}\quad \frac{P \mathrel{-\!\!*} \mathsf{wp}\ e_{\mathsf{main}}\ @\ \varnothing, \Psi\ \{V.\ Q(V)\} \qquad \Psi \mathrel{\#\#} \mathsf{dom}\ [e_{\mathsf{main}}]_{\text{FFI}}}{[\Psi]_{\text{FFI}} \vdash [e_{\mathsf{main}}]_{\text{FFI}} : \Psi^{\Pi}_{\mathsf{FFI}} \sqcup \Psi^{P,Q}_{\mathsf{main}_{e_{\mathsf{main}}}}}$$

To achieve the combined correctness rule CombinedCorrect we saw earlier, we can combine this with the linking operator correctness WP-Link-Modules roughly as follows, ignoring all side conditions:

$$\dfrac{\dfrac{\dfrac{P \mathrel{-\!\!*} \mathsf{wp}\ e_{\mathsf{main}}\ @\ \varnothing, \Psi\ \{V.\ Q(V)\}}{[\Psi]_{\text{FFI}} \vdash [e_{\mathsf{main}}]_{\text{FFI}} : \Psi^{\Pi}_{\mathsf{FFI}} \sqcup \Psi^{P,Q}_{\mathsf{main}_{e_{\mathsf{main}}}}}\ \textsc{WrapperCorrect} \qquad \dfrac{\Psi^{\Pi}_{\mathsf{FFI}} \vdash p : [\Psi]_{\text{FFI}}}{\Psi^{\Pi}_{\mathsf{FFI}} \sqcup \Psi^{P,Q}_{\mathsf{main}_{e_{\mathsf{main}}}} \vdash p : [\Psi]_{\text{FFI}}}\ \textsc{JudgmentWeaken}}{\bot \vdash [e_{\mathsf{main}}]_{\text{FFI}} \oplus p : [\Psi]_{\text{FFI}} \sqcup \Psi^{\Pi}_{\mathsf{FFI}} \sqcup \Psi^{P,Q}_{\mathsf{main}_{e_{\mathsf{main}}}}}\ \textsc{WP-Link-Modules}}{\bot \vdash [e_{\mathsf{main}}]_{\text{FFI}} \oplus p : \Psi^{P,Q}_{\mathsf{main}_{e_{\mathsf{main}}}}}\ \textsc{JudgmentWeaken}$$

The interface Π is the interface available for callbacks. As we saw in Section 6.2, this is usually equal to Ψ. The main difficulty in all these proofs is showing that the we can switch the states from OCaml to C. In other words, once Theorem 6.7 has been shown, the remaining proofs are no longer difficult. Once these have been completed, our program logic is fully formalized. We can be sure that programs proven correct in it are indeed safe to execute using our formal semantics.

6.4 Fractional Trading

While our program logic, as presented, is pretty great, we can make it even greater. For example, the program logic is currently not able to verify the following λ_{C} program:

$$\text{caml\_first}(v) \triangleq \mathsf{Field}(v, 0)$$

Of course, this statement is slightly inaccurate. There are many specifications we can verify for this program. For example, we can verify that when passed a pair, it returns the first component of that pair. Another specification we can verify is that it returns the first element of an λ_{ML} array:

$$\forall \ell\, V.\ \langle \ell \mapsto_{\mathrm{ML}} [V, \ldots] \rangle\ \text{caml\_first}\ [\ell]\ \langle V'.\ V' = V * \text{MLLoc} \mapsto_{\mathrm{ML}}^{V, \cdots} \rangle$$

But this specification has a fully owned points-to as a precondition. In λ_{ML}, we do not need a fully owned points-to to read from a location. Instead, only a *fractionally-owned* points-to $\ell \mapsto_{\mathrm{ML}}^{d} [V, \ldots]$ with $d : \mathbb{Q}_{\boxdot}$ is sufficient. When we look at Field, we can see that the specification already allows reading from a fractionally-owned block-level points-to. This is necessary, since for immutable blocks, we typically do not have the full points-to (compare the definition of $\gamma \mapsto_{\mathsf{blk}[t|\mathsf{imm}]} \vec{v}$ from Figure 6.3).

The problem with verifying the above specification for a fractionally-owned points-to is that we have no way of exchanging a fraction of a λ_{ML} points-to for a fraction of a block-level points-to. The rule UPDATEMLTOBLOCK requires full ownership, since it internally modifies the virtual λ_{ML} heap by setting the passed-in location to $\lightning$. Our goal now is to change this. Specifically, we want update laws that also allow trading fractional points-tos. We describe our solution in three steps. First, we discuss the rules we would like to have. To keep things simple, we ignore discardable fractions for now, and focus only on regular positive fractions $q : \mathbb{Q}_{>0}$. We then show that implementing these wish-to-have rules is impossible. Specifically, we find a program which breaks a program logic with the above rules. This is in Section 6.4.1. To work around this, we have need to tweak our program logic a bit, specifically by adding a new parameter to the GC token. Finally, we strengthen this fixed program logic to also work with discardable fractions. To do so, we prove a new law about discardable fractions, that allows one to un-discard them. We showcase this in Section 6.4.2. But first, let's look at the update laws we would like to have, defined in Figure 6.14.

The first law is a direct generalization of UPDATEMLTOBLOCK. The second law, however, is not. The old law UPDATEBLOCKTOML allowed the person verifying the program to choose a new λ_{ML} array $\vec{V}$ that was then stored in the virtual λ_{ML} heap. When we only trade fractional points-tos, changing what is stored in the virtual λ_{ML} is no longer possible. Thus, the resulting λ_{ML} array that the λ_{ML} points-to points to is

$$\textsc{UpdateMlToBlockFractionalWrong}\quad \frac{\mathsf{GC}(\theta) \qquad \ell \mapsto^{q}_{\mathrm{ML}} \vec{V}}{\exists \gamma\, \vec{v}.\, \mathsf{GC}(\theta) * \gamma \mapsto^{q}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}}$$

$$\textsc{UpdateBlockToMlFractionalWrong}\quad \frac{\mathsf{GC}(\theta) \qquad \gamma \mapsto^{q}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \exists \vec{V}.\, \vec{V} \sim_{\mathrm{ML}} \vec{v}}{\exists \ell\, \vec{V}.\, \mathsf{GC}(\theta) * \ell \mapsto^{q}_{\mathrm{ML}} \vec{V} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}}$$

Figure 6.14: The fractional view reconciliation laws we would like to prove.

existentially quantified. The reason we need to still $\exists \vec{V}. \vec{V} \sim_{\mathrm{ML}} \vec{v}$ in order to apply the rule is for a single corner case: The case where the GC token still has ownership over the entire λ_{ML} points-to, and the virtual λ_{ML} heap is still storing the special value $\lightning$. If we want to hand out a useful λ_{ML} points-to, it should not store $\lightning$, and so we need to give a default value that could be chosen here, should this corner case occur. Thus, these laws are not a strict replacement for the old laws, but only augment them. In particular, we still need the law UpdateBlockToMl.

6.4.1 A Pathological OCaml Program

Unfortunately, these rules can not be implemented. While they work just fine as long as one stays in glue code, they lead to issues when fractional points-tos persist across the language barrier, specifically the switch from OCaml back to C. To see why, consider the following actual OCaml program:

```
let pathologic (l : (int * int) ref) =
  let p = !l in l := (#1 p, #2 p)
```

This program takes a reference `l` pointing to a pair `p`, and stores a new pair into this reference. But the new pair is extensionally equal to the old pair: Both components are the same. Thus, in λ_{ML}, we can verify that such a program is correct even when it does not have full ownership of `l`. Formally, it satisfies the following specification:

$$\{\ell \mapsto^{q}_{\mathrm{ML}} [\langle V_1, V_2 \rangle]\}\ \mathsf{pathologic}(\ell)\ @\ \varnothing, \bot\ \{\langle\rangle.\, \ell \mapsto^{q}_{\mathrm{ML}} [\langle V_1, V_2 \rangle]\}_{\mathrm{ML}}$$

The reason is that the state does not change, since the heap is updated idempotently: $\sigma_{\mathrm{ML}}[\ell := \vec{V}] = \sigma_{\mathrm{ML}}$ if ℓ already stored $\vec{V}$. Thus, the state interpretation does not change, and no full points-to is necessary. The problem is that this new pair might be extensionally equal, but it is not "the same pair." This difference[6] is observable in λ_{C}. If we look at the block-level value corresponding to this pair, then it is changed

[6]Compiler optimizations might change this. We tested it, and were able to observe the difference.

by this assignment, since compiled OCaml code creates a new block to back this new pair, even though the new pair is the same as the old one. Our operational semantics capture this: When switching from the OCaml side to the C side, a new part of the block-level heap ζ_{imm} that stores immutable points-tos (like the one backing this pair) is chosen demonically. It is also demonically chosen whether such a pair is backed by a new block in ζ_{imm}, or by re-using the old block that is part of the old block-level heap. Now imagine that we have the half-owned λ_{ML} points-to $\ell \mapsto^{1/2}_{\mathrm{ML}} [\langle 1,2 \rangle]$, and that the other half has been traded in (using UpdateMlToBlockFractionalWrong): $\gamma \mapsto^{1/2}_{\mathsf{blk}[0|\mathsf{mut}]} [\gamma_{\mathsf{pair}}]$, where $\gamma_{\mathsf{pair}} \mapsto_{\mathsf{blk}[0|\mathsf{imm}]} [1, 2]$. If we now switch to λ_{ML}, and execute the above pathological program, the reference now contains a new block-level location, since the pair is now backed by another block. Instead of being γ_{pair}, it is $\gamma_{\mathsf{newpair}}$. But the half-owned block-level points-to still references γ_{pair}, which is no longer correct. But we also can not modify this points-to, since it might be framed out. We therefore are stuck–our rules can not be implemented.

There are (at least) two ways of fixing this. The first is changing the definition of the block-level points-to to be more fuzzy. One could tweak it so that a store does not return precisely the value that is stored in there, but only returns a value that is observationally equivalent (for some notion of observational equivalence). This should then allow changing the physical state from γ_{pair} to $\gamma_{\mathsf{newpair}}$, without invalidating the old points-to. While this is an interesting approach, we do not implement it. Instead, we pursue a much simpler solution: We simply forbid such partially traded points-tos from existing when switching between OCaml and C. For this, we add a second parameter to our GC token $\mathsf{GC}(\theta)\mathsf{ds}$: the dirty set $\mathsf{ds} : \mathsf{DirtySet} \triangleq \mathcal{P}_{\mathsf{fin}}(\mathsf{Loc})$. This dirty set tracks all block-level locations for which there are fractions of both block-level and λ_{ML} points-tos around. When we use one of the fractional trading laws, this set gets larger:

UpdateMlToBlockFractional
$$\frac{\mathsf{GC}(\theta, \mathsf{ds}) \qquad \ell \mapsto^{q}_{\mathrm{ML}} \vec{V}}{\exists \gamma\, \vec{v}.\ \mathsf{GC}(\theta)\mathsf{ds} \cup \{\gamma\} * \gamma \mapsto^{q}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}}\ast$$

UpdateBlockToMlFractional
$$\frac{\mathsf{GC}(\theta, \mathsf{ds}) \qquad \gamma \mapsto^{q}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \exists \vec{V}.\ \vec{V} \sim_{\mathrm{ML}} \vec{v}}{\exists \ell\, \vec{V}.\ \mathsf{GC}(\theta, \mathsf{ds} \cup \{\gamma\}) * \ell \mapsto^{q}_{\mathrm{ML}} \vec{V} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}}\ast$$

Almost all uses of the GC token are now changes to support an arbitrary large dirty set. The only exceptions are in the specification of callback, and in the definition of the protocol wrapping $[\Psi]_{\mathrm{FFI}}$, since these define what conditions must hold when the state switches to λ_{ML}. Here, the dirty set must be empty, so that executing a pathological program like the one above does not cause issues. Finally, we need some rules to again remove locations from the dirty set. For this, we have a few confrontation laws:

$$\textsc{ConfrontBlock}\quad \frac{\mathsf{GC}(\theta, \mathrm{ds}) \qquad \ell \mapsto^{1}_{\mathrm{ML}} \vec{V} \qquad \mathsf{isLoc}(\ell, \gamma)}{\mathsf{GC}(\theta, \mathrm{ds} \setminus \{\gamma\}) * \ell \mapsto^{1}_{\mathrm{ML}} \vec{V}} *$$

$$\textsc{ConfrontMl}\quad \frac{\mathsf{GC}(\theta, \mathrm{ds}) \qquad \gamma \mapsto^{1}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v}}{\mathsf{GC}(\theta, \mathrm{ds} \setminus \{\gamma\}) * \gamma \mapsto^{q}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v}} *$$

$$\textsc{ConfrontMlBlockFractional}\quad \frac{\mathsf{GC}(\theta, \mathrm{ds}) \qquad \gamma \mapsto^{q_1}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \ell \mapsto^{q_2}_{\mathrm{ML}} \vec{V} \qquad \mathsf{isLoc}(\ell, \gamma)}{q_1 + q_2 \leqslant 1 * \vec{V} \sim_{\mathrm{ML}} \vec{v}} *$$

The last law does not actually remove anything from the dirty set, but it formalizes the connection between these fractionally owned points-tos. As such, it is a generalization of ConfrontMlBlock. Finally, we need to implement this fractional points-to trading. We do this by modifying the per-location invariant to the following:

$$\begin{aligned}
\mathsf{PerLocInvariant}(\mathrm{ds}, \zeta, \sigma_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{V}\, \vec{v}.\ & (\ell \mapsto_{\mathrm{ML}} \text{↯} * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut}, 0, \vec{v})) \\
& \vee (\exists q_1\, q_2.\, \ell \mapsto^{q_1}_{\mathrm{ML}} \vec{V} * \gamma \mapsto^{q_2}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \\
& \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v} * q_1 + q_2 = 1 * \gamma \in \mathrm{ds}) \\
& \vee \left(\sigma_{\mathrm{ML}}[\ell] = \vec{V} * \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v}\right) \\
& \vee (\sigma_{\mathrm{ML}}[\ell] = \text{↯} * \gamma \notin \zeta) \\
& \vee (\ell \notin \sigma_{\mathrm{ML}} * \gamma \notin \zeta)
\end{aligned}$$

The second case is new: It describes that we have fractions of both points-tos, and asserts that the amount we own is exactly 1, so that the sum of all points-tos not saved in the GC token is also at most 1. It also asserts that the location γ is part of the dirty set. The special λ_{ML} heap cell ↯ is only used when ownership is fully moved from λ_{ML} to the block level. As long as a fraction of λ_{ML} points-to ownership remains in the GC token, the λ_{ML} heap stores a proper value. If we need to switch the state from C to OCaml, we know that this case is empty. Thus, we have the same proof obligations we had before we introduced this case, so that the overall program logic can be proven correct.

6.4.2 A Law About Discardable Fractions

The new fractional update laws only allow trading points-tos that are fractionally owned. They forbid trading points-tos that have been discarded. The reason for this is that addition on these is less well-defined, so that handing out a points-to to keep the sum $q_1 + q_2 = 1$ is not possible, unlike for regular fractions. Instead of trying to generalize this equations to discardable fractions, we use the following law about points-tos:

Theorem 6.10 (**Un-Discarding of Points-Tos**)

$$\ell \mapsto^{\boxdot} v \Rrightarrow\!\!\ast\ \exists q'.\, \ell \mapsto^{q'} v$$

In fact, this holds for all ghost state that "updates like a discardable fraction."

Proof The definition of the update modality allows us to inspect the frame, which tells us which other fractionalities are around. Let q be the total sum of these fractionalities, including that of the one we are tying to update. Since they are all valid in combination, and include at least one persistent points-to, we have $q < 1$. But notice that $q < 1 \to \exists q'.\, q + q' < 1$ by simply choosing $q' := \frac{1-q}{2}$. □

To our knowledge, this law is novel. In particular, it was also not part of the paper [13] on which large parts of this thesis are based. With this law, using the update law on a discarded fraction is trivial: Simply un-discard it to a proper fraction, use the update law with this proper fraction, and again discard the result.

Chapter 7

View Reconciliation: Different Approaches

In order to support our view reconciliation rules (Figure 6.8), we added the special $\lambda_{\mathtt{ML}}$ heap cell $\lightning$. As a reminder, the view reconciliation laws allowed us to move the ownership of a $\lambda_{\mathtt{ML}}$ to the block level, so that it could be accessed using glue code. Specifically, these rules allow the user to exchange (or trade) a $\lambda_{\mathtt{ML}}$ points-to for a block-level points-to, and vice-versa. When doing so, one points-to is with the user, while the other points-to is stored internally within the program logic. The heap cell $\lightning$ was used as a placeholder to store in the $\lambda_{\mathtt{ML}}$ heap when the ownership was moved to glue code, it marked that heap cell as inaccessible to $\lambda_{\mathtt{ML}}$. This formal model is not realistic: In actual OCaml, heap cells do not become inaccessible. In this chapter, we develop alternative approaches that do not require this special heap cell. We then see what changes are needed to again verify all the programs we discussed in Section 6.2, as well as those in our original paper [13].

As a first step, we remove the $\lightning$ heap cell from the $\lambda_{\mathtt{ML}}$ language. In Section 7.1, we describe the resulting program logic, where view reconciliation has been restricted to cope with the absence of this token. In Section 7.2, we then try to recover the original view reconciliation laws, by introducing *canonical representatives*. For this, we have to develop a new ghost theory, which we can also use in other places to further strengthen our program logic. These further improvements are outlined in Section 7.3

7.1 A Restrictive Theory

The special heap cell $\lightning$ was originally introduced to answer the question of what $\lambda_{\mathtt{ML}}$ value is stored in a heap cell whose ownership remains at the block level. This question is not easy to answer otherwise. When the ownership of a points-to is at the block level, the user should be able to freely read and modify this block. It is the obligation of the program logic to keep its internally tracked $\lambda_{\mathtt{ML}}$ points-to consistent with the changes made at the block level. If simply storing $\lightning$ is not an option, then the program logic needs to actually find $\lambda_{\mathtt{ML}}$ values for each new block-level value that the user stores in a block. This is not trivial: If the user stores 1 into a block, this could be either a number, or the boolean true. Even worse, if one stores a block-level location into this block, the λ_{ML} value depends on the contents of the block at this location, and recursively on the contents of blocks pointed at by locations stored within this block, *etc.*. We do not want to implement a program logic where changing one block can have a cascading effect that requires updating many λ_{ML} representatives. Even further, there are some block-level values that can not be represented as a λ_{ML} value at all (like a block with tag 1, length 2). So, if we remove $\lightning$, we end up with a theory where there might be locations for which no suitable λ_{ML} value exists. This is actually only an issue when returning to λ_{ML}–before this, the λ_{ML} is virtual, and completely controlled by the program logic. We can therefore still support trading points-tos, as long as all points-tos are traded back to their λ_{ML} form before switching state to the OCaml side. To implement this, we use the mechanism already introduced in Section 6.4: The dirty set ds. Specifically, we change our view reconciliation rules as follows:

$$\textsc{UpdateMlToBlockDirty}\qquad \frac{\mathsf{GC}(\theta, \mathsf{DirtySet}) \qquad \ell \mapsto_{\mathrm{ML}} \vec{V}}{\exists \gamma\, \vec{v}.\ \mathsf{GC}(\theta, \mathsf{ds} \cup \{\gamma\}) * \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \mathsf{isLoc}(\ell, \gamma) * \vec{V} \sim_{\mathrm{ML}} \vec{v}} *$$

$$\textsc{UpdateBlockToMlDirty}\qquad \frac{\mathsf{GC}(\theta, \mathsf{ds}) \qquad \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v}}{\exists \ell.\ \mathsf{GC}(\theta, \mathsf{ds} \setminus \{\gamma\}) * \ell \mapsto_{\mathrm{ML}} \vec{V} * \mathsf{isLoc}(\ell, \gamma)} *$$

Since we enforced that the dirty set ds is empty before switching sides, this effectively requires that all λ_{ML} points-tos that were exchanged for block-level ones are exchanged back. Internally, this is implemented by changing the per-location invariants (where ~~crossed-out~~ parts are considered removed):

$$\begin{aligned}
\mathsf{PerLocInvariant}(\mathsf{ds}, \zeta, \sigma_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{V}\, \vec{v}.\ & (\ell \mapsto_{\mathrm{ML}} \cancel{\lightning}\, \overbrace{[0, \ldots, 0]}^{|\vec{V}|\text{ many}} * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut},0,\vec{v}) * \gamma \in \mathsf{ds}) \\
& \vee (\exists q_1\, q_2.\ \ell \mapsto^{q_1}_{\mathrm{ML}} \vec{V} * \gamma \mapsto^{q_2}_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \\
& \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v} * q_1 + q_2 = 1 * \gamma \in \mathsf{ds}) \\
& \vee \left(\sigma_{\mathrm{ML}}[\ell] = \vec{V} * \gamma \mapsto_{\mathsf{blk}[0|\mathsf{mut}]} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v}\right) \\
& \vee \cancel{(\sigma_{\mathrm{ML}}[\ell] = \lightning * \gamma \notin \zeta)} \\
& \vee (\ell \notin \sigma_{\mathrm{ML}} * \gamma \notin \zeta) \\
\mathsf{PerLocInvML}(\zeta, \zeta_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{v}.\ & \cancel{(\ell \mapsto_{\mathrm{ML}} \lightning * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut},0,\vec{v}))} \\
& \vee (\gamma \notin \zeta * \gamma \mapsto_{\mathrm{FFI}} \mathsf{B}(\mathsf{Mut},0,\vec{v})) \\
& \vee (\gamma \notin \zeta * \gamma \notin \zeta_{\mathrm{ML}})
\end{aligned}$$

For the OCaml side variant, we can simply remove the first case, since all original $\lambda_{\mathtt{ML}}$ points-tos must be traded back to that form before switching, and thus this case is no longer needed. For C, we need to change the first case, which describes what is stored in the virtual $\lambda_{\mathtt{ML}}$ heap when the ownership is moved to the block level. Previously, this was $\lightning$, and now we need a new value, namely an array (of the proper length) just storing zeros. Which concrete value we require does not matter, since we also mark this case as being dirty, by requiring $\gamma \in \mathsf{ds}$, so that one is forced to switch their points-tos back. Additionally, we can remove one of the phony cases.

With this program logic, we can still verify all examples, both of Section 6.2 and of the original paper [13]. This is because these examples never used the feature of keeping ownership of $\lambda_{\mathtt{ML}}$ locations at the block level. A reason for this is that this was already not possible for semantically well-typed glue code: There, the semantic type of $\lambda_{\mathtt{ML}}$ locations is already such that it forces one to move the ownership back to $\lambda_{\mathtt{ML}}$ before continuing to execute well-typed code.

7.2 Canonical Representatives

We mentioned earlier that finding a $\lambda_{\mathtt{ML}}$ value for each block-level value seemed impossible. In this chapter, we will try this approach anyway. To make selecting the $\lambda_{\mathtt{ML}}$ value that represents the low-level modification easier, we want there to be canonical representatives. Specifically, we want a $\lambda_{\mathtt{ML}}$ value V to be the canonical representative of a block-level value v, such that this relation is right-unique (functional): V should not be the canonical representative of any other block-level value v'. For integers, we already have this: The canonical representative of a block-level integer can simply be defined as the corresponding $\lambda_{\mathtt{ML}}$ integer. The canonical representative of a block is what previously caused difficulty. But observe that custom blocks already have canonical representatives, namely foreign values ι. We now try to extend this to all kinds of blocks, so that every location can be represented by a unique foreign value ι. For this, we change the type of visibilities as follows:

$$\mathsf{LocKind} \ni \mathsf{k} ::= \mathsf{Pub}(\ell, \iota) \mid \mathsf{Priv}\,\iota \mid \mathsf{Fgn}\,\iota$$

We could even merge private and foreign visibilities. Now, each block-level location is always assigned a foreign location, in addition to being potentially assigned a $\lambda_{\mathtt{ML}}$ location ℓ.

We now want to change $\mathsf{isForeign}(\iota, \gamma)$ to describe that γ is assigned foreign location ι, no matter which kind of visibility. Our current constructions require this to be persistent. But we still want to have $\mathsf{isPriv}^{d}(\gamma)$ to be fully ownable (with $d = 1$), so that we can expose a location by updating it from $\mathsf{Priv}\,\iota$ to $\mathsf{isLoc}(\ell, \iota)$, as long as the foreign identifier ι stays the same. This requires a custom ghost theory.

7.2.1 Ghost Maps with Remainders

We develop the theory of ghost maps with remainders. These do not only have the regular points-tos $\mapsto$, but also the always-persistent "remainder points-to" $\overset{\mathsf{rem}}{\mapsto}$. This ghost theory is parameterized by a function $\mathsf{F}_{\mathsf{rem}}$, that tells us which part of of the pointed-to value should be persistent and immutable. In the above example, this function is the function $\mathsf{fid} : \mathsf{LocKind} \to \mathsf{ForeignId}$, which tells us the foreign id contained in a visibility annotation, while ignoring the kind (*e.g.*, Pub vs Priv). The semantics of the regular points-to $\mapsto$ are now that it tells us the full value, and it also still allows mutating the value stored in the map (if fully owned). But this mutation is limited: The value of $\mathsf{F}_{\mathsf{rem}}$ must not change. Formally, this is the following law:

$$\textsc{GhostMapInsertRemainder}\qquad \frac{\boxed{\bullet\sigma}^{\gamma} \qquad \ell \mapsto v \qquad \mathsf{F}_{\mathsf{rem}}(v) = \mathsf{F}_{\mathsf{rem}}(v')}{\boxed{\bullet\sigma[\ell := v]}^{\gamma} * \ell \mapsto v'}\ \Rrightarrow$$

We now use ghost maps with remainders to back our visibility map, so that each block-level location has a unique foreign identifier, but still remains updatable from a private block to a public block.

7.2.2 Achieving Canonical Representatives

Formally, we define canonical representatives as follows:

$$\textsc{CanonInt}\ \frac{}{n \sim_{\mathsf{canon}} n}\ * \qquad\qquad \textsc{CanonLoc}\ \frac{\mathsf{isForeign}(\iota, \gamma)}{\textcircled{\iota} \sim_{\mathsf{canon}} \gamma}\ *$$

We can now add this to our per-location invariants. The first case of the C side location invariant is now no longer dirty, since it instead enforces that the $\lambda_{\mathtt{ML}}$ points-to stores the canonical representative of the block-level values stored in the block-level heap:

$$\begin{aligned}
\mathsf{PerLocInvariant}(ds, \zeta, \sigma_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{V}\, \vec{v}.\ & (\ell \mapsto_{\mathrm{ML}} \vec{V} * \zeta[\gamma] = \mathsf{B}(\mathsf{Mut},0,\vec{v}) * \vec{V} \sim_{\mathsf{canon}} \vec{v}) \\
& \vee (\exists q_1\, q_2.\ \ell \mapsto^{q_1}_{\mathrm{ML}} \vec{V} * \gamma \mapsto^{q_2}_{\mathsf{blk[0|mut]}} \vec{v} * \\
& \qquad \vec{V} \sim_{\mathrm{ML}} \vec{v} * q_1 + q_2 = 1 * \gamma \in ds) \\
& \vee \left(\sigma_{\mathrm{ML}}[\ell] = \vec{V} * \gamma \mapsto_{\mathsf{blk[0|mut]}} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v}\right) \\
& \vee (\ell \notin \sigma_{\mathrm{ML}} * \gamma \notin \zeta)
\end{aligned}$$

$$\begin{aligned}
\mathsf{PerLocInvML}(\zeta, \zeta_{\mathrm{ML}}, \gamma, \ell) \triangleq \exists \vec{v}.\ & \left(\ell \mapsto_{\mathrm{ML}} \vec{V} * \zeta_{\mathrm{ML}}[\gamma] = \mathsf{B}(\mathsf{Mut},0,\vec{v}) * \vec{V} \sim_{\mathsf{canon}} \vec{v} * \gamma \notin \zeta\right) \\
& \vee (\gamma \notin \zeta * \gamma \mapsto_{\mathrm{FFI}} \mathsf{B}(\mathsf{Mut},0,\vec{v})) \\
& \vee (\gamma \notin \zeta * \gamma \notin \zeta_{\mathrm{ML}})
\end{aligned}$$

In the per-location invariant for the OCaml side, we need to be a bit more careful. Because ↯ no longer exists, all blocks that back a $\lambda_{\mathtt{ML}}$ array are always moved back to $\lambda_{\mathtt{ML}}$ ownership, which also means that they are *removed* from the block-level heap ζ (compare CToMlAngelic). This includes the blocks whose ownership should remain within glue code. Since these blocks are no longer part of the physical block-level heap, we add them to the virtual block-level heap ζ_{ML}, which keeps the blocks around that were removed during state switching. The final difficulty is proving that the state can be switched from OCaml back to C. There, one has the old virtual heap ζ_{ML}, and a new heap ζ'_{ML} that reflects the changes made by $\lambda_{\mathtt{ML}}$ code. But since the $\lambda_{\mathtt{ML}}$ points-to storing the canonical representative was retained by the wrapper program logic, we know that $\lambda_{\mathtt{ML}}$ did not modify this location. And since further canonical representatives are unique, this allows us to conclude that the block in the new block-level heap ζ'_{ML} is equal to the old block. This allows us to again prove Theorem 6.7.

7.2.3 Finding Canonical Representatives

Almost all primitives are easily verified to still work with this new program logic. The only problematic one is Store_field, since this primitive actually modifies a block that could back a $\lambda_{\mathtt{ML}}$ array. We then need to ensure that there also is a new canonical representative for the new block-level value that is being stored. Unfortunately, this is not straightforward. The issue again is that we do not properly distinguish between source and runtime values, so that it seems to be possible to store a block-level location γ that is not in the domain of the visibility map χ, and therefore is without an assigned foreign identifier. Luckily, this can be worked around. Primitives like Store_field are not invoked with block-level values directly, but only with $\lambda_{\mathtt{C}}$ values that encode these block-level values. If we want to store a location γ into a block, we invoke the primitive with w such that $\gamma \sim^{\theta}_{C} w \Leftrightarrow \theta[\gamma] = w$. We thus know that all locations for which a canonical representative needs to be found are in $\mathsf{dom}\ \theta$. If we change our operational semantics to always ensure $\mathsf{dom}\ \theta \subseteq \mathsf{dom}\ \zeta$, we can ensure that all such values have canonical representatives. Adding this requirement to the operational semantics does not cause any harm. Without this requirement, it was already the case that both θ and ζ, which are demonically chosen/extended when switching sides from OCaml to C, could grow without bound[1] (see MlToCDemonic). Almost all requirements imposed by our semantics only provide lower bounds (*e.g.*, θ must contain at least all roots), there are no restrictions on these maps that would prevent them from being larger than required. If we now enforce $\mathsf{dom}\ \theta \subseteq \mathsf{dom}\ \zeta$, than the address map θ is bounded. But it is not really bounded, since it is bounded by ζ, which can still grow without bound to accommodate all locations in $\mathsf{dom}\ \theta$.

[1]The address map θ is actually bounded, but only for locations $\gamma \in \mathsf{dom}\ \zeta$.

7.3 Further Improvements to the Ghost Theory

We have now developed a custom ghost theory for the visibility map θ. We can extend this further, by requiring that this map, as well as the fragments (which define points-tos) has the injectivity requirement we imposed on these maps. This allows us to prove IsLocInjective and IsFgnInjective without requiring that the GC token is present as well:

$$\textsc{IsLocInjectiveNoGC}\quad \frac{\mathsf{isLoc}(\ell_1, \gamma_1) \qquad \mathsf{isLoc}(\ell_2, \gamma_2)}{\ell_1 = \ell_2 \iff \gamma_1 = \gamma_2}\,*$$

$$\textsc{IsFgnInjectiveNoGC}\quad \frac{\mathsf{isForeign}(\iota_1, \gamma_1) \qquad \mathsf{isForeign}(\iota_2, \gamma_2)}{\iota_1 = \iota_2 \iff \gamma_1 = \gamma_2}\,*$$

Besides this, we can use our ghost maps with remainders to back both the block-level heap ζ, and the $\lambda_{\mathtt{ML}}$ heap σ_{ML}. For $\lambda_{\mathtt{ML}}$, we can pick $\mathsf{F}_{\mathsf{rem}}$ to be the length of the block. This establishes that in $\lambda_{\mathtt{ML}}$, the length of a block never changes. Similarly, for the block-level heap, we can make the *header* of a block persistent. The header stores the tag, and for regular blocks also stores the length. With this addition, we can now verify that the polymorphic equality function we verified as part of the original paper [13, §5] is safe[2] when invoked with a location. This program crucially relies on the fact that the length of a block does not change. Even further, by using read_tag to guard against non-standard blocks, we can even verify that the program is safe to call on function types. What prevents us from claiming this program is safe on all types are recursive and existential types. Both roll and pack have transparent block-level representations,[3] but we need to know the actual kind of value we are working with. In particular, if this value is a $\lambda_{\mathtt{ML}}$ location, we also need ownership over this location. But if our current type is existential, it might just be $\exists\tau.\,\tau$, which is a type that has location values as elements, but does not (necessarily) carry ownership for them. If it is a recursive type, then the ownership is hidden behind a potentially unbounded number of later modalities.

[2]Note that comparing locations is not proper: In $\lambda_{\mathtt{ML}}$, they are compared by-reference (shallowly), but the $\lambda_{\mathtt{C}}$ function implements a deep comparison, since it can not tell blocks backing references from those backing, *e.g.*, pairs

[3]Both are just syntactic sugar for the identity function, so they are already transparent in $\lambda_{\mathtt{ML}}$.

Chapter 8

Conclusion

In the paper backing this thesis [13], we presented the first program logic for reasoning about multi-language programs constructed from languages with different memory models. While the paper only gives a very brief description of how this is achieved, this thesis covers the full construction in-detail.

The main goal in developing this program logic was *language-locality*: When verifying the correctness of a multi-language program, most of the correctness proof should be able to be carried out in the program logic of each single language. Additionally, when reasoning about code not using multi-language features, this should be as if the whole program was single-language. It is only when reasoning about glue code, that the true multi-language nature of the overall program must be considered. The motivation for this is that in practice, only a small part of a multi-language program is genuine glue code. In that regard, our examples are better understood as excerpts, only attempting to showcase the glue code design patters that are found in larger programs.

Formally, our language-locality shows up in the compositional correctness rules of the wrapper, the linker, and the language-to-module lifting rules (CombinedCorrect). Additionally, the view reconciliation and other update laws are theorems of the λ_C program logic alone, and can therefore be used to reason about glue code within just the λ_C program logic. To achieve this language-locality, we started our development with the two single languages (Chapter 4). We then formally defined how these languages are connected, by means of the wrapper (Sections 5.2 and 5.3) and the linker (Section 5.1.3). The program logic for reasoning about glue code (Section 6.3) is then forced to make the usually disjoint resources of the single-language program logics be compatible. This is achieved by a cleverly constructed state interpretation / GC token, which validate the view reconciliation laws UpdateMlToBlock and UpdateBlockToMl. While much of what we developed is specific to OCaml and C, we expect such view reconciliation laws to show up whenever two languages with different ABIs are combined.

Note that we only claim to be the first program logic for reasoning about multi-language programs across different memory models. Indeed, there is a lot of related work on multi-language program in general.

8.1 Related and Prior Work

The paper [13, §6] already discusses the related work, which covers all of our related work.

Iris-Wasm

Iris-Wasm [37] is a program logic for reasoning about WebAssembly code, including its interaction with the host. In the interaction between both, the host is more "in control" of the WebAssembly parts: It has to instantiate them before they can be executed, and can, *e.g.*, manipulate WebAssembly function tables. The Host takes a role that is more similar to our linker, since it embeds the WebAssembly semantics. The Host responsible for receiving calls that "bubble up" in WebAssembly, and for dispatching them properly. In our work, OCaml and C are completely different languages, which in particular have different memory models, requiring a more sophisticated FFI. They communicate as equal partners via the linker. Instead of using protocols to reason about external calls, these are captured by terminating with a continuation value, which is then handled by the surrounding context.

Cito

Cito [48, 36] is a verified compiler with support for linking with languages. To accomplish this, they build an operational semantics, where calls can be specified using specifications similar to our interfaces. Since they build interfaces into the operational semantics, their interfaces are pure, while ours are separation-logic-based. Cito focuses more on compiler correctness, and while there are single-language program logics, they do not construct a program logic for multi-language programs.

Cogent

Cogent [30, 31] is a functional language with a verified compiler. Cheung *et al.* [5] study how this compiler correctness can be extended to functions implemented in C. To make this work, the paper considers a restricted version of C, and outlines what kinds of C functions can be safely added to Cogent code, by defining requirements on the specifications of these C functions. Since the paper aims to extend a compiler correctness proof, it does not consider how the C functions can be verified, and starts by assuming specifications about them. In contrast, we assume the behavior of the OCaml compiler and runtime, and show how OCaml specifications can be proven for C code. Since OCaml does not have a linear value system, we do not need to formally restrict its specifications.

Logical Relations Across Different Languages

Patterson *et al.* [33] consider several different case studies of language interoperability, where in each case study, the languages have an interesting difference (*e.g.*, affine vs not affine). They then construct logical relations, relating types across both languages. Their model assumes that both languages are compiled to a shared target language, to which semantic safety then carries over. Interestingly, the rules for relating values of both languages are defined by the user, in the shared target language. We do not consider a shared target language, instead linking both languages by constructing the wrapper, which also has the conversion rules for values baked into its operational semantics.

More on Multi-Language Semantics

A lot of work has been focused on verifying compilers that work for more than one language, handling *e.g.*, closure lowering [25, 34], or using logical relations across different languages [35]. Hur and Dreyer [16] verify a compiler from an OCaml-like language with a garbage collector to a lower-level language, and support linking with low-level code. However, they do not build a program logic, and the behavior of low-level code must be specified in refinement style, by writing identically behaved high-level code. There also is work on extending CompCert to handle linking with code written in another language [46, 44, 20, 12]. The present work does include any compiler, nor does it attempt to verify any.

More theoretically, Matthew and Findler [26] considered multi-language programs already in 2007, and identified concepts like boundary states, which we used for great success. Our semantics are also inspired by DimSum [41], which gives a compositional framework for designing multi-language semantics. The reason our semantics are not expressed in DimSum itself is that it was published after we started our work. Its treatise of angelic non-determinism, *undefined*, and *no behavior* helped clarify our approach and our definitions–even though multi-relations were already introduced in 2012 [24]. While DimSum introduces a modular way of describing multi-language state switches, it does neither feature a program logic, nor combine languages with similar features as ours.

Formal Accounts of the OCaml-C-FFI

Furr and Forster [11] construct a type system for verifying C glue code that is written when interacting with OCaml. They include a soundness proof, and cover full inductive datatypes. They type system uses the OCaml types assigned to external C functions as a basis. While their type system is sound for glue code, it can not be used to ensure soundness of entire multi-language programs using all features of C.

8.2 Future Work

We mentioned in Section 3.5 that our formal model of the OCaml FFI lacks several features, like support for exceptions, for OCaml 5's multi-threading features, for full constructor type definitions, for fully featured custom blocks, for more specialized allocation procedures like `caml_alloc_small`, or for direct C access to the block-level heap. All of these are future work. While some of these (*e.g.*, `caml_alloc_small`) seem rather easy to handle, properly accounting for non-linear control flow or for multi-threaded language interoperability seems more difficult.

Another issue with our operational semantics is the use of angelic (and demonic) non-determinism, which do not exist in physical machines. We used angelic non-determinism to model the translation between OCaml and block-level values. One approach for showing that our semantics are realizable is by formally verifying an OCaml compiler, which compiles OCaml code down to code operating directly on block-level values, so that no angelic non-determinism is needed. One can then show that such a translation preserves our original semantics, but no longer has angelic non-determinism.

Finally, our theory can be used as a foundation for more practical formal methods. For instance, our program logic could be encoded into Viper [28], a tool for automatically checking whether programs implement separation logic specifications, using SMT solvers.

8.3 Coq Development

This thesis (as well as the paper [13] it is based on) are fully formalized in the Coq proof assistant.[1] The Coq development of this thesis is part of the supplementary material, and can also be found under the following DOI:

`http://doi.org/10.5281/zenodo.8197195`

Since this thesis presents three variations of the view reconciliation rule (see Chapter 7), there are three versions of the Coq development. The first version, called `melocoton_lightning`, contains the version presented in Chapter 6, including the special λ_{ML} heap cell ↯. The version of Section 7.1 is called `melocoton_restricted`, it includes the version where points-to trading is restricted. Finally, the version of Section 7.2 with canonical representatives is called `melocoton_canon_reps`. It also includes the extended use of ghost maps with remainders, as described in Section 6.4, as well as the example discussed in that section.

All three versions have the fractional view reconciliation rules discussed in Section 7.3. Additionally, all three versions can be found on GitHub.

[1]This is not fully true: The two example programs `caml_plus1` and `caml_plus1_ref` have not been formally verified.

Appendix A

Linking

We give a detailed description of the linking operator, which was only discussed very briefly in Chapter 5. In Section 4.1.2, we saw a simple linking operator for linking two programs written in the same language. (The linking operator was simply the disjoint union.) We now develop a linking operator that can link different languages. Note that the concept of a linking operator is overloaded here: The linking operator on modules produces a module $\lambda_L \oplus \lambda_R$ given two modules λ_L and λ_R. The linking operator on programs takes two programs, p_L of λ_L and p_R on λ_R, and combines these to a program $p_L \oplus p_R$ of the language $\lambda_L \oplus \lambda_R$. These linkage operators require that the two modules being linked employ the same linkage models. Remember that the linkage model describes the interface for function calls, in particular the type of their arguments, and also describe the shared mutable state that all functions are able to operate on. Formally, this is captured by a type of values, describing the arguments of functions, and a type of state. For the C linkage model, this is simply the type of values and state defined by λ_C.

It turns out that simply using the entire state of a language to define the linkage model is too simplistic. This would make it impossible to define a module that can be linked with C, but also has its own private state, used for internal book-keeping, that should not be accessible to functions it can link with. But this, allowing modules to retain some private state when linking with another module, and those modules being able to rely on that state remaining unchanged, is precisely what we need to define the wrapper in Section 5.2.

Public and Private State

We thus introduce a distinction between the public (or ABI, or linkage model) state and the private state of a module. The *public state* is what defines the linkage model. The *private state* is internal to the module. As a further relaxation, we allow modules to blur the distinction between public and private state, by defining a third kind of state: *overall state*. Most of the time, a module will operate on its

$$
\begin{aligned}
\mathit{Ctx}_{\lambda_L \oplus \lambda_R} &\ni K ::= \bullet \mid K \cdot K_L \mid K \cdot K_R \\
\mathit{SExpr}_{\lambda_L \oplus \lambda_R} &\ni E ::= v \mid \lceil e_L \rceil \mid \lceil e_R \rceil \mid \mathsf{BeforeCall}\ fn\ \vec{v} \mid \mathsf{HandleCallL}\ e_L \mid \mathsf{HandleCallR}\ e_R \\
\mathit{Expr}_{\lambda_L \oplus \lambda_R} &\ni e \triangleq \mathit{SExpr}_{\lambda_L \oplus \lambda_R} \times \mathit{Ctx}_{\lambda_L \oplus \lambda_R} \\
\Sigma_{\lambda_L \oplus \lambda_R} &\ni \sigma ::= \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},L}, \sigma_{\mathsf{priv},R}) \\
&\qquad \mid \mathsf{StateL}(\sigma_L, \sigma_{\mathsf{priv},R}) \mid \mathsf{StateR}(\sigma_{\mathsf{priv},L}, \sigma_R) \\
\Sigma_{\mathsf{priv},\lambda_L \oplus \lambda_R} &\ni \sigma_{\mathsf{priv}} \triangleq \Sigma_{\mathsf{priv},L} \times \Sigma_{\mathsf{priv},R} \\
\mathit{Func}_{\lambda_L \oplus \lambda_R} &\ni F ::= F_L \mid F_R
\end{aligned}
$$

Figure A.1: Syntax of the linking module $\lambda_L \oplus \lambda_R$.

overall state. This state can be split into public and private state, and inversely, public and private state can be joining into the overall state. But this splitting and joining is not possible all the time. It only needs to be possible when the linker intercepts an external call to switch execution to the other language. Since in the linker, splitting and joining is angelic, not being able to split (or join) is undefined behavior. Later, we use separation logic to enforce that these kinds of splits and joins are possible. Formally, we now have the overall state σ, which can be split into the public state Σ_{pub} and Σ_{priv}. To describe the valid splittings and mergings, the relation $\mathit{Split} : \Sigma \to \Sigma_{\mathsf{pub}} \to \Sigma_{\mathsf{priv}} \to \mathbb{P}\mathrm{rop}$ is used. So, for a module to be linkable, it needs to define its public and private state, as well as its splitting relation Split.

The Linking Operator

We now define the linking operator on modules. Given two modules λ_L and λ_R, the linking operator defines a new module $\lambda_L \oplus \lambda_R$. These two languages must be ABI-compatible, that is, they have the same kind of values, and the same public state. They also bring their own types of expressions e_L and e_R, their own notions of private and overall states, of evaluation contexts, *etc.*. We index these with L and R to indicate their side.

The syntax of this new module is defined in Figure A.1. The definition of contexts is already interesting: To allow external calls to repeatedly switch sides, evaluation contexts are now an arbitrary composition of contexts of either side. If we think of contexts as encoding the call stack, then we allow call stacks that switch between both implementations. The definition of functions of the linking module is tightly coupled to the linking operator on programs. When linking two programs p_L and p_R, linking should resemble taking the disjoint union $p_L \dot{\cup} p_R$. We want a program in the linked module to consist of several functions, each implemented either in λ_1 or λ_2. This formally captures the intuitive understanding of a multi-language program outlined in Section 2.5, with the choice of language notably being *per-function*. The

$$\begin{tikzcd}
\mathsf{BeforeCall}\ \mathrm{fn}\ \vec{v} \arrow[r] & \mathsf{HandleCallL}\ e_L \arrow[r] & \lceil e_L \rceil \arrow[r, dotted] & \lceil e'_L \rceil \arrow[r] & \lceil v \rceil \arrow[r] & v \\
\mathsf{Boundary} & \mathsf{Boundary} & \mathsf{StateL} & \mathsf{StateL} & \mathsf{StateL} & \mathsf{Boundary}
\end{tikzcd}$$

Execution trace of a function not performing external calls.

$$\begin{tikzcd}
 & & & \mathsf{BeforeCall}\ \mathrm{fn}\ \vec{v} \arrow[r, dotted] & v & \\
\lceil e_L \rceil \arrow[r] & \lceil K_L[\mathsf{call}_L\ \mathrm{fn}\ \vec{v}] \rceil \arrow[r] & K_L & & K_L \arrow[r] & \lceil K_L[v] \rceil \\
\mathsf{StateL} & \mathsf{StateL} & \mathsf{Boundary} & & \mathsf{Boundary} & \mathsf{StateL}
\end{tikzcd}$$

Execution trace when one side calls a function implement in the other language.

Figure A.2: Two execution traces of a linked program.

definition of functions as $F_L \mid F_R$ facilitates exactly this. Given a program p of $\lambda_L \oplus \lambda_R$, each function in this program is either a function F_L or a function F_R Thus, the program p of the linked module can always be written as a combination of two programs, each in one language: $p = p_L \dot{\cup} p_R$.[1]

The definition of expressions themselves is also more complicated: An expression is now a simple expression SExpr, along with a context (representing the call stack). By designing our expressions like this, we avoid having to distinguish head redexes. This is fine, since a module does not require so detailed tracking of head redexes. The simple expressions, which describe the "currently active method," have four different variations, of which two exist twice due to symmetry between λ_L and λ_R. One expression merely is a value, indicating that the top-level method is done and can return. The next one is a plain embedding of an expression of one of the source languages, which is the default case when the method has not terminated. The next two expressions are *administrative*. The BeforeCall expression is the state a new stack frame starts in. Then, if the call is resolved internally, it will step to HandleCall, which then steps to the contained expression. The BeforeCall expression is also the special expression that could facilitate an external call out of the entire linked module. This is useful if one wishes to link more than two modules. It is also the expression we start in when showing that a linked program satisfies a protocol. Apart from this, the administrative expressions exist to manipulate the state, mostly by splitting and merging it. Since the sequence of operations is a bit convoluted, we explain it in Figure A.2. There, we show some sample reduction sequences, showing both the expression and the kind of state.

The first subfigure shows what happens once a function is invoked. In the first step,

[1]Formally, we first f'map either program with the injection $F_{L/R} \to F$.

$$\textsc{SLinkStepL}\quad \frac{\begin{array}{c}\neg\mathrm{IsVal}\ e_L \\ (e_L, \sigma_L) \twoheadrightarrow^{p_L}_{L} X_L \qquad \forall e'_L\ \sigma'_L.\, X_L(e'_L, \sigma'_L) \to X(\lceil e'_L \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma'_L))\end{array}}{(\lceil e_L \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma_L)) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkToExtCallL}\quad \frac{\begin{array}{c}fn \notin \mathsf{dom}\ p_L \\ \mathit{Split}_L\ \sigma_L\ \sigma_{\mathsf{pub}}\ \sigma_{\mathsf{priv},L} \qquad X(\mathsf{BeforeCall}\ fn\ \vec{v}, \mathsf{K} \cdot \mathsf{K}_L, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L}))\end{array}}{(\lceil \mathsf{K}_L[\mathsf{call}\ fn\ \vec{v}] \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma_L)) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkToValL}\quad \frac{\mathit{Split}_L\ \sigma_L\ \sigma_{\mathsf{pub}}\ \sigma_{\mathsf{priv},L} \qquad X(v, \mathsf{K}, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L}))}{(\lceil v \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma_L)) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkResolveCallL}\quad \frac{\begin{array}{c}p_L[fn] = F_L \\ \mathit{applyFunc}_L\ F_L\ \vec{v} = e_L \qquad X(\mathsf{HandleCallL}\ e_L, \mathsf{K}, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L}))\end{array}}{(\mathsf{BeforeCall}\ fn\ \vec{v}, \mathsf{K}, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L})) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkHandleCallL}\quad \frac{\mathit{Split}_L\ \sigma_L\ \sigma_{\mathsf{pub}}\ \sigma_{\mathsf{priv},L} \qquad X(\lceil e_L \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma_L))}{(\mathsf{HandleCallL}\ e_L, \mathsf{K}, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L})) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkReturnL}\quad \frac{\mathit{Split}_L\ \sigma_L\ \sigma_{\mathsf{pub}}\ \sigma_{\mathsf{priv},L} \qquad X(\lceil \mathsf{K}_L[v] \rceil, \mathsf{K}, \mathsf{StateL}(\sigma_{\mathsf{priv},R}, \sigma'_L))}{(v, \mathsf{K} \cdot \mathsf{K}_L, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L})) \twoheadrightarrow^{p}_{\oplus} X}$$

$$\textsc{SLinkTerminate}\quad \frac{}{(v, \bullet, \mathsf{Boundary}(\sigma_{\mathsf{pub}}, \sigma_{\mathsf{priv},R}, \sigma_{\mathsf{priv},L})) \twoheadrightarrow^{p}_{\oplus} X}$$

Figure A.3: Operational Semantics of $\lambda_L \oplus \lambda_R$.

the function is found to be defined on the left side, and is dispatched there. In the next step, the public state and the private state of λ_L are merged into the overall state of λ_L. Afterwards, the left expression starts reducing. Eventually, it reaches a value, which is then, in a final state, extracted to the administrative return state. In the second case, we show what happens if somewhere during the actual execution of e_L, an external call to the other side happens. Formally, this happens when e_L steps to a call $\mathsf{call}_L\ \mathtt{fn}\ \vec{v}$, somewhere within an evaluation context K_L. (Remember that each language and each module has a special expression call for external calls, which is tested for here.) If that call is not immediately resolved inside of the module it came from (*i.e.*, is not internal), we step to the administrative state $\mathsf{BeforeCall}$, while also splitting the overall state of λ_L into its public and private components. However, we do so by creating a new stack frame, which means that we add the context K_L to the current list of stack frames. Formally, if $e = (K_L[\mathsf{call}_L\ \mathtt{fn}\ \vec{v}], K)$ was our overall expression (composed of a simple expression and a list of stack frames K), we step to $(\mathsf{BeforeCall}\ \mathtt{fn}\ \vec{v}, K \cdot K_L)$. Then, this reduces (in several steps) as shown in the first subfigure (but now with e_R and StateR). Eventually, this stack frame terminates with a value v, which is then returned by substituting it into the next lower stack frame. This is also where the state is merged again.

The actual formal semantics can be found in Figure A.3. It defines the step-relation $\twoheadrightarrow_{\oplus}$ of the linking module. We only give the semantics for the left side. All rules, except SLinkTerminate, also exist symmetrically for the right side, by simply replacing all L with R (and vice-versa). The rule SLinkStepL describes the normal case, where the left side simply steps according to the semantics of λ_L. When the left side wants to make an external call (which are undefined behavior in λ_L alone), this is picked up by SLinkToExtCallL, which steps to the boundary state. There are now two options. If the call is not implemented by either module, we have undefined behavior of the overall linking module. (If we use the linking operator on itself, this is where the outer linking operator would intercept the call.) The other case is that either side can implement the call, which is indicated by rule SLinkResolveCallL (for the left side), which continues by stepping to $\mathsf{HandleCallL}$. From there, SLinkResolveCallL uses $\mathfrak{applyFunc}_L$ to invoke the left function with the arguments passed from the other side. Eventually, a call returns with SLinkToValL, followed by SLinkReturnL, which passes the returned value back to the calling stack frame. The final rule, SLinkTerminate, makes values (which indicate termination) have *no behavior*.

To conclude that our linking operator defines a module, we need to prove that it never exhibits *no behavior*, except for the one allowed by SLinkTerminate.

Theorem A.1 (Absence Of *No Behavior* For Linking) *Let* $(E, K) : \mathrm{Expr}_{\lambda_L \oplus \lambda_R}$ *not be a value and* σ *be given. If* $((E, K), \sigma) \twoheadrightarrow^{p}_{\oplus} X$*, then* X *is nonempty.*

Proof The only rules where X is not obviously non-empty is SLinkStepL, and symmetrically SLinkStepR. There, absence of *no behavior* follows from the fact that λ_L, being a module, also never exhibits *no behavior*. □

Bibliography

[1] 2023. The OCaml manual – Chapter 22: Interfacing C with OCaml. `https://v2.ocaml.org/manual/intfc.html`

[2] Ralph-JR Back. 1989. Changing data representation in the refinement calculus. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, Vol. 2. IEEE Computer Society, 231–232.

[3] Lars Birkedal and Aleš Bizjak. 2022. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. (2022). Available at `https://iris-project.org/tutorial-material.html`.

[4] 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`

[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`

[6] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae* (*Proceedings*), Vol. 75. Elsevier, 381–392.

[7] 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`

[8] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[9] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. *ACM SIGPLAN Notices* 53, 4 (2018), 242–255.

[10] Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey, and Jan Menz. 2022. Semantics Lecture Notes. (2022). Available at `https://plv.mpi-sws.org/semantics-course/`.

[11] 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`

[12] 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`

[13] 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. In *OOPSLA*. ACM.

[14] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[15] 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`

[16] 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`

[17] ISO. 2011. *C11 Standard*. `/bib/iso/C11/n1570.pdf` ISO/IEC 9899:2011.

[18] 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`

[19] 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`

[20] Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C

components. In *PLDI*. ACM, 1095–1109. https://doi.org/10.1145/3453483.3454097

[21] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[22] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP* (*LNCS, Vol. 10201*). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[23] Robbert Jan Krebbers. 2015. *The C standard formalized in Coq*. Ph. D. Dissertation. [Sl]:[Sn].

[24] 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

[25] 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

[26] 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

[27] 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

[28] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation* (*VMCAI*) (*LNCS, Vol. 9583*), B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[29] 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

[30] 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`

[31] 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`

[32] Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

[33] 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`

[34] 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`

[35] 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`

[36] 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`

[37] 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.

[38] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[39] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of functional programming* 24, 5 (2014), 529–607.

[40] 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`

[41] 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`

[42] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*. Springer, 359–374.

[43] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30.

[44] 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`

[45] 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`

[46] 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`

[47] 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).

[48] 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`