Realistic Realizability: Specifying ABIs You Can Count On

ANDREW WAGNER, Northeastern University, USA
ZACHARY EISBACH, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

The Application Binary Interface (ABI) for a language defines the interoperability rules for its target platforms, including data layout and calling conventions, such that compliance with the rules ensures "safe" execution and perhaps certain resource usage guarantees. These rules are relied upon by compilers, libraries, and foreign-function interfaces. Unfortunately, ABIs are typically specified in prose, and while type systems for source languages have evolved, ABIs have comparatively stalled, lacking advancements in expressivity and safety.

We propose a vision for richer, semantic ABIs to improve interoperability and library integration, supported by a methodology for formally specifying ABIs using realizability models. These semantic ABIs connect abstract, high-level types to unwieldy, but well-behaved, low-level code. We illustrate our approach with a case study formalizing the ABI of a functional source language in terms of a reference-counting implementation in a C-like target language. A key contribution supporting this case study is a graph-based model of separation logic that captures the ownership and accessibility of reference-counted resources using modalities inspired by hybrid logic. To highlight the flexibility of our methodology, we show how various design decisions can be interpreted into the semantic ABI. Finally, we provide the first formalization of library evolution, a distinguishing feature of Swift's ABI.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Separation logic**; • **Software and its engineering → Semantics**; **General programming languages**.

Additional Key Words and Phrases: application binary interfaces, type soundness, semantics, logical relations, separation logic, reference counting, program logics

ACM Reference Format:
Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (October 2024), 30 pages. https://doi.org/10.1145/3689755

1 Introduction

What is an ABI? The Swift *Library Evolution Manifesto* defines it as "the run-time contract for using a particular API (or for an entire library), including things like symbol names, calling conventions, and type layout information" [Apple 2015]. Adherence to this contract ensures safe execution and, for certain ABIs, may even ensure stronger guarantees such as memory safety. It is a promise between components about what each component assumes and what it guarantees.

Authors' Contact Information: Andrew Wagner, Northeastern University, Boston, USA, ahwagner@ccs.neu.edu; Zachary Eisbach, Northeastern University, Boston, USA, eisbach.z@northeastern.edu; Amal Ahmed, Northeastern University, Boston, USA, amal@ccs.neu.edu.

ACM 2475-1421/2024/10-ART315
https://doi.org/10.1145/3689755

Current ABI Landscape. Not every language publicizes an ABI, but internally, every compiler maintains its own invariants which may be thought of as an *internal* or undocumented ABI. Stabilization of an ABI is a *big* commitment since a *stable* ABI means that future versions of the compiler will not be at liberty to vary any of the promises made by the ABI. Therefore, the big design question is what exactly those promises are. For instance, promising rigid layouts, as C does, affords opportunities for optimization, yielding better performance but sacrificing extensibility. On the other hand, promising flexible but dynamic layouts, as Swift does [Apple 2015, 2017], permits evolvability at the cost of performance overhead. This tradeoff has been more carefully examined in recent years by the Swift designers. In particular, the Swift ABI is designed to support *library evolution* by default, which affords library developers the flexibility to make certain changes to their A*P*I without breaking A*BI* compatibility. At the same time, it allows libraries to explicitly opt into a rigid layout in the interest of performance.

Because this tradeoff is difficult to resolve, most languages decide not to adopt a stable ABI. Thus, internally, they can have the best of both worlds: they can optimize while also evolving over time. The downside, however, is that this closes the language off from the rest of the world, as no other language or library can interoperate with it, not even older or newer versions of the same language. Instead, all interoperability must be routed through a different ABI that is stable, which in practice is typically the C ABI. This indirect interaction is often lossy, which means that if the language has an advanced type system, say like Rust's, it cannot expect the world to uphold its invariants—it can only expect what the much more impoverished C ABI requires.

While it would be nice to have richer ABIs—e.g., for languages like Rust, whose type system guarantees invariants involving ownership and borrowing—a limiting factor is that ABIs are specified in prose. Hence, including richer invariants would make them harder to understand and abide by. The lack of formal specification also makes it impossible to formally verify that a compiler or library is ABI compliant, which may undermine the very benefits that a richer ABI was intended to provide.

Realizability to the Rescue. In this paper we propose a methodology, based on *realistic realizability* models [Benton 2006], for specifying ABIs. The main idea is to specify the meaning of source language types as sets of target-level configurations. As originally employed by Benton [Benton and Tabareau 2009; Benton and Zarfaty 2007], such models were used to prove that compilers ensure type soundness. Our key observation is that these models must specify all required invariants on data layout, calling conventions, and potentially richer properties such as ownership, aliasing restrictions, or garbage freedom, so they have all the right ingredients for specifying ABIs.

In fact, realizability models are just an instance of *logical relations*, except that the types that index the relations and the terms that inhabit them are drawn from different languages. We draw the following analogy between standard logical relations and our "realizable" semantic ABIs.

ABI-Compliant Code Typically, we say that a term that inhabits a logical relation is *semantically well typed*. From the perspective of an ABI, this means that the target code that inhabits the relation is ABI compliant with the source type indexing the relation.

ABI-Compliant Compiler The Fundamental Property of a logical relation says that all syntactically well typed terms are semantically well typed. From the perspective of an ABI, this corresponds to compiler compliance, i.e., that the compiler maps syntactically well typed source terms to ABI-compliant target terms.

Safe Linking In a standard logical relation, we can show that code that is not syntactically well typed is still semantically well typed, and this means that it can be composed with other semantically well typed code. From the perspective of an ABI, this means that any target code that is ABI compliant can safely be linked with code expecting that type, regardless of

whether the compliant code came from the same or a different compiler for the source, was compiled from a different source language, or was written directly in the target.

Library Evolution In a standard logical relation, one type may semantically refine another (semantic subset). From the perspective of an ABI, this corresponds to library evolution; i.e., if the new type of a library refines its old type, then the new library may be used by any client in place of the old one.

The idea of using realizability models to relate high-level types to low-level code is hardly new [Benton and Tabareau 2009; Benton and Zarfaty 2007], but here we are treating the *model itself* as an independent artifact, not just a proof device.

Case Study: ABI for Quick. We demonstrate our realizability approach by formalizing the ABI of a functional source language, Quick. Like Swift, Quick is automatically reference-counted (ARC) [Apple 2024b] and its ABI (as presented in § 5) supports library evolution as advocated by Swift's ABI designers [Apple 2015, 2017]. Our realizability model is specified using a separation logic extended with a novel modality that supports reasoning about the ownership of *shares* of a reference-counted resource. The main challenge in designing the model was in defining this new modality and specifying the composition of these shares so that the reference counts "add up" correctly.

Quick is implemented in an untyped C-like target language with a memory model similar to CompCert's [Leroy et al. 2014]. Our realizability model is thus indexed by Quick types and inhabited by configurations of the target language. The target is clearly not "binary", so in what sense are we constructing an application *binary* interface? In the strictest sense, "the" ABI for a language is more commonly a *family* of ABIs, one for each supported *platform*. Nonetheless, a practical alternative to building an ABI from scratch in this way is to build atop an *existing* ABI family, ideally one that already supports a number of platforms (e.g., C). In choosing this approach, one would sacrifice some control—we *cannot* specify which registers are saved on every platform—for some convenience—we *need not* specify which registers are saved on every platform. There is precedent for defining an ABI on top of an existing language's ABI (e.g., the proposed Rust ABI, crABI [RFCs 2023]). In this paper, we adopt this approach by specifying an ABI for our typed functional language at the level of a C-like target, though in practice one can refine another language's ABI without actually compiling to it, as we do.

Contributions. This paper proposes the use of *realizability logical relations* for a semantic specification of ABIs that supports both the analysis of ABI design and the verification of ABI compliance. We justify our novel methodology for ABIs by applying the technique to a case study and demonstrating that it indeed captures expected properties of an ABI. Concretely, we make the following contributions.

- We propose a vision for rich, semantic specifications of ABIs using realizability logical relations specified in separation logic. We demonstrate our approach by applying it to a core functional language, Quick, with an automatic reference counting (ARC) compiler into a C-like target (§ 2).
- We develop a novel graph-based model of separation logic for reference counting with two distinguishing modalities: the jump modality $@_\ell\, P$, which expresses the shared ownership of a reference counter at ℓ guarding a resource satisfying P; and the reachability modality $\diamond\, P$, which expresses the accessibility of a resource satisfying P through a chain of references (§ 3.1).
- We use our domain-specific separation logic to specify a semantic ABI for Quick (§ 3.2) and prove that our compiler is ABI compliant (§ 3.3).

- Since the specification of an ABI is fundamentally a *design* problem, we treat a number of potential variations on the semantic ABI and show that the technique would scale in the presence of various practical considerations (§ 4).
- We give the first formalization of a Swift-style ABI with library evolution (§ 5).

Definitions and proofs elided from this paper are included in our supplementary material [Wagner et al. 2024].

2 The Language Stack

In this section, we present our functional source language, Quick, and C-like target language, along with our automatic reference counting (ARC) compiler. For now, we only compile closed, non-recursive functions, to avoid too much complexity in the first ABI we present in §3. We will extend the Quick compiler and ABI with recursion and closures in § 4.1.

2.1 Source Language

$$\mathsf{Type} \ni \mathsf{T} ::= \mathbb{Z} \mid \overline{\mathsf{T_1}} \to \mathsf{T_2} \mid \mathsf{X}$$
$$\mathsf{Expr} \ni \mathsf{e} ::= \mathsf{x} \mid \mathsf{let\,x = e_1; e_2} \mid \mathsf{n} \mid \mathsf{e_1} \oplus \mathsf{e_2} \mid \mathsf{fn\,f\,\overline{x}\{e\}} \mid \mathsf{e_1\,\overline{e_2}} \mid \mathsf{\{\overline{s : e}\}} \mid \mathsf{e.s} \mid \mathsf{s\,e} \mid \mathsf{case\,e_1\,\{\overline{s\,x \Rightarrow e_2}\}}$$

$$\mathsf{Sig} \ni \Sigma ::= \varnothing \mid \Sigma, \mathsf{k\,X\,\{\overline{s : T}\}} \qquad \mathsf{Kind} \ni \mathsf{k} ::= \mathsf{struct} \mid \mathsf{enum} \qquad \mathsf{Ctx} \ni \Gamma ::= \varnothing \mid \Gamma, \mathsf{x : T}$$

$\boxed{\Sigma;\Gamma \vdash \mathsf{e} : \mathsf{T}}$ "Under signature Σ and context Γ, expression e has type T"

$$\frac{\Gamma_1 \vdash \mathsf{e_1} : \mathsf{T_1} \quad \Gamma_2, \mathsf{x} : \mathsf{T_1} \vdash \mathsf{e_2} : \mathsf{T_2} \quad \Gamma_2 \not\ni \mathsf{x}}{\Gamma_1, \Gamma_2 \vdash \mathsf{let\,x = e_1; e_2} : \mathsf{T_2}}\ (\mathsf{let}) \qquad \mathsf{x} : \mathsf{T} \vdash \mathsf{x} : \mathsf{T}\ (\textsc{var}) \qquad \varnothing \vdash \mathsf{n} : \mathbb{Z}\ (\mathbb{Z}\text{-I})$$

$$\frac{\Gamma \ni \mathsf{x} : \mathsf{T}' \quad \Gamma, \mathsf{x} : \mathsf{T}' \vdash \mathsf{e} : \mathsf{T}}{\Gamma \vdash \mathsf{e} : \mathsf{T}}\ (\textsc{dup}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathsf{T}}{\Gamma, \mathsf{x} : \mathsf{T}' \vdash \mathsf{e} : \mathsf{T}}\ (\textsc{drop}) \qquad \frac{\Gamma_1 \vdash \mathsf{e_1} : \mathbb{Z} \quad \Gamma_2 \vdash \mathsf{e_2} : \mathbb{Z}}{\Gamma_1, \Gamma_2 \vdash \mathsf{e_1} \oplus \mathsf{e_2} : \mathbb{Z}}\ (\mathbb{Z}\text{-}\oplus)$$

$$\frac{\Gamma, \mathsf{f} : \overline{\mathsf{T_1}} \to \mathsf{T_2}, \overline{\mathsf{x} : \mathsf{T_1}} \vdash \mathsf{e} : \mathsf{T_2} \quad \Gamma \not\ni \mathsf{f}, \overline{\mathsf{x}}\ \text{distinct}}{\Gamma \vdash \mathsf{fn\,f\,\overline{x}\{e\}} : \overline{\mathsf{T_1}} \to \mathsf{T_2}}\ (\to\text{I}) \qquad \frac{\overline{\Gamma_1 \vdash \mathsf{e_1} : \mathsf{T_1}} \quad \Gamma_2 \vdash \mathsf{e_2} : \overline{\mathsf{T_1}} \to \mathsf{T_2}}{\overline{\Gamma_1}, \Gamma_2 \vdash \mathsf{e_2\,\overline{e_1}} : \mathsf{T_2}}\ (\to\text{E})$$

$$\frac{\Sigma \ni \mathsf{struct\,X\,\{\overline{s : T}\}} \quad \overline{\Sigma;\Gamma \vdash \mathsf{e} : \mathsf{T}}}{\Sigma;\overline{\Gamma} \vdash \mathsf{\{\overline{s : e}\}} : \mathsf{X}}\ (\mathsf{struct}\text{-I}) \qquad \frac{\Sigma \ni \mathsf{struct\,X\,\{\overline{s_i : T_i}^{\,i<n}\}} \quad \Sigma;\Gamma \vdash \mathsf{e} : \mathsf{X} \quad j < n}{\Sigma;\Gamma \vdash \mathsf{e.s_j} : \mathsf{T_j}}\ (\mathsf{struct}\text{-E})$$

$$\frac{\Sigma \ni \mathsf{enum\,X\,\{\overline{s_i : T_i}^{\,i<n}\}} \quad \Sigma;\Gamma \vdash \mathsf{e} : \mathsf{T_j} \quad j < n}{\Sigma;\Gamma \vdash \mathsf{s_j\,e} : \mathsf{X}}\ (\mathsf{enum}\text{-I}) \qquad \frac{\Sigma \ni \mathsf{enum\,X\,\{\overline{s : T_1}\}} \quad \Sigma;\Gamma_1 \vdash \mathsf{e_1} : \mathsf{X} \quad \overline{\Sigma;\Gamma_2, \mathsf{x} : \mathsf{T_1} \vdash \mathsf{e_2} : \mathsf{T_2}} \quad \Gamma_2 \not\ni \overline{\mathsf{x}}}{\Sigma;\Gamma_1, \Gamma_2 \vdash \mathsf{case\,e_1\,\{\overline{s\,x \Rightarrow e_2}\}} : \mathsf{T_2}}\ (\mathsf{enum}\text{-E})$$

Fig. 1. The source language, Quick, styled in blue sans serif. Σ is fixed and elided when unused.

The source, Quick, summarized in Fig. 1, is a functional language with first-class recursive functions and mutually recursive algebraic data type definitions. ADTs are *nominal* rather than structural in order to support library evolution, which we describe in § 5.

Internally, the type system looks substructural: typing contexts Γ are split between multiple subexpressions (as in, e.g., let) and leaves in the typing derivation do not admit superfluous entries in the context (as in, e.g., VAR). However, this substructurality is completely hidden from the programmer by the DUP and DROP rules, which recover the usual contraction and weakening properties, respectively. This means that a binding $\mathsf{x} : \mathsf{T}$ can appear *multiple times* in a typing context,

and it is these occurrences that are split across subexpressions. Therefore, unlike a more standard substructural type system in which $\Gamma, x : T$ tacitly implies $\Gamma \not\vdash x$, we require side conditions on binding forms to ensure that every variable in scope has a unique type. Note that shadowing of names can still be achieved by first applying DROPs before the shadowed binder, though we do not allow parameters of the same function to shadow each other.

Importantly, even though DUP and DROP are applied implicitly—insofar as there is no program *syntax* for them—they appear *explicitly* in the typing derivation. Inspired by the Perceus reference counting scheme [Reinking et al. 2021], these explicit rules provide a handle for our type-directed compiler (§ 2.3) to insert `dup` and `drop` instructions for automatic reference counting. In the example below, the program on the left—which would be rejected by a standard linear type system—is type-checked by the derivation on the right, which makes it clear when `dup` and `drop` should be inserted.

$$\text{fn double } x\{x + x\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathbb{Z} \vdash x : \mathbb{Z}}\,\text{VAR} \quad \dfrac{}{x : \mathbb{Z} \vdash x : \mathbb{Z}}\,\text{VAR}}{x : \mathbb{Z}, x : \mathbb{Z} \vdash x + x}\,\mathbb{Z}\text{-}\oplus}{\text{double} : \mathbb{Z} \to \mathbb{Z}, x : \mathbb{Z} \vdash x + x}\,\text{DUP } x, \text{DROP double}}{\varnothing \vdash \text{fn double } x\{x + x\}}\,{\to}\text{I}$$

The type system is parameterized by a *signature* (Σ) of data type definitions, which we leave implicit when unused because it never changes during type-checking. A data type definition $k\,X\,\{\overline{s : T}\}$ includes the data kind k (`struct` or `enum`), the type name X, and a map from selector names s to types T (i.e., the fields of a `struct` or the cases of an `enum`). For simplicity, we assume that all type and selector names are distinct in the signature. Signatures, types, and contexts must be well-formed insofar as they can only refer to type names that are defined in the signature, but there are otherwise no restrictions (e.g., strict positivity) on the positions in which type names can appear recursively. Data type definitions cannot be directly nested, but must use intermediate names. For example,

$$\text{struct Node } \{\text{left} : \text{Tree}, \text{right} : \text{Tree}\},\ \text{enum Tree } \{\text{leaf} : \mathbb{Z}, \text{node} : \text{Node}\}$$

2.2 Target Language

The target, summarized in Fig. 2, is an untyped imperative language intended to approximate a small subset of C. It uses a block-based memory model similar to CompCert's [Leroy et al. 2014], and does not distinguish between the stack and the heap. The language is syntactically restricted to be in single-assignment form (as indicated by `const`) and therefore models local variables with substitution (as in, e.g., [Birkedal et al. 2006; Jung et al. 2017]).

As in the CompCert memory model, a location (ℓ) is a pair of a block identifier—with 0 denoting the null block and code denoting a distinguished immutable code block—and an offset into that block. Arithmetic on a location only affects its offset, so it cannot be used to access memory outside of the location's block. Variables and block cells may only hold word-sized values (`w`). The memory (μ) maps locations with positive block identifiers (and any offset) to the word stored at the location.

The operational semantics is specified as a small-step reduction relation that is closed over a standard set of left-to-right evaluation contexts (K) specified in the supplementary material. It is parameterized by an immutable function table (F) that contains top-level function definitions. Dynamic program configurations contain the memory (μ) and the *size map* (ψ), which tracks which block identifiers (positive numbers) have been allocated so far (via `malloc`) and their respective sizes. The size of the block is tracked because `free` deallocates the entire block at once. Unlike in CompCert's memory model, every top-level function `f` in F is assigned a unique location $\langle \texttt{f} \rangle_{\texttt{F}}$ in a single immutable code block, and occurrences of the name `f` are dynamically replaced by

$$
\begin{array}{llll}
\texttt{Word} & \ni & \texttt{w} & ::= \ \texttt{n} \mid \texttt{null} \mid \ell \mid \text{☣} \\
\texttt{Expr} & \ni & \texttt{e} & ::= \ \texttt{x} \mid \texttt{f} \mid \texttt{w} \mid \texttt{const x} = \texttt{e}_1\texttt{;}\ \texttt{e}_2 \mid \texttt{e}_1\,(\overline{\texttt{e}_2}) \mid \texttt{e}_1 \oplus \texttt{e}_2 \mid \texttt{if}\,(\texttt{e}_1)\,\{\texttt{e}_2\}\ \texttt{else}\ \{\texttt{e}_3\} \\
 & & & \quad \mid \texttt{malloc}\,(\texttt{e}) \mid *\texttt{e} \mid *\texttt{e}_1 = \texttt{e}_2\texttt{;}\ \texttt{e}_3 \mid \texttt{free}\,(\texttt{e}_1)\texttt{;}\ \texttt{e}_2 \mid \texttt{++e} \mid \texttt{--e} \\
\texttt{Funs} & \ni & \texttt{F} & ::= \ \varnothing \mid \texttt{F}, \texttt{f}\,(\overline{\texttt{x}})\,\{\texttt{e}\}
\end{array}
$$

$$
\begin{array}{llllll}
\ell & \in & \text{Loc} & \triangleq & \langle \text{id} : (\mathbb{N} + \text{code}), \text{off} : \mathbb{N} \rangle & \qquad \texttt{null} \triangleq \langle 0, 0 \rangle \\
\psi & \in & \text{Sizes} & \triangleq & \mathbb{N}^+ \xrightarrow{\text{fin}} \mathbb{N}^+ & \qquad \langle b, i \rangle + n \triangleq \langle b, i + n \rangle \\
\mu & \in & \text{Mem} & \triangleq & \text{Loc}_{\mathbb{N}^+} \xrightarrow{\text{fin}} \texttt{Word} & \qquad \texttt{e}_1\texttt{;}\,\texttt{e}_2 \triangleq \texttt{const \_} = \texttt{e}_1\texttt{;}\ \texttt{e}_2 \\
 & & \text{Loc}_X & \triangleq & \{\ell : \text{Loc} \mid \ell.\text{id} \in X\} & \qquad \texttt{e}_1[\texttt{e}_2] \triangleq *(\texttt{e}_1 + \texttt{e}_2) \\
 & & \langle - \rangle_{\texttt{F}} & : & \text{dom}(\texttt{F}) \xrightarrow{\text{inj}} \text{Loc}_{\text{code}} & \qquad \texttt{havoc} \triangleq \texttt{malloc}\,(-1)
\end{array}
$$

$\texttt{F} \vdash (\psi, \mu, \texttt{e}) \longrightarrow (\psi', \mu', \texttt{e}')$ Small-step reduction. Presupposes $\text{dom}(\mu) \subseteq [(b, i) \mid b \in \text{dom}(\psi) \wedge i < n]$.

$$
\frac{}{\texttt{const x} = \texttt{w;}\ \texttt{e} \longrightarrow \texttt{e}[\texttt{w/x}]}\ (\texttt{let}) \qquad \frac{\texttt{F} \ni \texttt{f}\,(\overline{\texttt{x}})\,\{\texttt{e}\}}{\texttt{F} \vdash \texttt{f} \longrightarrow \langle \texttt{f} \rangle_{\texttt{F}}}\ (\texttt{funptr}) \qquad \frac{\texttt{F} \ni \texttt{f}\,(\overline{\texttt{x}})\,\{\texttt{e}\}}{\texttt{F} \vdash \langle \texttt{f} \rangle_{\texttt{F}}\,(\overline{\texttt{w}}) \longrightarrow \texttt{e}[\overline{\texttt{w/x}}]}\ (\texttt{app})
$$

$$
\frac{\texttt{w} \notin \{\texttt{null}, 0, \text{☣}\}}{\texttt{if}\,(\texttt{w})\,\{\texttt{e}_1\}\ \texttt{else}\ \{\texttt{e}_2\} \longrightarrow \texttt{e}_1}\ (\texttt{if-truthy}) \qquad \frac{\texttt{w} \in \{\texttt{null}, 0\}}{\texttt{if}\,(\texttt{w})\,\{\texttt{e}_1\}\ \texttt{else}\ \{\texttt{e}_2\} \longrightarrow \texttt{e}_2}\ (\texttt{if-falsy})
$$

$$
\frac{\mu(\ell) = \texttt{w}}{(\mu, *\ell) \longrightarrow (\mu, \texttt{w})}\ (\texttt{load}) \qquad \frac{n > 0 \quad b \in \mathbb{N}^+ \setminus \text{dom}(\psi)}{(\psi, \mu, \texttt{malloc}\,(\texttt{n})) \longrightarrow (\psi[b \mapsto n], \mu[\langle b, i \rangle \mapsto \text{☣} \mid i < n], \langle b, 0 \rangle)}\ (\texttt{malloc})
$$

$$
\frac{\ell \in \text{dom}(\mu)}{(\mu, *\ell = \texttt{w;}\ \texttt{e}) \longrightarrow (\mu[\ell \mapsto \texttt{w}], \texttt{e})}\ (\texttt{store}) \qquad \frac{\mu(\ell) = \texttt{n} \quad n' = n + 1}{(\mu, \texttt{++}\ell) \longrightarrow (\mu[\ell \mapsto \texttt{n}'], \texttt{n}')}\ (\texttt{incr})
$$

$$
\frac{\psi(b) = n}{(\psi, \mu \uplus [\langle b, i \rangle \mapsto \texttt{w}_i \mid i < n], \texttt{free}\,(\langle b, 0 \rangle)\texttt{;}\ \texttt{e}) \longrightarrow (\psi, \mu, \texttt{e})}\ (\texttt{free})
$$

Fig. 2. Target language (excerpts). Surface syntax is styled with `red typewriter`. Unused and unmodified components are elided.

its function pointer. Uninitialized memory is populated with poison values (☣) [Lee et al. 2017], on which any operation is undefined behavior. By convention, when part of the configuration is elided, the reader should assume it is passed along unmodified.

2.3 Compiler

Like all ABIs, our semantic ABI specification is intended to be *independent* of any particular compiler. Nevertheless, an ABI does prescribe and proscribe certain implementation strategies, so to elucidate those decisions, we first present a candidate compiler for Quick to capture the shape of implementations that the ABI supports. The compiler is type-directed, and, as in prior work on type-directed compilers [Morrisett et al. 1999; Tarditi et al. 1996], this formally means that we compile *typing derivations*, not syntax. We also adopt existing conventions regarding binders; namely, that there is a canonical mapping from source names to target names, and that any name used internally by the compiler does not conflict with a source name. We present excerpts of the preliminary compiler in Fig. 3. It implements a call-by-value, left-to-right reduction strategy.

In this first version of our compiler, all values are stored in memory and are automatically reference counted [Apple 2024b], which means that reference counting instructions are inserted at

$\boxed{\Sigma; \Gamma \vdash e : T \rightsquigarrow e \dashv F}$ "Derivation $\Sigma; \Gamma \vdash e : T$ compiles to e using top-level functions in F."

$$\frac{\Gamma_1 \vdash e_1 : T_1 \rightsquigarrow e_1 \quad \Gamma_2, x : T_1 \vdash e_2 : T_2 \rightsquigarrow e_2 \quad \Gamma_2 \not\ni x}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\, x = e_1; e_2 : T_2 \rightsquigarrow \mathtt{const}\, x = e_1;\ e_2}\ (\mathsf{let}^+) \qquad x : T \vdash x : T \rightsquigarrow x\ (\mathsf{var}^+)$$

$$\frac{\Gamma \ni x : T' \quad \Gamma, x : T' \vdash e : T \rightsquigarrow e}{\Gamma \vdash e : T \rightsquigarrow \underline{\mathsf{dup}}_{T'}(x); e}\ (\mathsf{dup}^+) \qquad \frac{\Gamma \vdash e : T \rightsquigarrow e}{\Gamma, x : T' \vdash e : T \rightsquigarrow \underline{\mathsf{drop}}^{\Sigma}_{T'}(x); e}\ (\mathsf{drop}^+)$$

$$\varnothing \vdash n : \mathbb{Z} \rightsquigarrow \mathtt{const}\, r = \mathtt{malloc}(2);\ r[0] = 1;\ r[1] = n;\ r\ (\mathbb{Z}\text{-}\mathrm{I}^+)$$

$$\frac{\Gamma_1 \vdash e_1 : \mathbb{Z} \rightsquigarrow e_1 \quad \Gamma_2 \vdash e_2 : \mathbb{Z} \rightsquigarrow e_2}{\Gamma_1, \Gamma_2 \vdash e_1 \oplus e_2 : \mathbb{Z}}\ (\mathbb{Z}\text{-}\oplus^+)$$
$$\rightsquigarrow \mathtt{const}\, r = \mathtt{malloc}(2);\ r[0] = 1;\ \overline{\mathtt{const}\, r_i = e_i;}\ r[1] = r_1[1] + r_2[1];\ \overline{\underline{\mathsf{drop}}_{\mathbb{Z}}(r_i); r}$$

$$\frac{\overline{x : T_1} \vdash e : T_2 \rightsquigarrow e \dashv F \quad \overline{x}\ \text{distinct} \quad F \ni f(\overline{x})\,\{e\}}{\varnothing \vdash \mathsf{fn}\, \overline{x}\{e\} : \overline{T_1} \to T_2 \rightsquigarrow \mathtt{const}\, r = \mathtt{malloc}(2);\ r[0] = 1;\ r[1] = f;\ r \dashv F}\ (\to\mathrm{I}^+)$$

$$\frac{\overline{\Gamma_1 \vdash e_1 : T_1 \rightsquigarrow e_1} \quad \Gamma_2 \vdash e_2 : \overline{T_1} \to T_2 \rightsquigarrow e_2}{\overline{\Gamma_1}, \Gamma_2 \vdash e_2\, \overline{e_1} : T_2 \rightsquigarrow \mathtt{const}\, r_2 = e_2;\ \mathtt{const}\, f_2 = r_2[1];\ \underline{\mathsf{drop}}_{\overline{T_1} \to T_2}(r_2); f_2(\overline{e_1})}\ (\to\mathrm{E}^+)$$

$$\frac{\Sigma \ni \mathsf{struct}\, X\, \{\overline{s_i : T_i}^{\,i<n}\} \quad \overline{\Sigma; \Gamma_i \vdash e_i : T_i \rightsquigarrow e_i}^{\,i<n}}{\Sigma; \overline{\Gamma_i}^{\,i<n} \vdash \{\overline{s_i : e_i}^{\,i<n}\} : X \rightsquigarrow \mathtt{const}\, r = \mathtt{malloc}(n+1);\ r[0] = 1;\ \overline{r[i+1] = e_i;}^{\,i<n}\ r}\ (\mathsf{struct}\text{-}\mathrm{I}^+)$$

$$\frac{\Sigma \ni \mathsf{struct}\, X\, \{\overline{s_i : T_i}^{\,i<n}\} \quad \Sigma; \Gamma \vdash e : X \rightsquigarrow e \quad j < n}{\Sigma; \Gamma \vdash e.s_j : T_j \rightsquigarrow \mathtt{const}\, r = e;\ \mathtt{const}\, r_j = r[j+1];\ \underline{\mathsf{dup}}_{T_j}(r_j); \underline{\mathsf{drop}}^{\Sigma}_{X}(r); r_j}\ (\mathsf{struct}\text{-}\mathrm{E}^+)$$

$$\frac{\Sigma \ni \mathsf{enum}\, X\, \{\overline{s_i : T_i}^{\,i<n}\} \quad \Sigma; \Gamma \vdash e_j : T_j \rightsquigarrow e_j \quad j < n}{\Sigma; \Gamma \vdash s_j\, e_j : X \rightsquigarrow \mathtt{const}\, r = \mathtt{malloc}(3);\ r[0] = 1;\ r[1] = j;\ r[2] = e_j;\ r}\ (\mathsf{enum}\text{-}\mathrm{I}^+)$$

$$\frac{\Sigma \ni \mathsf{enum}\, X\, \{\overline{s_i : T_i}^{\,i<n}\} \quad \Sigma; \Gamma \vdash e : X \rightsquigarrow e \quad \overline{\Sigma; \Gamma', x_i : T_i \vdash e_i : T \rightsquigarrow e_i}^{\,i<n} \quad \Gamma_2 \not\ni \overline{x}}{\Sigma; \Gamma, \Gamma' \vdash \mathsf{case}\, e\, \{\overline{s_i\, x_i \Rightarrow e_i}\} : T}\ (\mathsf{enum}\text{-}\mathrm{E}^+)$$
$$\rightsquigarrow \mathtt{const}\, r = e;\ \mathtt{if}\,(r[1] = i)\ \overline{\left\{\mathtt{const}\, x_i = r[2];\ \underline{\mathsf{dup}}_{T_i}(x_i); \underline{\mathsf{drop}}^{\Sigma}_{X}(x); e_i\right\}}^{\,i<n}\ \mathtt{else}\ \{\mathtt{havoc}\}$$

Fig. 3. Candidate Quick compiler with ARC.

compile-time. The dup (variously called "retain" or "increment") instruction increases the reference count, and the drop (variously called "release" or "decrement") instruction decreases the reference count. While the reference count is non-zero, the reference is live and should not be deallocated. Conversely, once the reference count hits zero after a drop, the reference is dead and should be deallocated. One weakness of reference counting compared to garbage collection is that reference cycles can create deadlocks or memory leaks, which necessitates the use of some external mechanism, like weak references. However, our functional source language has no means to create reference cycles—data types may be recursive, but their inhabitants cannot be cyclic.

Data Layout. Every introduction form allocates and stores type-directed data into memory, along with a word at the head of the block to hold the reference count, a non-negative integer. Integers require only a single additional word for storage when compiled. For now, we only compile closed, non-recursive functions, which need just one word to store the function pointer; we revisit recursion and closures in § 4.1. Keeping in mind that all values are word-sized (just locations, for now), a struct needs as many additional words as it has fields, which are stored in *declaration order*. An enum needs two additional words, one for the tag—the index of the constructor, in declaration order—and one for the payload.

$$
\begin{array}{lll}
\underline{\texttt{dup}}_{\mathsf{T}}\,(\texttt{r}) & \triangleq & \texttt{++r} \\
\underline{\texttt{drop}}^{\Sigma}_{\mathsf{T}}\,(\texttt{r}) & \triangleq & \texttt{const c = --r; if (c) \{c\} else } \left\{\underline{\texttt{destr}}^{\Sigma}_{\mathsf{T}}\,(\texttt{r})\right\} \\
\underline{\texttt{destr}}_{\mathsf{T}}\,(\texttt{r}) & \triangleq & \texttt{free (r); 0} \qquad\qquad (\mathsf{T} = \mathbb{Z} \text{ or } \mathsf{T} = \bullet \to \bullet) \\
\underline{\texttt{destr}}^{\Sigma}_{\mathsf{X}}\,(\texttt{r}) & \triangleq & \overline{\texttt{const r}_\texttt{i} = \texttt{r[i + 1]; } \underline{\texttt{drop}}^{\Sigma}_{\mathsf{T}_\texttt{i}}\,(\texttt{r}_\texttt{i})\texttt{;}}^{i<n} \texttt{ free (r); 0} \quad (\Sigma \ni \mathsf{struct}\ \mathsf{X}\ \{\overline{\mathsf{s}_\mathsf{i} : \mathsf{T}_\mathsf{i}}^{i<n}\}) \\
\underline{\texttt{destr}}^{\Sigma}_{\mathsf{X}}\,(\texttt{r}) & \triangleq & \begin{array}{l}\texttt{if (r[1] = i) \{} \\ \quad \overline{\texttt{const r}_\texttt{i} = \texttt{r[2]; } \underline{\texttt{drop}}^{\Sigma}_{\mathsf{T}_\texttt{i}}\,(\texttt{r}_\texttt{i})\texttt{; free (r); 0}}^{i<n} \\ \texttt{\} else \{havoc\}}\end{array} \quad (\Sigma \ni \mathsf{enum}\ \mathsf{X}\ \{\overline{\mathsf{s}_\mathsf{i} : \mathsf{T}_\mathsf{i}}^{i<n}\})
\end{array}
$$

Fig. 4. Compiler macros.

Dup and Drop. Before moving to the elimination forms, we consider the structural rules dup and drop. Unsurprisingly, the compiler dups and drops in these cases, but in our target these are not atomic instructions—we must implement them first. In Fig. 4, they are defined as type-indexed macros, $\underline{\texttt{dup}}_{\mathsf{T}}$ and $\underline{\texttt{drop}}_{\mathsf{T}}$. For now, $\underline{\texttt{dup}}_{\mathsf{T}}$ simply increments its argument since all values are boxed. Likewise, $\underline{\texttt{drop}}_{\mathsf{T}}$ can always decrement since its argument will be boxed, but it must consider two cases. If the count is still non-zero, the reference is live and the new count is returned. Otherwise, the reference is dead and must be deallocated, which is done with the type-indexed $\underline{\texttt{destr}}_{\mathsf{T}}$ macro, which drops all references held by its argument and then frees it. Note that in the case of enums, one must case on the tag to find out the type of payload to drop, and it is undefined behavior if the tag is not declared.

Calling Convention. Now that we have $\underline{\texttt{dup}}_{\mathsf{T}}$ and $\underline{\texttt{drop}}_{\mathsf{T}}$, we can move on to the elimination forms. An integer operator loads the data from its operand references, boxes the result in a new reference, and—importantly—drops the operands. A function call loads the function pointer, drops the function reference, and then calls the function with its input references. A struct projection loads the component reference from the appropriate offset, dups it, and *then* drops the struct reference. It is essential to dup the component before dropping the struct, since dropping the struct first might deallocate the component, but duping it first ensures that it remains live. To case on an enum, we compare the tag against its position in declaration order, and it is undefined behavior if the tag is not declared. When a comparison succeeds, the payload reference is bound and dup'd, and the enum reference is dropped *last*.

3 A First ABI

Stepping back from our particular compiler, let us extrapolate some high-level design goals for our Quick ABI, which we will use as guiding principles in its specification. In practice, ABI proposals (e.g., for Rust [RFCs 2023], Swift [Apple 2017], or Python [Foundation 2023]) often begin with a list of objectives like ours, even if the exact aims are different (e.g., Rust strives for memory safety but not garbage freedom).

Safety. Every program must be *safe*, in the sense that it does not get stuck, though it may diverge.

Garbage Freedom. Every program must be *garbage-free*, in the sense that if it terminates with a value, then any newly allocated memory that remains must be associated with that value (in a manner specified by its type).

RCV Layout. Every *reference-counted value* (RCV) is represented in memory as a positive integer reference counter followed by data suitable for its type.

RCV Ownership. *Owning* a RCV means having exclusive rights to a single *share* of its reference count. While a RCV is owned, it must not be deallocated and its data must not be modified.

RCV Release. An owned RCV must be either be *moved* to another owner or released, and any releasing context must assume ownership of the underlying memory if the count reaches zero.

RCV Retain. Any RCV reachable from an owned RCV may be retained.

Calling Convention. The calling convention is that functions and operators take ownership of input RCVs (i.e., they are *moved in* [Rust 2023b] or stolen [Foundation 2023]).

3.1 Semantic Structures

Resources. Physical memories do not have enough structure to support the reasoning principles stipulated in our design goals. For one thing, memories are global, but we want to be able to reason about ownership locally. Also, memories cannot distinguish between a reference-counted block and any other block that happens to have the same layout. In order to reason about reference counting, we instrument memories (Fig. 6) with additional *ghost state* that tracks whether a location is a unique pointer to "plain old data" (unq) or a shared reference counter (shr). In the latter case, we also track its *local share* of the reference count and what sub-resource it holds a share of, which we call its *object*. This means that, while physical memories are linear, these logical resources form graphs, where the root node contains all the root locations bound to variables and the reachable nodes correspond to reachable objects. For example, consider the following code snippet.

$$\begin{array}{l}\mathsf{struct\ Point}\ \{\mathsf{x} : \mathbb{Z}, \mathsf{y} : \mathbb{Z}\} \\ \mathsf{let\ p} = \mathsf{Point}\ \{\mathsf{x} : 20, \mathsf{y} : 24\};\ \boxed{5a};\ \boxed{5b};\ \boxed{5c};\ \mathsf{let\ \_} = \{\mathsf{let\ x} = \mathsf{p.x};\ \boxed{5d};\ e_{px}\};\ e_p \\ \quad (\text{where } e_{px} \text{ uses } \mathsf{p} \text{ and } \mathsf{x}, \text{ and } e_p \text{ uses } \mathsf{p})\end{array}$$

In the target, we cannot create the Point p atomically. Instead, we must allocate and initialize its memory, stored at some location ℓ, leading to Fig. 5a. However, this view does not capture the logical structure intended for Point, so we shift our perspective and analyze the resource in Fig. 5b instead. Under this view, plain old data (like 20 and 24) is tagged with a unq cell, which means that its location will only appear once in the resource graph. On the other hand, a reference counter is tagged with a share of the count in a shr cell, and it points to its object—the sub-resource it is managing. Since p is used in multiple sub-expressions, the type system (and therefore, the compiler) will insert a dup, and then split the context to type-check e_{px} and e_p independently. We want to mirror this reasoning principle with resources: after incrementing ℓ's count to 2 at the root (which corresponds to dup'ing p), we can *decompose* the resource into two identical resources, each with 1 share of ℓ's counter (Fig. 5c). Notice that, in accordance with RCV Ownership, this decomposition (or its converse, the composition $\bullet$) does not change the contents of any reachable *objects*, only the root. In particular, there must be *agreement* on the reachable objects, but composition does not descend down the graph and compose the reachable objects themselves, which ensures that the reference counts are not inflated and that unique cells do not conflict. Having split the resource,

one half may pass to p.x, which dups the x component's reference counter at ℓ_x, a reachable but non-root location. RCV Retain permits this, and doing so confers one share of the count to the root (Fig. 5d), but leaves the original share intact, as required by RCV Ownership.

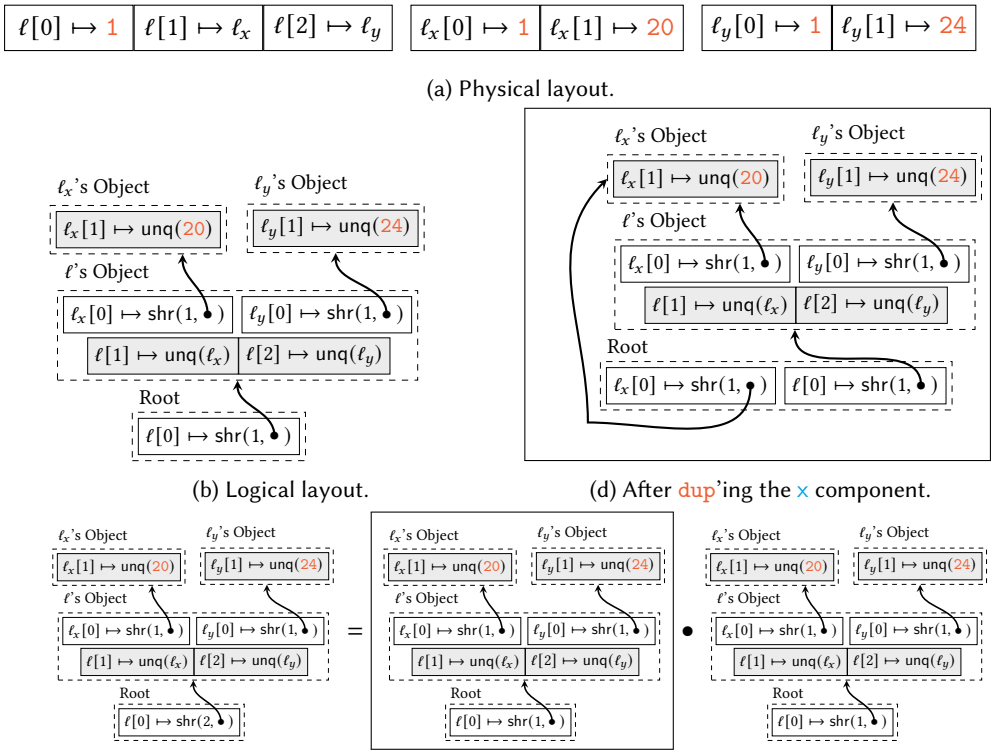


Fig. 5. Manipulating a struct Point {x : ℤ, y : ℤ} at ℓ.

The formalization of this visual intuition is given in Fig. 6. A resource $\rho \in \mathsf{Res}$ is a tree with $\mathsf{unq}(w)$ leaves and $\mathsf{shr}(n+1, \rho)$ nodes, but as motivated in Fig. 5d, we will restrict ourselves to the resources that we can view logically as DAGs. Moreover, we must ensure that unq locations really are unique within the DAG, and that there is internal agreement on which shr locations manage which objects. These three properties are captured by the *validity* predicate $\checkmark\ \rho$, which ensures that any two reachable nodes in the tree—either the root ρ or a reachable object in $\mathsf{objs}(\rho)$—are either *shallowly compatible* ($\sharp_{\mathsf{sh}}$) or else should be collapsible into a single node (i.e., they are objects associated with the same shr location). As the name suggests, shallow compatibility only compares the root node of two resources, where it enforces disjointness on unique locations and agreement on shared locations. The objects map $\mathsf{objs}(\rho)$ associates reachable shr locations with their objects and is defined using the reachability relation $\rightarrowtriangle$, which includes "jumps" from shr nodes to their associated objects and decompositions of resources into smaller sub-resources.

Shallow compatibility is also a prerequisite for the composition operator ($\bullet$) from Fig. 5c. As we saw there, composition only affects the root node of the composed resources: it unions disjoint entries and sums the reference counts of overlapping, shared entries. A pair of shallowly compatible resources whose composition is valid is said to be simply *compatible*.

$$
\begin{array}{lcl}
\rho \in \mathsf{Res} & \triangleq & \mathsf{Loc}_{\mathbb{N}^+} \xrightarrow{\text{fin}} \mathsf{unq}(\mathtt{Word}) + \mathsf{shr}(\mathsf{count} : \mathbb{N}^+, \mathsf{obj} : \mathsf{Res}) \\
\checkmark\, \rho & \triangleq & \forall\, (\ell_1, \rho_1), (\ell_2, \rho_2) \in \mathsf{objs}(\rho).\ \rho \sharp_{\mathsf{sh}} \rho_1 \wedge \big((\ell_1 \neq \ell_2 \wedge \rho_1 \sharp_{\mathsf{sh}} \rho_2) \vee (\ell_1 = \ell_2 \wedge \rho_1 = \rho_2)\big) \\
\mathsf{objs}(\rho) & \triangleq & \big[(\ell, \rho_\ell) \mid \rho \twoheadrightarrow [\ell \mapsto \mathsf{shr}(-, \rho_\ell)]\big] \\
\rho_1 \sharp_{\mathsf{sh}} \rho_2 & \triangleq & \forall\, \ell \in \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2), \exists\, \rho.\ \rho_1(\ell) = \mathsf{shr}(-, \rho) \wedge \rho_2(\ell) = \mathsf{shr}(-, \rho) \\
\rho_1 \bullet \rho_2 & \triangleq & \begin{cases} [\ell \mapsto \rho_1(\ell) \mid \ell \in \mathrm{dom}(\rho_1) \setminus \mathrm{dom}(\rho_2)] \\ \uplus\, [\ell \mapsto \rho_2(\ell) \mid \ell \in \mathrm{dom}(\rho_2) \setminus \mathrm{dom}(\rho_1)] & \text{if } \rho_1 \sharp_{\mathsf{sh}} \rho_2 \\ \uplus\, [\ell \mapsto \mathsf{shr}(n_1 + n_2, \rho) \mid \rho_1(\ell) = \mathsf{shr}(n_1, \rho) \wedge \rho_2(\ell) = \mathsf{shr}(n_2, \rho)] \end{cases} \\
\rho_1 \sharp \rho_2 & \triangleq & \rho_1 \sharp_{\mathsf{sh}} \rho_2 \wedge \checkmark\, (\rho_1 \bullet \rho_2) \\
\mathsf{erase}(\rho) & \triangleq & \begin{cases} [\ell \mapsto \mathtt{w} \mid \rho_{\mathsf{flat}}(\ell) = \mathsf{unq}(\mathtt{w})] \\ \uplus\, [\ell \mapsto \mathtt{n} \mid \rho_{\mathsf{flat}}(\ell) = \mathsf{shr}(n, -)] \end{cases} \quad \text{if } \checkmark\, \rho \text{ and } \rho_{\mathsf{flat}} = \rho \bullet \left(\bullet_{(\ell, \rho_\ell) \in \mathsf{objs}(\rho)} \rho_\ell\right)
\end{array}
$$

$\boxed{\rho \twoheadrightarrow \rho'}$ ρ can reach ρ' via jumps and decompositions.

$$
\begin{array}{lll}
(\twoheadrightarrow\text{-JUMP}) & (\twoheadrightarrow\text{-SUB}) & \dfrac{\rho_1 \twoheadrightarrow \rho_2 \quad \rho_2 \twoheadrightarrow \rho_3}{\rho_1 \twoheadrightarrow \rho_3}\ (\twoheadrightarrow\text{-TRANS}) \\
\ell \mapsto \mathsf{shr}(-, \rho) \twoheadrightarrow \rho & \rho_1 \bullet \rho_2 \twoheadrightarrow \rho_1 &
\end{array}
$$

Fig. 6. Memories instrumented as logical resources.

Next, the *erasure* of resources into physical memories proceeds in two steps. First, the root is composed with all of its reachable objects, which effectively moves all unique locations to the root and sums up all the shares for each reference counter. Visually, working with a DAG is crucial for this step: a naive recursive descent over the *tree* interpretation might duplicate counts from identical objects, but with the DAG interpretation we can simply talk about the *set* of reachable objects, which naturally has no duplicates. Note that this flattening composition is always defined for valid resources, by design. After moving all objects to the root, these root cells are erased into plain values—a unique cell keeps its value, and a shared cell only keeps its reference count, since, having been moved during the first step, its object already exists elsewhere in the root.

Separation logic. Having defined a suitable resource and composition operator, we can now overlay a separation logic [Appel 2014; Dinsdale-Young et al. 2013; Jung et al. 2018; Ley-Wild and Nanevski 2013] on top, which we will use to specify and reason about the ABI. As defined in Fig. 7, a proposition of separation logic is a predicate on resources. The quintessential separation logic connective, the separation conjunction $P \star Q$, holds for all resources that can be decomposed into a resource satisfying P and a resource satisfying Q. We use all of the usual intuitionistic and *linear* separation logic connectives; i.e., resources may not be freely discarded.

In order to reason about recursive functions and recursive types, our separation logic uses *step-indexing* [Ahmed 2006, 2004; Appel and McAllester 2001], which is a technique that breaks the circularity of inherently circular language features by approximating a proposition using the number of program steps on which it is true. Just as separation logic abstracts away particular resources, the *later modality* [Appel et al. 2007; Jung et al. 2018; Nakano 2000] $\triangleright P$ abstracts away particular step-indices—it holds whenever P holds after taking one more step (assuming there are steps left to take). Importantly, the later modality supports a general induction principle and allows one to construct well-founded recursive definitions so long as recursive occurrences occur below a later.

To tie step-indices to physical program steps, we use the *weakest precondition modality* $wp_{\mathtt{F}}\, (\mathtt{e})\, \{\hat{Q}\}$, which characterizes when $\mathtt{e}$ is safe to run in order to satisfy the post condition $\hat{Q}$, using top-level functions in $\mathtt{F}$. The first part of the weakest precondition decides which *physical* configuration to run with. Physical configurations require block sizes, so we add the block sizes as

$$
\begin{array}{lll}
P, Q, R \in \mathit{Prd} & \triangleq & \{P : \mathsf{Wld} \to \mathsf{Res} \to \mathbb{P} \mid \forall\, \rho, \omega \sqsubseteq \omega^+.\ P(\omega, \rho) \Rightarrow P(\omega^+, \rho)\} \\
\hat{P}, \hat{Q}, \hat{R} \in \mathit{Prd}\,(X) & \triangleq & X \to \mathit{Prd} \\
\omega \in \mathsf{Wld} & \triangleq & \langle \mathsf{step} : \mathbb{N}, \mathsf{sizes} : \mathsf{Sizes} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\omega_1 \sqsubseteq \omega_2 & \triangleq & \omega_1.\mathsf{step} \geq \omega_2.\mathsf{step} \wedge \omega_1.\mathsf{sizes} \subseteq \omega_2.\mathsf{sizes} \\
\ell \mapsto \mathtt{w} \quad (\omega, \rho) & \triangleq & \rho = [\ell \mapsto \mathsf{unq}(\mathtt{w})] \\
P \star Q \quad (\omega, \rho) & \triangleq & \exists\, \rho_p, \rho_q.\ \rho = \rho_p \bullet \rho_q \wedge P(\omega, \rho_p) \wedge Q(\omega, \rho_q) \\
P \mathrel{-\!\!\star} Q \quad (\omega, \rho) & \triangleq & \forall\, \omega^+ \sqsupseteq \omega, \rho_p \perp \rho, \rho_q.\ \rho \bullet \rho_p = \rho_q \Rightarrow P(\omega^+, \rho_p) \Rightarrow Q(\omega^+, \rho_q) \\
\triangleright P \quad (\omega, \rho) & \triangleq & \omega.\mathsf{step} = 0 \vee (\omega.\mathsf{step} > 0 \wedge P(\omega[\mathsf{step} := \omega.\mathsf{step} - 1], \rho)) \\
!\, P \quad (\omega, \rho) & \triangleq & \rho = \varnothing \wedge P(\omega, \varnothing) \\
\ulcorner P \urcorner \quad (\omega, \rho) & \triangleq & \rho = \varnothing \wedge P \\
\mathit{wp}_{\mathtt{F}}\,(\mathtt{e})\,\{\hat{Q}\} \quad (\omega, \rho) & \triangleq & \left\{ \begin{array}{l} \forall\, \omega^+ \sqsupseteq \omega, \rho_f \perp \rho, k < \omega^+.\mathsf{step}, \psi', \mu', \mathtt{e}', \omega'. \\ \quad \mathtt{F} \vdash (\omega^+.\mathsf{sizes}, \mathsf{erase}(\rho \bullet \rho_f), \mathtt{e}) \to^k (\psi', \mu', \mathtt{e}') \twoheadrightarrow \\ \quad \Rightarrow \omega' = \langle \mathsf{step} : \omega^+.\mathsf{step} - k, \mathsf{sizes} : \psi' \rangle \\ \quad \Rightarrow \exists\, \rho' \perp \rho_f.\ \mathsf{erase}(\rho' \bullet \rho_f) = \mu' \wedge \mathtt{e}' \in \mathtt{Word} \wedge \hat{Q}(\mathtt{e}')(\omega', \rho') \end{array} \right. \\
\{P\}\ \mathtt{e}\ \{\hat{Q}\}_{\mathtt{F}} & \triangleq & !\left(P \mathrel{-\!\!\star} \mathit{wp}_{\mathtt{F}}\,(\mathtt{e})\,\{\hat{Q}\}\right) \\
@_\ell\, P \quad (\omega, \rho) & \triangleq & \exists\, \rho_p.\ \rho = [\ell \mapsto \mathsf{shr}(1, \rho_p)] \wedge P(\omega, \rho_p) \\
\Diamond P \quad (\omega, \rho) & \triangleq & \exists\, \rho_p.\ \rho \rightharpoonup \rho_p \wedge P(\omega, \rho_p) \\
\mathit{size}\,(\ell,\ n) \quad (\omega, \rho) & \triangleq & \rho = \varnothing \wedge \exists\, b.\ \ell = \langle b, 0 \rangle \wedge \omega.\mathsf{sizes}(b) = n
\end{array}
$$

Fig. 7. Separation logic predicates (excerpts), styled with *emerald italic*.

a parameter to our logical predicates. Predicates must be *monotone* with respect to both the step index and the block sizes, and both are shared rather than split by separating conjunction, so we group them together into a *Kripke world* whose accessibility relation admits smaller step-indices and larger block size maps. Now, we can run $\mathtt{e}$ with the block sizes and the erasure of its resource composed with an arbitrary *framing* resource, which prevents $\mathtt{e}$ from acting beyond its means. If the program terminates within the given step-index bound, the resulting configuration must satisfy three conditions: the framing resource must be preserved, the program must be a value, and the postcondition must hold for that value given the new logical configuration (which uses a potentially *smaller* step-index). Similar to Iris [Jung et al. 2018], the more familiar Hoare triple $\{P\}\ \mathtt{e}\ \{\hat{Q}\}_{\mathtt{F}}$ is defined in terms of *wp* and the *unrestricted modality* $!\, P$, which, in our case, is true when P holds and the resource is empty.[1]

Extensions. Whereas unique locations are characterized by the familiar points-to connective $\ell \mapsto \mathtt{w}$ from separation logic, shared locations are characterized by the *jump modality* $@_\ell\, P$, where ℓ is the location holding the reference count and P characterizes its object. The name and notation for the modality is inspired by *hybrid logic* [Braüner and de Paiva 2006], in which the jump modality relativizes the truth of a proposition to a named Kripke world. As in other adaptations of this concept to programming languages [Balzer et al. 2019; Caires et al. 2019], the set of possible jumps is restricted by an *accessibility relation*, which, in our case, is the set of reference-counted objects reachable from the root using one edge. Returning to the earlier visualization of resources as DAGs, the jump modality lets us "jump over" an ℓ-edge in the graph to reach a resource satisfying P, as in the example in Fig. 8.

[1] Iris' Hoare triple is defined in terms of their persistence modality $\Box P$, which is similar in spirit to our unrestricted modality but it does not exactly require the resource to be empty.

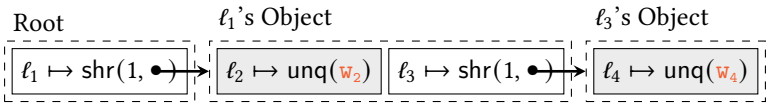


Fig. 8. Resource that satisfies $@_{\ell_1}\ (\ell_2 \mapsto \mathtt{w}_2 \star @_{\ell_3}\ (\ell_4 \mapsto \mathtt{w}_4))$, as well as $\diamond\ (\ell_2 \mapsto \mathtt{w}_2)$ and $\diamond\ (\ell_4 \mapsto \mathtt{w}_4)$ and $\diamond\ (\exists\, \ell,\, \mathtt{w}.\ \ell \mapsto \mathtt{w})$.

The rules for manipulating jumps are given in Fig. 9. Each occurrence of $@_\ell\, P$ confers a *single share* of the reference counter at ℓ, so that, e.g., one can HT-@-DUP ℓ to move from $@_\ell\, P$ to $@_\ell\, P \star @_\ell\, P$ and vice versa with HT-@-DROP. In accordance with RCV RELEASE, dropping may return ownership of ℓ and the underlying object satisfying P if the count hits zero, so the logical continuation in HT-@-DROP has a disjunct for this case. To create a jump with HT-@-SHARE, the location of the new reference counter must point to 1 and its new object must be owned. Note that creating a jump is a logical view shift—it does not need to take a physical step, since the memory is already in the correct shape physically.

(HT-@-DUP)
$$\frac{P \vDash \diamond\, @_\ell\, Q}{\{P\} +\!+\ell\, \{\mathtt{n}.\ \ulcorner n > 1 \urcorner \star P \star @_\ell\, Q\}}$$

(HT-@-DROP)
$$\{@_\ell\, P\} -\!-\ell\, \{\mathtt{n}.\ \ulcorner n > 0 \urcorner \vee (\ulcorner n = 0 \urcorner \star \ell \mapsto 0 \star P)\}$$

(HT-@-SHARE)
$$\frac{\{@_\ell\, P \star Q\}\ \mathtt{e}\ \{\hat{R}\}}{\{\ell \mapsto 1 \star P \star Q\}\ \mathtt{e}\ \{\hat{R}\}}$$

(HT-MALLOC)
$$\{emp\}\ \mathtt{malloc}\,(\mathtt{n})\ \{\ell.\ \bigstar_{i<n} \ell + i \mapsto \text{☣} \star size\,(\ell,\, n)\}$$

(HT-FREE)
$$\frac{\{P\}\ \mathtt{e}\ \{\hat{Q}\}}{\{P \star \bigstar_{i<n} \ell + i \mapsto \mathtt{w}_i \star size\,(\ell,\, n)\}\ \mathtt{free}\,(\ell)\,;\ \mathtt{e}\ \{\hat{Q}\}}$$

(HT-LOAD)
$$\frac{P \vDash \diamond\, (\ell \mapsto \mathtt{w})}{\{P\} *\ell\, \{\mathtt{w}'.\ulcorner \mathtt{w} = \mathtt{w}' \urcorner \star P\}}$$

($\diamond$-JUMP) $@_\ell\, P \vDash \diamond P$

($\diamond$-REFL) $P \vDash \diamond P$

($\diamond$-SUB) $\diamond\ (P \star Q) \vDash \diamond P$

($\diamond$-TRANS) $\diamond\diamond P \vDash \diamond P$

(@-MONO)
$$\frac{P \vDash Q}{@_\ell\, P \vDash @_\ell\, Q}$$

($\diamond$-MONO)
$$\frac{P \vDash Q}{\diamond P \vDash \diamond Q}$$

Fig. 9. An excerpt of valid Hoare triples and entailments.

The *reachability modality* $\diamond P$ internalizes the reachability relation $\multimap\!\!\bullet$, characterizing resources that can reach some sub-resource satisfying P anywhere in the graph ($\diamond$-TRANS), using an arbitrary number of jumps ($\diamond$-JUMP) and decompositions ($\diamond$-REFL, $\diamond$-SUB), as demonstrated in Fig. 8. Reachability is only used in non-destructive operations like reading (HT-LOAD) or dup'ing (HT-@-DUP); it may not be used to write, drop, or deallocate. It is a monad-style modality ($\diamond$-REFL and $\diamond$-TRANS may be used to derive a familiar monadic bind rule), which prevents memory from being leaked by $\diamond$-SUB. As is characteristic of modal operators, both @ and $\diamond$ are monotone with respect to entailment (@-MONO, $\diamond$-MONO).

The *size predicate* $size\,(\ell,\, n)$ says that the block size map has an entry rooted at ℓ (i.e., its offset is zero) with size n (as introduced in HT-MALLOC). It is used to safely deallocate entire blocks at once: in order to $\mathtt{free}\,(\ell)$, one must have $size\,(\ell,\, n)$ and own the n locations offset from ℓ (as in HT-FREE). Since *size* only talks about the block sizes and not the resource, it is an unrestricted predicate and may be freely duplicated and discarded.

To get a sense for how these predicates and proof rules are used in context, Fig. 10 shows how they can be used to verify the manipulation of integer references, as done in our compiler in Figs. 3 and 4.

Creating an integer reference

$\{emp\}$
$\texttt{const r = malloc (2); r[0] = 1; r[1] = n; r}$
$\{\ell.\ \ell \mapsto 1 \star \ell + 1 \mapsto \texttt{n} \star size\,(\ell, 2)\}$ using HT-MALLOC
$\{\ell.\ @_\ell\ (\ell + 1 \mapsto \texttt{n}) \star size\,(\ell, 2)\}$ using HT-@-SHARE

Duplicating an integer reference

$\{@_\ell\ (\ell + 1 \mapsto \texttt{n})\}$
$\texttt{++}\ell$
$\{\texttt{n}'.\ \ulcorner n' > 1 \urcorner \star @_\ell\ (\ell + 1 \mapsto \texttt{n}) \star @_\ell\ (\ell + 1 \mapsto \texttt{n})\}$ using HT-@-DUP

Dropping an integer reference

$\{@_\ell\ (\ell + 1 \mapsto \texttt{n}) \star size\,(\ell, 2)\}$
$\texttt{const c} = \texttt{--}\ell\texttt{;}$
$\{\ulcorner c > 0 \urcorner \vee (\ulcorner c = 0 \urcorner \star \ell \mapsto 0 \star \ell + 1 \mapsto \texttt{n}) \star size\,(\ell, 2)\}$ using HT-@-DROP
$\texttt{if (c) \{}$
$\quad \{\ulcorner c > 0 \urcorner \star size\,(\ell, 2)\}\ \texttt{c}\ \{\texttt{n}'.\ \ulcorner n' \geq 0 \urcorner\}$
$\texttt{\} else \{}$
$\quad \{\ulcorner c = 0 \urcorner \star \ell \mapsto 0 \star \ell + 1 \mapsto \texttt{n} \star size\,(\ell, 2)\}\ \texttt{free (}\ell\texttt{); 0}\ \{\texttt{n}'.\ \ulcorner n' \geq 0 \urcorner\}$ using HT-FREE
$\texttt{\}}$
$\{\texttt{n}'.\ \ulcorner n' \geq 0 \urcorner\}$

Fig. 10. Proof sketches for manipulating integer references, eliding standard rules.

Note that while we use inference notation, as in prior work [Appel et al. 2007; Dreyer et al. 2011], we do not fix a syntax and set of inference rules—instead, we work with predicates directly in the meta-language, and the "inference rules" are actually just lemmas about their relationship as mathematical objects.

3.2 Preliminary Definition

We now have all the structure needed to define a preliminary semantic ABI for Quick, shown in Fig. 11. The ABI must characterize when a target program *realizes* (or behaves like) a source type, and it must do so in a way that respects the *logical structure* of source types (e.g., composing a $\mathsf{T}_1 \to \mathsf{T}_2$ program with a T_1 program should result in a T_2 program). The *expression predicate* $\mathcal{E}[\![\mathsf{T}]\!](\texttt{e})$ uses the weakest precondition to characterize when a closed expression e realizes a type T by using the *value predicate* $\mathcal{V}[\![\mathsf{T}]\!]$ as the postcondition. Using the weakest precondition immediately achieves our SAFETY goal exactly because the definition of *wp* was designed to require safety.

For now, values of all types are boxed, so the value predicate simply requires the value to be a non-null location and passes it to the *reference predicate* $\mathcal{R}[\![\mathsf{T}]\!]$. In accordance with RCV LAYOUT, $\mathcal{R}[\![\mathsf{T}]\!](\ell)$ characterizes ℓ as a RCV for T if there is jump over ℓ to a suitable *object* $\mathcal{O}[\![\mathsf{T}]\!](\ell + 1)$ in adjacent memory. By using a jump, the desiderata RCV OWNERSHIP, RCV RELEASE, and RCV RETAIN will follow from HT-@-DUP and HT-@-DROP. The indirection between $\mathcal{V}[\![\mathsf{T}]\!]$ and $\mathcal{O}[\![\mathsf{T}]\!]$ may seem superfluous now, but it will be useful when we look at ABI variations in § 4.

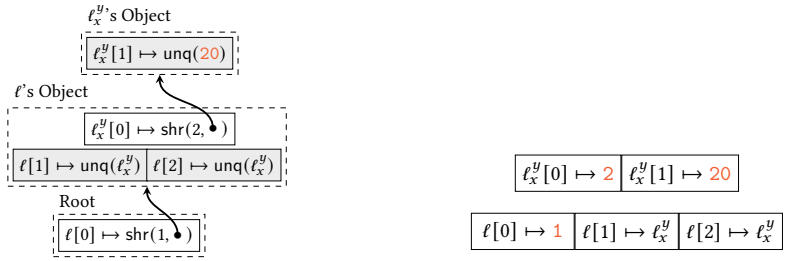
Whereas the reference predicate $\mathcal{R}[\![\mathsf{T}]\!]$ is uniform for all T, the object predicate $\mathcal{O}[\![\mathsf{T}]\!]$ varies by T, making it the heart of the ABI. The object relation is responsible for specifying size and layout,

$$
\begin{array}{lcl}
\mathcal{E}[\![\mathsf{T}]\!]_{\mathsf{F}}(\mathtt{e}) & \triangleq & wp_{\mathsf{F}}\,(\mathtt{e})\,\{\mathcal{V}[\![\mathsf{T}]\!]_{\mathsf{F}}\} \\
\mathcal{V}[\![\mathsf{T}]\!]_{\mathsf{F}}(\mathtt{w}) & \triangleq & \ulcorner \mathtt{w} \in \mathsf{Loc} \setminus \mathtt{null} \urcorner \star \mathcal{R}[\![\mathsf{T}]\!]_{\mathsf{F}}(\mathtt{w}) \\
\mathcal{R}[\![\mathsf{T}]\!]_{\mathsf{F}}(\ell) & \triangleq & @_\ell\, O[\![\mathsf{T}]\!]_{\mathsf{F}}(\ell+1) \\
O[\![\mathbb{Z}]\!]_{\mathsf{F}}(\ell+1) & \triangleq & \exists\, \mathtt{n}.\ \ell+1 \mapsto \mathtt{n} \star \mathit{size}\,(\ell,\,2) \\
O[\![\mathsf{X}]\!]_{\mathsf{F}}(\ell+1) & \triangleq & \begin{cases} \bigstar_{i<n} \exists\, \mathtt{w}_i.\ \ell+1+i \mapsto \mathtt{w}_i \star \triangleright\, \mathcal{V}[\![\mathsf{T}_i]\!]_{\mathsf{F}}(\mathtt{w}_i) \star \mathit{size}\,(\ell,\,n+1) \\ \quad \text{if } \Sigma \ni \mathtt{struct}\,\mathsf{X}\,\{\overline{\mathtt{s}_i : \mathsf{T}_i}^{\,i<n}\} \\ \bigvee_{i<n} \ell+1 \mapsto \mathtt{i} \star \exists\, \mathtt{w}_i.\ \ell+2 \mapsto \mathtt{w}_i \star \triangleright\, \mathcal{V}[\![\mathsf{T}_i]\!]_{\mathsf{F}}(\mathtt{w}_i) \star \mathit{size}\,(\ell,\,3) \\ \quad \text{if } \Sigma \ni \mathtt{enum}\,\mathsf{X}\,\{\overline{\mathtt{s}_i : \mathsf{T}_i}^{\,i<n}\} \end{cases} \\
O\left[\!\left[\overline{\mathsf{T}_i}^{\,i<n} \to \mathsf{T}\right]\!\right]_{\mathsf{F}}(\ell+1) & \triangleq & \exists\, \mathtt{call}.\ \ell+1 \mapsto \langle \mathtt{call} \rangle_{\mathsf{F}} \star \mathit{size}\,(\ell,\,2) \\
 & & \quad \star\, \forall\, \overline{\mathtt{w}_i}^{\,i<n}.\ \{\bigstar_{i<n} \mathcal{V}[\![\mathsf{T}_i]\!]_{\mathsf{F}}(\mathtt{w}_i)\}\,\langle \mathtt{call} \rangle_{\mathsf{F}}\,(\overline{\mathtt{w}_i}^{\,i<n})\ \{\mathcal{V}[\![\mathsf{T}]\!]_{\mathsf{F}}\}_{\mathsf{F}} \\
C[\![\Gamma]\!]_{\mathsf{F}}(\gamma) & \triangleq & \bigstar_{\mathtt{x}:\mathsf{T} \in \Gamma} \mathcal{V}[\![\mathsf{T}]\!]_{\mathsf{F}}(\gamma(\mathtt{x})) \\
\Gamma \vDash_{\mathsf{F}} \mathtt{e} : \mathsf{T} & \triangleq & \forall\, \mathsf{F}' \supseteq \mathsf{F}, \gamma.\ C[\![\Gamma]\!]_{\mathsf{F}'}(\gamma) \vDash \mathcal{E}[\![\mathsf{T}]\!]_{\mathsf{F}'}(\mathtt{e}[\gamma])
\end{array}
$$

Fig. 11. Preliminary semantic ABI, parameterized over a signature Σ.

but also for expressing semantic requirements on the data, as we will see. The simplest object is an integer, which uniquely holds an integer at the first offset and, importantly, *nothing else*. This means that if a program runs to a location ℓ satisfying $\mathcal{R}[\![\mathsf{T}]\!](\ell)$, we can be sure it does not hold any superfluous memory, in accordance with Garbage Freedom.

Objects for nominal data types require consulting the signature Σ for their definition. For a struct, its fields are laid out in declaration order. More interestingly, the semantic ABI captures *ownership* in addition to layout, so a struct object holds more memory *logically* than it does *physically*. We already saw an example of this in Fig. 5b—physically, a Point object simply holds two pointers, but logically, it also has shared ownership of its component references. Perhaps more subtly, the ABI also validates a Point whose components alias, in which case the Point owns *two* shares of the component's count, as shown in the example in Fig. 12. In the semantic ABI, ownership of fields is conferred by the recursive use of the value predicate on a struct's members, which must be guarded by the later modality since data types can be recursive.



(a) Logical view. (b) Physical view.

Fig. 12. Logical and physical representations of struct Point $\{\mathtt{x} : \mathbb{Z}, \mathtt{y} : \mathbb{Z}\}$ whose fields alias.

An enum is laid out with its tag in the second position and its payload in the third position. The tag is simply an index into the list of selectors in declaration order. As with structs, its logical memory footprint may be larger because it owns the RCV associated with its payload. Just as the

tag determines what physical data will appear in the payload, it also determines what logical resources will be owned by the payload. Whereas the size of a struct depends on the number of fields, every enum has a size of 3, and the use of a disjunction ($\vee$) instead of a separating conjunction ($\star$) ensures that only one variant is exhibited at a time.

Physically, a function object contains a function pointer, but logically it comes with a proof that the function obeys the Calling Convention, which is specified using a Hoare triple. Recall that Hoare triples are unrestricted, so the calling convention only talks about *behavior*; it does not own any resource of its own. Notice that the convention requires callers to move in ownership of the inputs, and it moves out ownership of the output, only.

Finally, the top-level judgment $\Gamma \vDash_{\mathtt{F}} \mathtt{e} : \mathtt{T}$ characterizes when an open program behaves like a $\mathtt{T}$ under Γ using top-level functions in $\mathtt{F}$. It quantifies over larger function tables $\mathtt{F}' \supseteq \mathtt{F}$, since the context—which can be thought to approximate a library—might require additional functions, but these must not conflict with the functions $\mathtt{e}$ expects to be available. It also quantifies over semantic substitutions $\mathcal{C}[\![\Gamma]\!](\gamma)$ that confer ownership of a $\mathcal{V}[\![\mathtt{T}]\!](\gamma(\mathtt{x}))$ for every $\mathtt{x} : \mathtt{T}$ in Γ. Recall that bindings can appear multiple times in a typing context, so, e.g., $\mathcal{C}[\![\mathtt{x} : \mathtt{T}, \mathtt{x} : \mathtt{T}]\!](\gamma) = \mathcal{V}[\![\mathtt{T}]\!](\gamma(\mathtt{x})) \star \mathcal{V}[\![\mathtt{T}]\!](\gamma(\mathtt{x}))$. As a technical subtlety, this means that, unlike in other logical relations for substructural types (e.g., [Ahmed et al. 2007]), splitting contexts in the source does not correspond to splitting *substitutions* in the model, only resources.

Just as for standard logical relations, we prove that ABI compliance is adequate to establish Safety and Garbage Freedom for whole programs of base type, $\mathbb{Z}$. For this version of the ABI, where even integers are reference counted, termination requires the production of an integer reference, which refers to the only block remaining in memory.

Theorem 3.1 (ABI Adequacy). *If* $\Sigma; \varnothing \vDash_{\mathtt{F}} \mathtt{e} : \mathbb{Z}$ *and* $\mathtt{F} \vdash (\varnothing, \varnothing, \mathtt{e}) \longrightarrow^* (\psi', \mu', \mathtt{e}') \not\rightarrow$, *then there is* $\mathtt{n}, \ell$ *such that* $\mu' = [\ell \mapsto 1, \ell + 1 \mapsto \mathtt{n}]$ *and* $\mathtt{e}' = \ell$.

3.3 Compiler Compliance

Now we can show that our candidate Quick compiler from § 2.3 is *compliant* with the semantic ABI. When an ABI is specified in prose, compliance can be difficult to establish, which can lead to subtle bugs (e.g., [Álvarez 2018; Desires 2023]). Here, since the semantic ABI is really just a logical relation, compiler compliance corresponds to what is usually called the *fundamental property*, which says that (the compilation of) any syntactically well-typed term is also semantically well-typed.

Lemma 3.2 (Compliant Compilation). *If* $\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{T} \rightsquigarrow \mathtt{e} \dashv \mathtt{F}$, *then* $\Sigma; \Gamma \vDash_{\mathtt{F}} \mathtt{e} : \mathtt{T}$.

As usual, this is proved by induction on the typing derivation—or in this case, the compilation derivation—and the proof is factored out into so-called "compatibility lemmas", one per compilation rule. Adequacy of the type system can then be established as a corollary of Compliant Compilation and ABI Adequacy.

Theorem 3.3 (Compiler Adequacy). *If* $\Sigma; \varnothing \vdash \mathtt{e} : \mathbb{Z} \rightsquigarrow \mathtt{e} \dashv \mathtt{F}$, *and* $\mathtt{F} \vdash (\varnothing, \varnothing, \mathtt{e}) \longrightarrow^* (\psi', \mu', \mathtt{e}') \not\rightarrow$, *then there is* $\mathtt{n}, \ell$ *such that* $\mu' = [\ell \mapsto 1, \ell + 1 \mapsto \mathtt{n}]$ *and* $\mathtt{e}' = \ell$.

A major benefit of ABI stabilization is that it facilitates cross-compiler linking.

Lemma 3.4 (Cross-Compiler Linking). *For any two compliant compilers* $\rightsquigarrow_1$ *and* $\rightsquigarrow_2$, *if* $\Sigma; \Gamma_1 \vdash \mathtt{e}_1 : \mathtt{T}_1 \rightsquigarrow_1 \mathtt{e}_1 \dashv \mathtt{F}_1$ *and* $\Sigma; \Gamma_2, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{e}_2 : \mathtt{T}_2 \rightsquigarrow_2 \mathtt{e}_2 \dashv \mathtt{F}_2$, *then* $\Sigma; \Gamma_1, \Gamma_2 \vDash_{\mathtt{F}_1, \mathtt{F}_2} \mathtt{const\ x} = \mathtt{e}_1; \mathtt{e}_2 : \mathtt{T}_2$.

In fact, the compilers themselves are not so important—all that matters is that we have ABI-compatible target programs, no matter how exactly they are produced. Indeed, like any *logical* relation, the semantic ABI satisfies a pleasant substitution-like property.

Lemma 3.5 (Safe Linking). *If* $\Sigma; \Gamma_1 \vDash_{F_1} e_1 : T_1$ *and* $\Sigma; \Gamma_2, x : T_1 \vDash_{F_2} e_2 : T_2$, *then* $\Sigma; \Gamma_1, \Gamma_2 \vDash_{F_1,F_2} \mathtt{const}\ x = e_1;\ e_2 : T_2$.

Cross-Compiler Linking easily follows from this more general property. This property also allows one to link compliantly compiled code with compliant libraries implemented directly in the target, or even compliantly compiled code from a different source language entirely. In the latter case, the other language would typically provide some compiler directive, like #repr in Rust or foreign in Haskell, in order to target the ABI of our language.

4 Variations on the ABI

4.1 Calling Convention

Caller vs. callee retain. As seen in Fig. 11, a Hoare triple is a natural mechanism for describing a calling convention. The convention we have seen so far takes ownership of (or *steals* [Foundation 2023]) inputs, which means that the caller is responsible for dup'ing any inputs they wish to retain. It is just as easy to express the convention that returns ownership of (or *borrows*) inputs, in which case the callee is responsible for dup'ing inputs. For a function type $\overline{T_i}^{i<n} \to T$, this convention can be expressed as follows.

$$\forall\ \overline{w_i}^{i<n}.\ \{\bigstar_{i<n} \mathcal{V}[\![T_i]\!]_F(w_i)\}\ \langle \mathtt{call} \rangle_F\ (\overline{w_i}^{i<n})\ \{w.\ \bigstar_{i<n} \mathcal{V}[\![T_i]\!]_F(w_i) \star \mathcal{V}[\![T]\!]_F(w)\}_F$$

As with low-level calling conventions, one could fine-tune the calling convention to behave differently on different argument types. Additionally, some languages have explicit program syntax, either on types or on programs, that indicate different calling conventions, like borrow types $\&T$ in Rust or inout parameters in Swift.

Closures and Recursion. In the preliminary ABI and compiler, we considered neither closures nor recursion. As before, to build an intuition for the ABI specification of recursive closures, we give a candidate compilation strategy. Since the target does not have closures, we need to perform closure conversion and store the closure environment in memory.

$$\frac{\Gamma \not\ni f, \overline{x} \text{ distinct} \quad F \supseteq \left\{ \begin{array}{l} \mathtt{call}_k\,(f, \overline{x_i}^{i<n})\ \left\{\overline{\mathtt{const}\ y_j = f[3+j];\ \underline{\mathtt{dup}}_{T_j}\,(y_j)}^{j<m};\ e\right\} \\ \mathtt{destr}_k\,(f)\ \left\{\overline{\mathtt{const}\ y_j = f[3+j];\ \underline{\mathtt{drop}}_{T_j}\,(y_j)}^{j<m};\ \mathtt{free}\,(f);\ 0\right\} \end{array} \right\}}{\overline{y_j : T_j}^{j<m} \vdash \mathtt{fn}\, f\, \overline{x_i}^{i<n}\{e\} : \overline{T_i}^{i<n} \to T} \ (\to I^+)$$

$$\overline{y_j : T_j}^{j<m}, f : \overline{T_i}^{i<n} \to T, \overline{x : T_i}^{i<n} \vdash e : T \rightsquigarrow e \dashv F$$

$$\rightsquigarrow \quad \mathtt{const}\ f = \mathtt{malloc}\,(3+m);\ f[0] = 1;\ f[1] = \mathtt{call}_k;\ f[2] = \mathtt{destr}_k;\ \overline{f[3+j] = y_j;}^{j<m}\ f \quad \dashv F$$

Since a closure's type does not tell us about its environment, as in prior work on type-directed closure conversion [Ahmed and Blume 2008; Mates et al. 2019; Morrisett et al. 1999], the environment will be represented using an existential package. In our ABI (see the $O\left[\!\left[\overline{T_i}^{i<n} \to T\right]\!\right]$ definition below), the environment is stored directly in the function object, but it would be just as natural to specify a version that stores a pointer to a separate environment object instead.

Next, the calling convention must be revised to require the environment as input. Instead of taking a pointer to the environment directly, the first argument to the function is the function value itself, which includes the environment in its object. Not only does this ensure the function has access to the environment, but it also naturally accommodates recursive functions. Note that in the compatibility lemma for closure introduction, this circular self-reference will require a use of Löb induction, $(\triangleright P \Rightarrow P) \vDash P$.

Finally, in accordance with RCV Release, owners of a function need to be able to deallocate its memory upon a final release. Since the type of a function's environment is not known statically

to an arbitrary caller, the function object must also store a dynamic *destructor*. As with the calling convention, the function object comes with a Hoare triple specification ensuring that the destructor correctly cleans up its memory: the precondition matches exactly the postcondition of a drop that hits zero.

$$\begin{array}{l} \mathcal{O}\left[\!\left[\overline{\mathsf{T}_i}^{i<n} \to \mathsf{T}\right]\!\right]_{\mathrm{F}}(\ell+1) \triangleq \exists\ \mathtt{call}, \mathtt{destr}, \mathit{Env}.\ \mathsf{let}\ \mathit{Self} = \ell+1 \mapsto \langle \mathtt{call}\rangle_{\mathrm{F}} \star \ell+2 \mapsto \langle \mathtt{destr}\rangle_{\mathrm{F}} \star \mathit{Env}\ \mathsf{in} \\ \quad \mathit{Self} \\ \quad \star\ \ \forall\ \overline{\mathtt{w}_i}^{i<n}.\{\bigstar_{i<n}\mathcal{V}[\![\mathsf{T}_i]\!]_{\mathrm{F}}(\mathtt{w}_i) \star @_\ell\ \mathit{Self}\}\ \langle \mathtt{call}\rangle_{\mathrm{F}}\ (\ell, \overline{\mathtt{w}_i}^{i<n})\ \{\mathcal{V}[\![\mathsf{T}]\!]_{\mathrm{F}}\}_{\mathrm{F}} \\ \quad \star\ \ \{\ell \mapsto 0 \star \mathit{Self}\}\ \langle \mathtt{destr}\rangle_{\mathrm{F}}\ (\ell)\ \{\mathit{emp}\}_{\mathrm{F}} \end{array}$$

4.2 Layout

Unboxed types. One may wish to unbox values of small types like $\mathbb{Z}$, or empty structs such as $\mathsf{struct\ Unit}\ \{\}$. This can be achieved by interposing an *unboxed predicate* $\mathcal{U}$ between the value and reference predicates, depending on the type.

$$\begin{array}{lll} \mathcal{V}[\![\mathsf{T}]\!](\mathtt{w}) & \triangleq & \begin{cases} \mathcal{U}[\![\mathsf{T}]\!](\mathtt{w}) & (\mathsf{T}\ \text{is an unboxed type; e.g., } \mathbb{Z} \text{ or } \mathsf{struct\ X}\ \{\}) \\ \ulcorner \mathtt{w} \in \mathsf{Loc} \setminus \mathtt{null} \urcorner \star \mathcal{R}[\![\mathsf{T}]\!](\mathtt{w}) & (\text{otherwise}) \end{cases} \\ \mathcal{U}[\![\mathbb{Z}]\!](\mathtt{w}) & \triangleq & \ulcorner \mathtt{w} \in \mathbb{Z} \urcorner \\ \mathcal{U}[\![\mathsf{X}]\!](\mathtt{w}) & \triangleq & \ulcorner \mathtt{w} = \mathtt{null} \urcorner \quad (\Sigma \ni \mathsf{struct\ X}\ \{\}) \end{array}$$

Since an unboxed type is not backed by any dynamic memory, it is characterized by an *unrestricted* predicate that only constrains its value. In practice, one might use distinguished types [Bolingbroke and Peyton Jones 2009; Jones and Launchbury 1991], kinds [Downen et al. 2020], or traits to mark a type as unboxed. Note that if $\mathbb{Z}$ is unboxed, then the statements of ABI Adequacy and Compiler Adequacy must change slightly—the return value must be an integer instead of a pointer, and the final memory must be completely empty.

Pointer tagging. It may only be sensible to unbox certain members of a type. For example, functions that do not close over any bindings can be represented directly as top-level function pointers, but if we allow this optimization, then a function's type is not enough to know whether it is unboxed or reference counted. Instead, we can employ *pointer tagging* [Chen and Chisnall 2019; Koparkar 2022; Marlow et al. 2007] to annotate the value itself with whether it is unboxed or a reference.

Recall that, in our memory model, a location is represented as a pair of a block identifier and an offset, and that every function has a canonical location $\langle \mathtt{f} \rangle$ that offsets into a distinguished, immutable code block. To tag the pointer, we partition the offset space: odd offsets represent unboxed functions, while even offsets represent reference-counted functions. So that the true offset i is recoverable, it is mapped to either $2i+1$ or $2i$. For unboxed functions, we only have to check that it really is a function pointer for some $\mathtt{f}$, and that $\mathtt{f}$ obeys the usual calling convention.

$$\begin{array}{l} \mathcal{V}[\![\mathsf{T}]\!]_{\mathrm{F}}(\mathtt{w}) \triangleq \begin{cases} \mathcal{U}[\![\mathsf{T}]\!]_{\mathrm{F}}(\mathtt{w}) & (\mathsf{T}\ \text{is an always-unboxed type}) \\ \exists\ b, i.\ \left(\begin{array}{l} \left(\ulcorner \mathtt{w} = \langle b, 2i+1\rangle \urcorner \star \mathcal{U}[\![\mathsf{T}]\!]_{\mathrm{F}}\ \langle b, i\rangle\right) \\ \vee\ \left(\ulcorner \mathtt{w} = \langle b, 2i\rangle \urcorner \star \mathcal{R}[\![\mathsf{T}]\!]_{\mathrm{F}}\ \langle b, i\rangle\right) \end{array}\right) & (\mathsf{T}\ \text{is a function type}) \\ \mathcal{R}[\![\mathsf{T}]\!]_{\mathrm{F}}(\mathtt{w}) & (\text{otherwise}) \end{cases} \\ \mathcal{U}\left[\!\left[\overline{\mathsf{T}_i}^{i<n} \to \mathsf{T}\right]\!\right]_{\mathrm{F}}(\ell) \triangleq \exists\ \mathtt{f}.\ \ulcorner \ell = \langle \mathtt{f}\rangle_{\mathrm{F}} \urcorner \star \forall\ \overline{\mathtt{w}_i}^{i<n}.\ \{\bigstar_{i<n}\mathcal{V}[\![\mathsf{T}_i]\!]_{\mathrm{F}}(\mathtt{w}_i)\}\ \langle \mathtt{f}\rangle_{\mathrm{F}}\ (\ell, \overline{\mathtt{w}_i}^{i<n})\ \{\mathcal{V}[\![\mathsf{T}]\!]_{\mathrm{F}}\}_{\mathrm{F}} \end{array}$$

In a memory model where pointers are flat integers, the optimization is justified by the observation that not *every* integer is a valid pointer; for example, on a 64-bit architecture where pointers have 8-byte alignment, the lowest three bits will always be zero and may be used as tag space.

Type-directed layout optimzations. Rust [Rust 2023c] and Swift [Apple 2017] support a number of type-directed layout optimizations for aggregate data that can be naturally expressed in the semantic ABI as well. For example, an enum with only one constructor need not be tagged:

$$O[\![\mathsf{X}]\!](\ell+1) \triangleq \exists\ \mathtt{w}.\ \ell+1 \mapsto \mathtt{w} \star \triangleright\, \mathcal{V}[\![\mathsf{T}]\!](\mathtt{w}) \star \mathit{size}(\ell,\ 2) \quad (\Sigma \ni \mathsf{enum\ X\ \{s : T\}})$$

"Niche optimizations" [Rust 2023d] are another common optimization that, like pointer tagging above, take advantage of the fact that certain values—so-called "niches"—are *physically* consistent with a type's layout but are not *logically* valid for that type.[2] In this class, the *null pointer optimization* is arguably the most well-known: a value of a reference type is guaranteed to be non-null, so null can be used as a one-bit tag. For example, in an Option over a reference type T, the none case can be represented by null since null would never be a valid payload in the some case:

$$\begin{array}{l}\mathsf{enum\ Option\ \{none : Unit, some : T\},\ struct\ Unit\ \{\}}\\ O[\![\mathsf{Option}]\!](\ell+1) \equiv \exists\ \mathtt{w}.\ \ell+1 \mapsto \mathtt{w} \star \triangleright\, (\mathcal{V}[\![\mathsf{Unit}]\!]^3(\mathtt{w}) \vee \mathcal{V}[\![\mathsf{T}]\!](\mathtt{w})) \star \mathit{size}(\ell,\ 2)\\ \qquad\qquad \Rightarrow \exists\ \ell'.\ \ell+1 \mapsto \ell' \star \triangleright\, (\ulcorner \ell' = \mathtt{null} \urcorner \vee (\ulcorner \ell' \neq \mathtt{null} \urcorner \star \mathcal{V}[\![\mathsf{T}]\!](\ell'))) \star \mathit{size}(\ell,\ 2)\end{array}$$

The exact mechanism that enables this optimization in Rust is slightly different: variants like None can take no data or zero-sized data, and these would not store a payload in the first place, but here, we store `null` as the payload, using the unboxing of Unit from above. Still, the core principle is the same—we take advantage of the fact that certain types have the same *physical* representation but disjoint *logical* representations, and therefore can safely share space without ambiguity. Even if the ABI itself does not adopt these optimizations, one can use the ABI specification and the specification of an optimization to prove the correctness of marshalling wrappers used at ABI-public boundaries.

$$\forall\ \mathtt{w}.\ \{\mathcal{V}_{\mathsf{opt}}[\![\mathsf{T}]\!](\mathtt{w})\}\ \mathtt{from\_opt\ (w)}\ \{\mathcal{V}[\![\mathsf{T}]\!]\} \star \{\mathcal{V}[\![\mathsf{T}]\!](\mathtt{w})\}\ \mathtt{to\_opt\ (w)}\ \{\mathcal{V}_{\mathsf{opt}}[\![\mathsf{T}]\!]\}$$

5 Library Evolution

Stabilizing an ABI has clear benefits both for interoperability and backward compatibility. Unfortunately, it can come at the cost of performance and expressivity over time, since committing to a fixed ABI can hinder opportunities to optimize and add functionality. As a result, library developers must carefully craft their interface to support forward compatibility. For example, in order to support updating a 2D `Point` struct to a 3D `Point`, a library developer might employ one of the following designs:

```
typedef struct _Point Point;
Point* mk_point(int x, int y);
int get_x(Point* p);
int get_y(Point* p);
```

upd ⇒

```
...
Point* mk_point3d(int x, int y, int z);
int get_z(Point* p);
```

```
typedef struct _Point {
 int x; int y;
 int _reserved[2];
} Point;
```

upd ⇒

```
typedef struct _Point {
 int x; int y; int z;
 int _reserved[1];
} Point;
```

These examples represent two extremes of the design space. On the top, the implementation of `Point` is totally opaque, which means it can change arbitrarily but clients must manipulate `Points` entirely by indirection. Not only can this introduce overhead and limit optimization, but also the way that the library interacts with `Point` is fundamentally different from the way that the client interacts with `Point`, so that even if the implementation is stabilized and made public, clients will

[2] In other contexts, niches are called *trap representations* [ANSSI-FR 2022].
[3] Using the unboxing of Unit as `null` from above.

not automatically take advantage of direct access. On the bottom, the implementation of `Point` is fully explicit, having reserved space for new fields like `z` ahead of time. An obvious disadvantage is that this approach is not as flexible: there is a fixed upper bound on the fields that may be added, and they may only be appended. A less obvious pitfall is that it would be easy to *misuse* the extra space: replacing `int _reserved[2];` with `long z;` would actually break compatibility because it increases the alignment from 4 to 8.[4] Additionally, both the library and newer clients would need to be prepared to handle *stale versions* from older clients, which is sometimes addressed with a version or size field in practice.

Both of these designs, and the various that fall somewhere in between them, suffer from two common weaknesses. First, the library developer must make a conscious and careful decision *up front*: if they choose wrong (or forget to make a choice), it may be impossible to preserve compatibility going forward. Second, questions about A*B*I compatibility affect the A*P*I: the most natural interface is not a forward compatible option. In its *ABI Stability Manifesto* [Apple 2017] and *Library Evolution Manifesto* [Apple 2015], Swift envisions a different world. Instead of forcing providers to carefully craft their library in a forward compatible way, the ABI *itself* provides helpful abstractions that enable certain extensibility automatically. By default, most types have a so-called *resilient layout* that trades some dynamic overhead for flexibility. Then the example above can be written in the natural way in each version, while still ensuring compatibility:

$$\mathsf{flex\ struct\ Point}\ \{x : \mathbb{Z}, y : \mathbb{Z}\} \overset{\mathsf{upd}}{\Rightarrow} \mathsf{flex\ struct\ Point}\ \{x : \mathbb{Z}, y : \mathbb{Z}, , z : \mathbb{Z}\}$$

To achieve this level of flexibility, clients may need to access data indirectly. In the example above, the offsets of the struct's fields are not necessarily stable, so they need to be looked up dynamically in a type metadata table. To mitigate overhead, only clients need to use this indirection—library code that is compiled together (which Swift calls a *resilience domain*) is always aware of the latest data definition and may continue to access data directly. Moreover, a library can opt out of resilience by marking data as @frozen, which fixes the layout so that clients can access the data directly.

The Swift ABI provides a list describing *supported evolution* for each type, dictating which changes to a library are ABI compatible and which are breaking. In terms of the semantic ABI, a library supports evolution to a new library if any program that is semantically well-typed against the old library is still semantically well-typed against the new one. In general, changing a library's implementation without changing its type is supported evolution (with certain special exceptions, like a public @inlinable function). The semantic ABI as defined in § 3.2 already has this property "baked in" because it quantifies over all semantically well-typed closing substitutions for the context, which we treat as an approximation for "the library". The more interesting case of library evolution is when the library's *interface* changes.

Definition 5.1 (Supported Evolution). Library interface $\Sigma \vdash \Gamma$ *supports evolution* to library interface $\Sigma' \vdash \Gamma'$ if $\Sigma; \Gamma \vDash_F e : T$ implies $\Sigma'; \Gamma' \vDash_F e : T$ for all e, T, F.

With our existing ABI, we have almost entirely negative evolution results. Some are sensible; for example, $\Gamma, x : T$ doesn't support evolution to Γ, since informally, e is expecting to have access to that binding. Perhaps surprisingly, the reverse direction also doesn't hold in general, since e is *not* expecting to have access to that binding but it is obligated to use or drop it, (unless $\mathcal{V}[\![T]\!]$ happens to be an unrestricted, unboxed type). More interesting are the evolution opportunities in Σ, as we saw in the examples above, and so from now on we will simply ask whether Σ supports

[4] Assuming that int is 32 bits.

evolution to Σ'. As with compiler compliance, the upshot of supported evolution is that it broadens the viable linking set; in particular, it enables *cross-version linking*.

Example 5.2 (Cross-Version Linking). Suppose a library exports data definitions in Σ_{lib} and a function $\mathsf{f}_{\mathsf{lib}} : \mathsf{T}_{\mathsf{lib}}$. Suppose a client imports these definitions, as in $\Sigma_{\mathsf{lib}}; \mathsf{f}_{\mathsf{lib}} : \mathsf{T}_{\mathsf{lib}} \vDash \mathsf{e}_{\mathsf{cli}} : \mathsf{T}_{\mathsf{cli}}$. Naturally, it is safe to link the client with any library implementation satisfying the expected interface, as in $\Sigma_{\mathsf{lib}} \vDash \mathsf{e}_{\mathsf{lib}} : \mathsf{T}_{\mathsf{lib}}$, so the library developer is free to update the implementation in any way that preserves that interface. But if Σ_{lib} supports evolution to Σ'_{lib}, then the library developer has even more flexibility: they can update the signature to Σ'_{lib} and the implementation to satisfy the new interface, $\Sigma'_{\mathsf{lib}} \vDash \mathsf{e}'_{\mathsf{lib}} : \mathsf{T}_{\mathsf{lib}}$, without breaking compatibility with the client $\mathsf{e}_{\mathsf{cli}}$.

Unfortunately, the ABI as defined so far does not support any interesting changes to Σ; not even the fields in a struct definition may be reordered, since the ABI laid them out in declaration order. Next, we will show how to revise the ABI to adopt some of the ideas from Swift's resilient layout.

5.1 Evolvable Types

$$\mathsf{Mode} \ni \mathsf{m} ::= \mathsf{flex} \mid \mathsf{rigid} \qquad \mathsf{Sig} \ni \Sigma ::= \varnothing \mid \Sigma, \mathsf{m}\ \mathsf{k}\ \mathsf{X}\ \{\overline{\mathsf{s} : \mathsf{T}}\}$$

$$\frac{\Sigma \ni \mathsf{rigid\ struct\ X}\ \{\overline{\mathsf{s} : \mathsf{T}}\} \quad \overline{\Sigma; \Gamma \vdash \mathsf{e} : \mathsf{T}}}{\Sigma; \overline{\Gamma} \vdash \{\overline{\mathsf{s} : \mathsf{e}}\} : \mathsf{X}}\ (\mathsf{rigid\ struct}\text{-I}) \qquad \frac{\Sigma \ni \mathsf{m\ struct\ X}\ \{\overline{\mathsf{s}_i : \mathsf{T}_i}^{\,i<n}\} \quad \Sigma; \Gamma \vdash \mathsf{e} : \mathsf{X} \quad j < n}{\Sigma; \Gamma \vdash \mathsf{e}.\mathsf{s}_j : \mathsf{T}_j}\ (\mathsf{m\ struct}\text{-E})$$

$$\frac{\Sigma \ni \mathsf{m\ enum\ X}\ \{\overline{\mathsf{s}_i : \mathsf{T}_i}^{\,i<n}\} \quad \Sigma; \Gamma \vdash \mathsf{e} : \mathsf{T}_j \quad j < n}{\Sigma; \Gamma \vdash \mathsf{s}_j\ \mathsf{e} : \mathsf{X}}\ (\mathsf{m\ enum}\text{-I}) \qquad \frac{\Sigma \ni \mathsf{rigid\ enum\ X}\ \{\overline{\mathsf{s} : \mathsf{T}_1}\} \quad \Sigma; \Gamma_1 \vdash \mathsf{e}_1 : \mathsf{X} \quad \overline{\Sigma; \Gamma_2, \mathsf{x} : \mathsf{T}_1 \vdash \mathsf{e}_2 : \mathsf{T}_2} \quad \Gamma_2 \not\ni \overline{\mathsf{x}}}{\Sigma; \Gamma_1, \Gamma_2 \vdash \mathsf{case}\ \mathsf{e}_1\ \{\overline{\mathsf{s}\ \mathsf{x} \Rightarrow \mathsf{e}_2}\} : \mathsf{T}_2}\ (\mathsf{rigid\ enum}\text{-E})$$

Fig. 13. Changes to the Quick source language.

To support library evolution, we update the Quick type system (Fig. 13) to mitigate the API pitfalls in the C example above by eliminating such *version confusion* statically. The first change is to the syntax of data type definitions, which now require a *resilience mode* m, either flex or rigid. rigid is analogous to @frozen in Swift, where flex is the tacit default. flex is similar to non_exhaustive in Rust [Rust 2024] in terms of its impact on the type system, but Rust does not compile non_exhaustive types differently, and the default is not quite "rigid" in the same way since Rust does not currently provide a stable ABI.

With respect to the type system, having a rigid definition in the signature is fully permissive; any data in this mode can be freely introduced and eliminated as in the original type system (Fig. 1). flex definitions are potentially less permissive. Since a flex struct may have more fields than do appear in the signature, one cannot use its constructor directly (just as in Swift [Apple 2024a]), but must use library functions instead. Likewise, since a flex enum may have more constructors than do appear in the signature, one cannot case on it exhaustively. The two options for eliminating flex enums are to require a default case, which is what Swift [Apple 2024c] and Rust [Rust 2024] do, or to outright forbid it and require clients to use library functions instead, which is the approach we adopt here. flex structs may be freely eliminated and flex enums may be freely introduced, since stability (as we will show) ensures a lower bound on the selectors available.

$$\frac{\Sigma \ni \mathsf{struct}\ X\ \{\overline{s_i : T_i}^{\,i<n}\} \quad \Sigma;\Gamma \vdash e : X \rightsquigarrow e \quad j < n}{\Sigma;\Gamma \vdash e.s_j : T_j \rightsquigarrow \mathtt{const\ r = e;\ const\ r_j = r}\left[\underline{\mathtt{sel}}\,^{s_j}_{\Sigma.X} + 1\right]\mathtt{;}\ \underline{\mathtt{dup}}_{T_j}(\mathtt{r_j})\mathtt{;}\underline{\mathtt{drop}}^{\Sigma}_{X}(\mathtt{r})\mathtt{;r_j}}\ (\text{m struct-E}^+)$$

$$\frac{\Sigma \ni \mathsf{enum}\ X\ \{\overline{s_i : T_i}^{\,i<n}\} \quad \Sigma;\Gamma \vdash e_j : T_j \rightsquigarrow e_j \quad j < n}{\Sigma;\Gamma \vdash s_j\ e_j : X \rightsquigarrow \mathtt{const\ r = malloc\,(3);\ r[0] = 1;\ r[1] = }\underline{\mathtt{sel}}\,^{s_j}_{\Sigma.X}\mathtt{;\ r[2] = e_j;\ r}}\ (\text{m enum-I}^+)$$

$$\frac{\forall\, \mathsf{rigid}\ k\ X\ \{\overline{s_i : T_i}^{\,i<n}\} \in \Sigma.\ F \supseteq \left\{\mathtt{destr}_X(\mathtt{r})\ \{\underline{\mathtt{destr}}^{\Sigma}_{X}(\mathtt{r})\},\ \overline{\mathtt{sel}^{s_i}_{X}()\ \{\underline{\mathtt{sel}}\,^{s_i}_{\Sigma.X}\}}^{\,i<n}\right\}}{\mathsf{rigid}\ \Sigma \dashv F}\ (\text{rigid } \Sigma^+)$$

$$\underline{\mathtt{sel}}\,^{s_j}_{\Sigma.X} \triangleq \begin{cases} j & (\Sigma \ni \mathsf{rigid}\ k\ X\ \{\overline{s_i : T_i}^{\,i<n}\} \wedge j < n) \\ \mathtt{sel}^{s_j}_{X}() & (\Sigma \ni \mathsf{flex}\ k\ X\ \{\overline{s_i : T_i}^{\,i<n}\} \wedge j < n) \end{cases}$$

$$\underline{\mathtt{destr}}^{\Sigma}_{X}(\mathtt{r}) \triangleq \begin{cases} \overline{\mathtt{const\ r_i = r[i+1];}\ \underline{\mathtt{drop}}^{\Sigma}_{T_i}(\mathtt{r_i})\mathtt{;}}^{\,i<n}\ \mathtt{free\,(r);\ 0} & (\Sigma \ni \mathsf{rigid\ struct}\ X\ \{\overline{s_i : T_i}^{\,i<n}\}) \\ \mathtt{destr}_X(\mathtt{r}) & (\Sigma \ni \mathsf{flex\ struct}\ X\ \{\cdots\}) \end{cases}$$

$$\underline{\mathtt{destr}}^{\Sigma}_{X}(\mathtt{r}) \triangleq \begin{cases} \mathtt{if\ (r[1] = i)\ \{} \\ \quad \overline{\mathtt{const\ r_i = r[2];}\ \underline{\mathtt{drop}}^{\Sigma}_{T_i}(\mathtt{r_i})\mathtt{;\ free\,(r);\ 0}}^{\,i<n} & (\Sigma \ni \mathsf{rigid\ enum}\ X\ \{\overline{s_i : T_i}^{\,i<n}\}) \\ \mathtt{\}\ else\ \{havoc\}} \\ \mathtt{destr}_X(\mathtt{r}) & (\Sigma \ni \mathsf{flex\ enum}\ X\ \{\cdots\}) \end{cases}$$

Fig. 14. Changes to the Quick compiler.

5.2 Compiler

Once again, we build intuition for the ABI itself by developing a candidate compiler. As mentioned above, we need to introduce some dynamic indirection in order to support resiliency. First, in flex struct projection and flex enum injection, we need to dynamically query the concrete value of the appropriate selector; we may no longer assume it to be its position in declaration order. Whereas Swift stores this information in a heap table, for simplicity, we make every selector s_i for every data definition X available as a top-level, constant function $\mathtt{sel}^{s_i}_{X}$ returning the selector value. To minimize changes to the compiler, we introduce a $\underline{\mathtt{sel}}\,^{s_i}_{\Sigma.X}$ macro that either calls $\mathtt{sel}^{s_i}_{X}$ if X is flex in Σ, or else returns the declaration order position if X is rigid in Σ, as before.

Next, in accordance with Garbage Freedom, we still need to clean up flex data, but a flex client does not have enough information to statically insert a destructor. One option would be to force clients to use library functions for dropping, as we did for introducing and eliminating flex structs and enums, respectively. But unlike those changes to the type system, this one would expose the underlying substructurality to the user, which we wish to avoid. Instead, we make a top-level destructor $\mathtt{destr}_X$ available for every data definition to ensure that flex variables can still be implicitly dropped by clients. Fortunately, we already have a $\underline{\mathtt{destr}}^{\Sigma}_{T}$ macro, so this change to the compiler is minimally invasive: we call $\mathtt{destr}_X$ if the definition is flex, else we inline its destructor if the definition is rigid, as before.

5.3 Semantic Type Substitutions

To support library evolution semantically, we synthesize the practical gadget—a type metadata table—with a classic semantic technique—*relational substitutions*, devised by Reynolds [Reynolds

1983] to characterize parametricity. The main idea is to parameterize the logical predicates by a mapping from type names to semantic types. Then the interpretation of a named type simply looks up the appropriate semantic type in the map. For type abstraction, it looks roughly like this (adapted from [Ahmed 2006]):

$$\begin{array}{lcl}
\mathcal{V}[\![\alpha]\!]_\delta(v) & \triangleq & \delta(\alpha)(v) \\
\mathcal{V}[\![\forall\, \alpha.\, T]\!]_\delta(v) & \triangleq & \forall\, P \in \mathsf{SemTy}.\, \mathcal{E}[\![T]\!]_{\delta[\alpha \mapsto P]}(v[]) \\
\mathcal{V}[\![\exists\, \alpha.\, T]\!]_\delta(v) & \triangleq & \exists\, P \in \mathsf{SemTy}, v'.\, v = \mathsf{pack}(v') \wedge \mathcal{V}[\![T]\!]_{\delta[\alpha \mapsto P]}(v') \\
\mathcal{D}[\![\Delta]\!](\delta) & \triangleq & \delta \in \mathrm{dom}(\Delta) \to \mathsf{SemTy} \\
\Delta; \Gamma \vDash e : T & \triangleq & \forall\, \delta, \gamma.\, \mathcal{D}[\![\Delta]\!](\delta) \Rightarrow \mathcal{C}[\![\Gamma]\!]_\delta(\gamma) \Rightarrow \mathcal{E}[\![T]\!]_\delta(e[\gamma])
\end{array}$$

In our case, the type name does not stand for an arbitrary type, but rather it is constrained by the signature, making it closer to a model of *bounded* polymorphism. Additionally, a $\mathsf{flex\ struct}\ \mathsf{X}\ \{\overline{\mathsf{s} : \mathsf{T}}\}$ has notable similarity to a *row polymorphic* record $\exists\, \rho.\ \{\overline{s : T}, \rho\}$, with the additional constraint that all instances of X share the same ρ.

Before deciding exactly what a *semantic signature substitution* ς is (analogous to δ above), and how we constrain it with the *signature predicate* $\mathcal{S}[\![\Sigma]\!](\varsigma)$ (analogous to $\mathcal{D}[\![\Delta]\!]$ above), let us integrate these pieces into the ABI abstractly and work backward from our goal to a suitable definition. Following the pattern for type abstraction, we extend all predicates to take a ς as a parameter, and the particular ς is chosen in the top-level judgment:

$$\Sigma; \Gamma \vDash_{\mathsf{F}} \mathsf{e} : \mathsf{T} \triangleq \forall\, \mathsf{F}' \supseteq \mathsf{F}, \varsigma, \gamma.\, \mathcal{S}[\![\Sigma]\!]_{\mathsf{F}'}(\varsigma) \star \mathcal{C}[\![\Gamma]\!]^\varsigma_{\mathsf{F}'}(\gamma) \vDash \mathcal{E}[\![\mathsf{T}]\!]^\varsigma_{\mathsf{F}'}(\mathsf{e}[\gamma])$$

Returning to the definition of Supported Evolution and unfolding this new definition for $\vDash_{\mathsf{F}}$, in order to show that Σ supports evolution to Σ', we would need

$$\forall\, \mathsf{F}' \supseteq \mathsf{F}, \varsigma, \gamma.\, \mathcal{S}[\![\Sigma]\!]_{\mathsf{F}'}(\varsigma) \star \mathcal{C}[\![\Gamma]\!]^\varsigma_{\mathsf{F}'}(\gamma) \vDash \mathcal{E}[\![\mathsf{T}]\!]^\varsigma_{\mathsf{F}'}(\mathsf{e}[\gamma]) \Rightarrow \mathcal{S}[\![\Sigma']\!]_{\mathsf{F}'}(\varsigma) \star \mathcal{C}[\![\Gamma]\!]^\varsigma_{\mathsf{F}'}(\gamma) \vDash \mathcal{E}[\![\mathsf{T}]\!]^\varsigma_{\mathsf{F}'}(\mathsf{e}[\gamma])$$

Then we want to design $\mathcal{S}$ to maximize the use of the following lemma, which easily falls out from the unfolding above.

Lemma 5.3 (Preserved Signature Evolution). *If* $\mathcal{S}[\![\Sigma']\!]_{\mathsf{F}}(\varsigma) \vDash \mathcal{S}[\![\Sigma]\!]_{\mathsf{F}}(\varsigma)$ *for all* F, ς, *then* Σ *supports evolution to* Σ'.

5.4 Evolvable Layout

Our first goal in designing $\mathcal{S}$ should be to support evolution of interest, which in our case will include reordering of selectors, addition of selectors, and marking a definition rigid. Following the pattern of relational substitutions, we should expect the object predicate for named types to roughly become

$$\mathcal{O}[\![\mathsf{X}]\!]^\varsigma(\ell + 1) \approx \varsigma(\mathsf{X})(\ell + 1)$$

When X is rigid in Σ, no further evolution is supported, so intuitively $\mathcal{S}[\![\Sigma]\!](\varsigma)$ should constrain the predicate at $\varsigma(\mathsf{X})$ to be equivalent to our preliminary definition of $\mathcal{O}[\![\mathsf{X}]\!]$ from Fig. 11, which morally was specifying rigid types already. On the other hand, when X is flex in Σ, $\mathcal{S}[\![\Sigma]\!](\varsigma)$ should constrain the predicate at $\varsigma(\mathsf{X})$ to be equivalent to *some evolution of* its rigid counterpart. For a $\mathsf{flex\ struct}\ \mathsf{X}\ \{\overline{\mathsf{s_i} : \mathsf{T_i}}^{i<n}\}$, this means its interpretation must accommodate any *superset* of fields. To achieve this, we add more structure to $\varsigma(\mathsf{X})$: it maps to a *data substitution* δ that in turn associates each selector s with an offset and a semantic type, which is exactly analogous to the type metadata table that is employed in practice. From δ, we have all the information needed to compute the actual object predicate, which refines the view that the flex definition gives us.

$$\delta.\mathsf{obj}(\ell + 1) \equiv \left(\begin{array}{ll}
 & \bigstar_{\mathsf{s} \in \{\mathsf{s_i} \mid i<n\}} \exists\, \mathsf{w_s}.\ \ell + 1 + \delta.\mathsf{sel}(\mathsf{s}).\mathsf{off} \mapsto \mathsf{w_s} \star \triangleright\, \mathcal{V}[\![\mathsf{T_i}]\!]^\varsigma(\mathsf{w_s}) \\
\star & \bigstar_{\mathsf{s} \in \mathrm{dom}(\delta.\mathsf{sel}) \setminus \{\mathsf{s_i} \mid i<n\}} \exists\, \mathsf{w_s}.\ \ell + 1 + \delta.\mathsf{sel}(\mathsf{s}).\mathsf{off} \mapsto \mathsf{w_s} \star \delta.\mathsf{sel}(\mathsf{s}).\mathsf{semty}(\mathsf{w_s}) \\
\star & \mathit{size}\,(\ell,\ 1 + |\mathrm{dom}(\delta.\mathsf{sel})|)
\end{array} \right)$$

Compared to the rigid definition, there are two notable changes. First, the offset for the selector $\mathsf{s_i}$ at position i in the *declaration order* is not necessarily at offset i *physically*, so we use the offset stored in the table $\delta.\mathsf{sel}(\mathsf{s_i}).\mathsf{off}$ instead. Second, δ may store more fields and resources than are known based on the flex definition. Note the similarity to the C example from earlier: this definition effectively applies the padding technique throughout the ABI automatically for flex definitions.

However, this definition is incomplete in two ways. First, even though the data table δ ensures that every instance of X will use the same offsets and extension types, the offsets are not yet bound to dynamically available names. In our candidate compiler, we stored each offset in a top-level function that could be used to dynamically project out of or inject into a flex struct or enum, respectively. In fact, these offset definitions should be available even for rigid definitions, since it will enable compatibility with flex clients (per Preserved Signature Evolution). We can use the weakest precondition wp to specify that the offset functions are available and return the correct offset.

$$wp_{\mathsf{F}}\left(\left\langle \mathtt{sel}_{\mathsf{X}}^{\mathsf{s_i}}\right\rangle_{\mathsf{F}}()\right)\{\mathtt{w}.\ulcorner \mathtt{w} = \varsigma(\mathsf{X}).\mathsf{sel}(\mathsf{s_i}).\mathsf{off}\urcorner\}$$

In a higher-level model, we might be able to stop here, but since one of our ABI goals is Garbage Freedom, we must ensure that deallocation is possible even for flex clients. The difficulty in deallocation comes from the extra selectors in the data substitution, which is not unlike the existentially quantified environment predicate in the closure ABI (§ 4.1). Just as we did there, the solution is to provide a dynamically available `destr` function, and use a Hoare triple to constrain it to actually deallocate the object.

$$\forall\ \ell.\ \{\ell \mapsto 0 \star \varsigma(\mathsf{X}).\mathsf{obj}(\ell+1)\}\ \langle \mathtt{destr}_{\mathsf{X}}\rangle_{\mathsf{F}}\ (\ell)\ \{emp\}_{\mathsf{F}}$$

The complete collection of changes is given in Fig. 15. $\mathcal{S}[\![\Sigma]\!]$ ensures that every entry in the signature Σ has a data table δ with the appropriate kind k and at least as many selectors s as appear in the definition. rigid definitions are further constrained so that the indices in the data table exactly match the declaration order. The data table must also map each declared selector to its expected semantic type, guarded by a later, as before, in order to support recursive definitions. These are combined into a single $\delta.\mathsf{obj}$ predicate, as above, but only after considering the kind k, though enums are handled in an analogous way using disjunction $\vee$. $\mathcal{S}$ additionally ensures that every definition has an available destructor and that all selectors are available. Notice that $\mathcal{S}$ is an *unrestricted* predicate—it does not own any resources of its own.

With the definition of $\mathcal{S}$ in place, we can prove the following lemma for use with Preserved Signature Evolution. Informally, it says that a flex definition for X is refined by any definition for X with a superset of its selectors, which includes extensions, reorderings, and making the definition rigid.

Lemma 5.4 (Signature Preservation). *If* $\{\mathsf{s_j} : \mathsf{T_j} \mid j < m\} \supseteq \{\mathsf{s_i} : \mathsf{T_i} \mid i < n\}$, *then*

$$\mathcal{S}\left[\!\left[\Sigma, \mathsf{m\ k\ X}\ \{\overline{\mathsf{s_j} : \mathsf{T_j}}^{\,j<m}\}\right]\!\right]_{\mathsf{F}}(\varsigma) \vDash \mathcal{S}\left[\!\left[\Sigma, \mathsf{flex\ k\ X}\ \{\overline{\mathsf{s_i} : \mathsf{T_i}}^{\,i<n}\}\right]\!\right]_{\mathsf{F}}(\varsigma)$$

One subtle impact of these changes is that when $\Sigma; \varnothing \vdash \mathsf{e} : \mathsf{T} \rightsquigarrow \mathsf{e} \dashv \mathsf{F}$, e may not be closed if Σ contains flex definitions—one may think of such a program as a client still waiting to be linked with a library implementation. This means that the statements of ABI Adequacy and Compiler Adequacy must additionally restrict Σ to be fully rigid, and the latter will also refer to signature compilation $\Sigma \dashv \mathsf{F}$. As is standard for typing contexts, this is not a significant limitation because one can always close off an open program, which for signatures would involve using library evolution to link with a fully rigid implementation.

$$\varsigma \in \mathsf{SigSub} \triangleq \mathsf{X} \xrightarrow{\text{fin}} \mathsf{DataSub}$$
$$\delta \in \mathsf{DataSub} \triangleq \left\langle \mathsf{kind} : \mathsf{Kind}, \mathsf{sel} : \mathsf{Sel} \xrightarrow{\text{fin}} \langle \mathsf{off} : \mathbb{N}, \mathsf{semty} : \mathit{Prd}\,(\mathtt{Word}) \rangle \right\rangle$$

$$\mathcal{S}[\![\Sigma]\!]_{\mathtt{F}}(\varsigma) \triangleq \; !\left(\begin{array}{l} \ulcorner \mathrm{dom}(\varsigma) \supseteq \mathrm{dom}(\Sigma) \urcorner \\ \star\ \forall\ \mathsf{m}\ \mathsf{k}\ \mathsf{X}\ \{\overline{\mathsf{s_i} : \mathsf{T_i}}^{\,i<n}\} \in \Sigma.\ \mathsf{let}\ \delta = \varsigma(\mathsf{X})\ \mathsf{in} \\ \quad \ulcorner \delta.\mathsf{kind} = \mathsf{k} \urcorner \\ \quad \star\ \ulcorner \mathrm{dom}(\delta.\mathsf{sel}) \supseteq \{\mathsf{s_i} \mid i < n\} \urcorner \\ \quad \star\ \forall\ i < n.\ !\ \mathit{wp}_{\mathtt{F}}\left(\left\langle \mathtt{sel}^{\mathsf{s_i}}_{\mathsf{X}} \right\rangle_{\mathtt{F}}()\right)\{\mathtt{w}.\ \ulcorner \mathtt{w} = \delta.\mathsf{sel}(\mathsf{s_i}).\mathsf{off} \urcorner\} \\ \quad \star\ \forall\ i < n, \mathtt{w}.\ \delta.\mathsf{sel}(\mathsf{s_i}).\mathsf{semty}(\mathtt{w}) \equiv \triangleright\, \mathcal{V}[\![\mathsf{T_i}]\!]^{\varsigma}_{\mathtt{F}}(\mathtt{w}) \\ \quad \star\ \forall\ \ell.\ \{\ell \mapsto 0 \star \delta.\mathsf{obj}(\ell+1)\}\ \langle \mathtt{destr}_{\mathsf{X}} \rangle_{\mathtt{F}}(\ell)\ \{\mathit{emp}\}_{\mathtt{F}} \\ \quad \star\ \ulcorner \mathsf{m} = \mathsf{rigid} \Rightarrow \mathrm{dom}(\delta.\mathsf{sel}) \subseteq \{\mathsf{s_i} \mid i < n\} \wedge \forall\, i < n.\ \delta.\mathsf{sel}(\mathsf{s_i}).\mathsf{off} = i \urcorner \end{array}\right)$$

$$\delta.\mathsf{obj}(\ell+1) \triangleq \begin{cases} \mathit{size}\,(\ell,\ 1 + |\mathrm{dom}(\delta.\mathsf{sel})|) \\ \star\ \bigstar_{\mathsf{s} \in \mathrm{dom}(\delta.\mathsf{sel})} \exists\ \mathtt{w_s}.\left(\begin{array}{l} \ell + 1 + \delta.\mathsf{sel}(\mathsf{s}).\mathsf{off} \mapsto \mathtt{w_s} \\ \star\ \delta.\mathsf{sel}(\mathsf{s}).\mathsf{semty}(\mathtt{w_s}) \end{array}\right) & (\delta.\mathsf{kind} = \mathsf{struct}) \\ \hline \mathit{size}\,(\ell,\ 3) \\ \star\ \bigvee_{\mathsf{s} \in \mathrm{dom}(\delta.\mathsf{sel})} \exists\ \mathtt{w_s}.\left(\begin{array}{l} \ell + 1 \mapsto \delta.\mathsf{sel}(\mathsf{s}).\mathsf{off} \\ \star\ \ell + 2 \mapsto \mathtt{w_s} \\ \star\ \delta.\mathsf{sel}(\mathsf{s}).\mathsf{semty}(\mathtt{w_s}) \end{array}\right) & (\delta.\mathsf{kind} = \mathsf{enum}) \end{cases}$$

$$O[\![\mathsf{X}]\!]^{\varsigma}_{\mathtt{F}}(\ell+1) \triangleq \varsigma(\mathsf{X}).\mathsf{obj}(\ell+1)$$
$$\Sigma; \Gamma \vDash_{\mathtt{F}} \mathtt{e} : \mathsf{T} \triangleq \forall\, \mathtt{F}' \supseteq \mathtt{F}, \varsigma, \gamma.\ \mathcal{S}[\![\Sigma]\!]_{\mathtt{F}'}(\varsigma) \star C[\![\Gamma]\!]^{\varsigma}_{\mathtt{F}'}(\gamma) \vDash \mathcal{E}[\![\mathsf{T}]\!]^{\varsigma}_{\mathtt{F}'}(\mathtt{e}[\gamma])$$

Fig. 15. Changes to the ABI.

6 Related Work and Discussion

Separation Logic for Low-Level Code. Separation logic [O'Hearn et al. 2001; Reynolds 2002] has long been employed to verify low-level code; e.g., for C [Tuch et al. 2007] and assembly [Jensen et al. 2013]. Its continued application has been supported by the development of increasingly rich notions of ghost state [Appel 2014; Dinsdale-Young et al. 2013; Jung et al. 2018; Ley-Wild and Nanevski 2013] and the broader adoption of step-indexing [Ahmed 2006, 2004; Appel and McAllester 2001]. Our use of separation logic to specify logical relations is inspired by Jung et al. [2017], who use it to model semantic types for Rust, and Timany et al. [2022], who use it to model semantic types for a concurrent ML-like language. While we have not mechanized our results, we do believe they would be naturally expressible in a framework like Iris [Jung et al. 2018], except that, as an affine separation logic, one would need a different mechanism or a different framework (e.g., [Bizjak et al. 2019; Jacobs et al. 2024]) to show garbage freedom.

"Realistic Realizability". Benton and collaborators [Benton 2006; Benton and Tabareau 2009; Benton and Zarfaty 2007] laid the groundwork for modelling high-level types with a realizability model over target programs, which they employed to prove semantic type soundness of compilers. Patterson and collaborators [Patterson et al. 2022, 2023] recently revived this idea to study static and dynamic mechanisms for cross-language interoperability, which are proven sound using realizability models over a common target language. Our work builds on these efforts by making the simple observation that, if designed appropriately, a realizability model can be more than just a *proof* technique—it can be broadly useful for analysis in any context where an ABI specification would be used in practice.

Dynamic Software Updates. Although library evolution as devised by Swift has not been previously studied, Hicks and collaborators studied the problem of *dynamic software updates* [Hicks et al. 2001; Sewell et al. 2008; Stoyle et al. 2005], which shares a similar goal of updating APIs without breaking existing clients. Whereas Swift is mostly interested in updating libraries without *recompiling* clients, dynamic software updates occur on the fly without even *restarting* clients. The mechanism that enables a dynamic software update is a *dynamic patch*, which is like a wrapper function interposed between outdated uses of an API. One can theoretically update an API arbitrarily given a sufficient patch, but this level of flexibility sometimes necessitates input from the library developer. On the other hand, library evolution—as formalized in this paper—only supports a restricted class of API updates, but these are entirely facilitated by the ABI and the compiler. In particular, as long as the library developer updates their API in a supported way, library evolution guarantees backwards compatibility "for free".

Reference Counting. Our use of an internally substructural type system for reference counting is inspired by Perceus [Reinking et al. 2021], which applies the technique in the Koka language along with more sophisticated optimizations like borrowing and reuse. To our knowledge, we are the first to prove semantic type soundness for a compiled language implementation with automatic reference counting. However, there has been work on verifying reference-counting algorithms using model checking and separation logic [Emmi et al. 2009; Windsor et al. 2017]. Notably, Doko and Vafeiadis [2017] define an $\mathsf{ARC}(\ell, \mathtt{w})$ proposition similar to $@_\ell\ (\ell + 1 \mapsto \mathtt{w})$, and Mulder et al. [2022] define a $\mathsf{token}_{\mathsf{id}}(P)$ proposition similar to $@_\ell\, P$, except that id is a ghost name rather than a physical location. Both abstractions implement an interface similar to that of our jumps, but their models track the total reference count using ghost state in a global invariant, which is not tracked by our model. Additionally, neither of their models use graphs for resources, so they do not seem to have a clear analog to our reachability modality $\Diamond P$, which supports reasoning about non-destructive updates to resources that are accessible through a chain of references. Madiot and Pottier [2022] develop a domain-specific separation logic with a *pointed-by assertion* $\ell \hookleftarrow L$, where L over-approximates all the locations holding a pointer to ℓ. They show that the pointed-by assertion supports a style of reasoning called *ghost* reference counting, which is useful when working in a garbage collected language. On the other hand, our logic is designed to reason about *physical* reference counters, and we *implement* the mechanism in terms of a target language—it is not built into the operational semantics, as their garbage collector is. Lorenzen et al. [2023] do not develop a separation logic for reference counting, but they do define a composition operator on flat, reference-counted heaps that satisfies the properties of a resource algebra and correctly populates reference counts.

ABI Formalization. To our knowledge, we are the first to use realizability to study ABIs, but others have explored alternative formalizations and related concepts. Drossopoulou et al. [1998] studied binary compatibility in the context of Java, proposing a formalization of safe linking and execution without recompilation. Whereas our specification employs realizability into a compilation target, theirs uses a type-preserving translation from source Java into an extension of Java with compiler annotations and run-time constructs like addresses. CompCert [Leroy 2023] aims to conform with its platforms' ABIs, though it does not formally specify each ABI and prove this. CompCertELF [Wang et al. 2020] extends CompCert with a verified translation to ELF object files, which is a popular format for platform ABIs to use. To reckon with the rigidity of the C/C++ ABI, Atkinson et al. [2010] use macros to automatically implement some of the idioms for forward compatibility described in § 5.

The Semantic ABI Recipe. The primary purpose of the Quick case study was to motivate the "realistic realizability" approach to a semantic ABI specification. However, other than Library Evolution (§ 5), which has the potential to be adopted by other languages but is presently most relevant to Swift, the rest of the approach can be readily adapted to suit other typed languages. To construct a semantic ABI, an important first step is to develop a suitable domain-specific logic over target terms. For Quick, this included predicates and proof rules tailored to manipulating reference counts, but for languages that use different dynamic mechanisms or stronger static mechanisms for memory safety, one would instead choose abstractions better suited for those techniques (e.g., for garbage collection [Madiot and Pottier 2022] or borrowing [Jung et al. 2017]). Then to define the ABI, one interprets each source type as a predicate over target terms. Finally, a logical next step for compiled languages is to verify compiler compliance by showing that the compiler only produces terms that satisfy the ABI.

Future Work. Cross-language interoperability is another important application of ABIs, and we are interested in using our formalization to specify a foreign function interface (FFI) and then verify the soundness of FFI guidelines (e.g., for Rust FFI [ANSSI-FR 2022]) and a bindings-generation tool (e.g., rust-bindgen [Rust 2023a]). Though we have approximated some of the important features and ABI considerations in Swift, we would like to incorporate more of its bespoke features like its mutable value semantics and object system. Finally, we plan to apply this realistic realizability technique to specify a Rust ABI, in hopes that using the semantic ABI for analysis will help settle some of the ongoing debates in Rust's ABI proposal [RFCs 2023]. For this ABI, we plan to target WebAssembly [Haas et al. 2017], perhaps leveraging its component model proposal [WebAssembly 2023] or a more richly-typed variant [Fitzgibbons et al. 2024].

Acknowledgments

We thank Daniel Patterson, John Li, and Olek Gierczak for their careful feedback and suggestions. We also thank the anonymous reviewers for OOPSLA'24 and PriSC'24 for their valuable comments. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agency.

References

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*. Springer, 69–83. https://doi.org/10.1007/11693024_6

Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 157–168. https://doi.org/10.1145/1411204.1411227

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: a linear language with locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University.

Emilio Cobos Álvarez. 2018. *i128 / u128 are not compatible with C's definition.* GitHub. https://github.com/rust-lang/rust/issues/54341 GitHub issue #54341.

ANSSI-FR. 2022. Rust Guide: Foreign Function Interface. https://github.com/ANSSI-FR/rust-guide/blob/master/src/en/07_ffi.md.

Andrew W Appel. 2014. *Program logics for certified compilers.* Cambridge University Press. https://doi.org/10.1017/cbo9781107256552

Andrew W Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 109–122. https://doi.org/10.1145/1190215.1190235

Apple. 2015. Library Evolution. https://github.com/apple/swift/blob/main/docs/LibraryEvolution.rst.

Apple. 2017. ABI Stability Manifesto. https://github.com/apple/swift/blob/main/docs/ABIStabilityManifesto.md.

Apple. 2024a. The Swift Book: Access Control—Default Memberwise Initializers. https://github.com/apple/swift-book/blob/main/TSPL.docc/LanguageGuide/AccessControl.md#default-memberwise-initializers-for-structure-types.

Apple. 2024b. The Swift Book: Automatic Reference Counting. https://github.com/apple/swift-book/blob/main/TSPL.docc/LanguageGuide/AutomaticReferenceCounting.md.

Apple. 2024c. The Swift Book: Switch Statement. https://github.com/apple/swift-book/blob/8b7826c2a1809b5d93651d92555510a9c46502dd/TSPL.docc/ReferenceManual/Statements.md#switch-statement.

Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. 2010. ABI compatibility through a customizable language. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 147–156. https://doi.org/10.1145/1868294.1868316

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

Nick Benton. 2006. Abstracting allocation: The new new thing. In *International Workshop on Computer Science Logic*. Springer, 182–196.

Nick Benton and Nicolas Tabareau. 2009. Compiling functional types to relational specifications for low level imperative code. In *Proceedings of the 4th international workshop on Types in language design and implementation*. 3–14. https://doi.org/10.1145/1481861.1481864

Nick Benton and Uri Zarfaty. 2007. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 1–12. https://doi.org/10.1145/1273920.1273922

Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. 2006. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science* 2 (2006). https://doi.org/10.2168/lmcs-2(5:1)2006

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing obligations in higher-order concurrent separation logic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30. https://doi.org/10.1145/3290378

Maximilian C Bolingbroke and Simon L Peyton Jones. 2009. Types are calling conventions. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. 1–12. https://doi.org/10.1145/1596638.1596640

Torben Braüner and Valeria de Paiva. 2006. Intuitionistic hybrid logic. *Journal of Applied Logic* 4, 3 (2006), 231–255. https://doi.org/10.1016/j.jal.2005.06.009

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2019. Domain-Aware Session Types. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands (LIPIcs, Vol. 140)*, Wan J. Fokkink and Rob van Glabbeek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:17. https://doi.org/10.4230/LIPICS.CONCUR.2019.39

Tony Chen and David Chisnall. 2019. Pointer Tagging for Memory Safety.

Aria Desires. 2023. abi-cafe. https://github.com/Gankra/abi-cafe.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 287–300. https://doi.org/10.1145/2480359.2429104

Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*. Springer, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

Paul Downen, Zena M Ariola, Simon Peyton Jones, and Richard A Eisenberg. 2020. Kinds are calling conventions. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29. https://doi.org/10.1145/3408986

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *Logical Methods in Computer Science* 7 (2011). https://doi.org/10.2168/lmcs-7(2:16)2011

Sophia Drossopoulou, David Wragg, and Susan Eisenbach. 1998. What is Java binary compatibility?. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 341–361. https://doi.org/10.1145/286936.286974

Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. 2009. Verifying reference counting implementations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 352–367. https://doi.org/10.1007/978-3-642-00768-2_30

Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1656–1679. https://doi.org/10.1145/3656444

Python Software Foundation. 2023. Python/C API Reference Manual. https://docs.python.org/3/c-api/intro.html#reference-counts.

Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200. https://doi.org/10.1145/3140587.3062363

Michael Hicks, Jonathan T Moore, and Scott Nettles. 2001. Dynamic software updating. *ACM SIGPLAN Notices* 36, 5 (2001), 13–23. https://doi.org/10.1145/381694.378798

Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1385–1417. https://doi.org/10.1145/3632889

Jonas B Jensen, Nick Benton, and Andrew Kennedy. 2013. High-level separation logic for low-level code. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 301–314. https://doi.org/10.1145/2480359.2429105

Simon L Peyton Jones and John Launchbury. 1991. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture: 5th ACM Conference Cambridge, MA, USA, August 26–30, 1991 Proceedings 5*. Springer, 636–666. https://doi.org/10.1007/3540543961_30

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/s0956796818000151

Chaitanya Koparkar. 2022. A primer on pointer tagging. *XRDS: Crossroads, The ACM Magazine for Students* 29, 1 (2022), 66–68. https://doi.org/10.1145/3558200

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. 2017. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices* 52, 6 (2017), 633–647. https://doi.org/10.1145/3062341.3062343

Xavier Leroy. 2023. *The CompCert C verified compiler: Documentation and user's manual*. Ph. D. Dissertation. Inria.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. The CompCert memory model. In *Program Logics for Certified Compilers*, Andrew W. Appel (Ed.). Cambridge University Press. http://vst.cs.princeton.edu/

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 561–574. https://doi.org/10.1145/2480359.2429134

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP (2023), 275–304. https://doi.org/10.1145/3607840

Jean-Marie Madiot and François Pottier. 2022. A separation logic for heap space under garbage collection. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498672

Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. 2007. Faster laziness using dynamic pointer tagging. *Acm sigplan notices* 42, 9 (2007), 277–288. https://doi.org/10.1145/1291151.1291194

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under control: Compositionally correct closure conversion with mutable state. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–15. https://doi.org/10.1145/3354166.3354181

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568. https://doi.org/10.1145/319301.319345

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 809–824. https://doi.org/10.1145/3519939.3523432

Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*. IEEE, 255–266.

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 609–624. https://doi.org/10.1145/3519939.3523703

Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation using Linking Types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*. 14–28. https://doi.org/10.1145/3609027.3609405

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111. https://doi.org/10.1145/3453483.3454032

John C Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 513–523.

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.

Rust Language RFCs. 2023. #3470: crABI v1. https://github.com/rust-lang/rfcs/pull/3470.

Rust. 2023a. rust-bindgen. https://github.com/rust-lang/rust-bindgen.

Rust. 2023b. The Rust Book: Variables and Data Interacting with Move. https://github.com/rust-lang/book/blob/main/src/ch04-01-what-is-ownership.md#variables-and-data-interacting-with-move.

Rust. 2023c. Unsafe Code Guidelines: Data Layout—Enums. https://github.com/rust-lang/unsafe-code-guidelines/blob/master/reference/src/layout/enums.md.

Rust. 2023d. Unsafe Code Guidelines: Glossary—Niche. https://github.com/rust-lang/unsafe-code-guidelines/blob/master/reference/src/glossary.md#niche.

Rust. 2024. The Rust Reference: The non_exhaustive Attribute. https://github.com/rust-lang/reference/blob/master/src/attributes/type_system.md#the-non_exhaustive-attribute.

Peter Sewell, Gareth Stoyle, Michael Hicks, Gavin Bierman, and Keith Wansbrough. 2008. Dynamic rebinding for marshalling and update, via redex-time and destruct-time reduction. *Journal of Functional Programming* 18, 4 (2008), 437–502.

Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. 2005. Mutatis mutandis: safe and predictable dynamic software updating. *ACM SIGPLAN Notices* 40, 1 (2005), 183–194. https://doi.org/10.1145/1040305.1040321

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 1996. TIL: A type-directed optimizing compiler for ML. *ACM Sigplan Notices* 31, 5 (1996), 181–192. https://doi.org/10.21236/ada306265

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. *Reported under submission on https://iris-project. org/(2022). https://iris-project. org/pdfs/2022-submitted-logical-type-soundness. pdf* (2022). https://doi.org/10.1145/3676954

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 97–108. https://doi.org/10.1145/1190215.1190234

Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On (Supplementary Material). *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024). https://doi.org/10.1145/3689755

Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: verified separate compilation of C programs into ELF object files. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428265

WebAssembly. 2023. Component Model. https://github.com/WebAssembly/component-model.

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 544–569. https://doi.org/10.1007/978-3-319-63387-9_27

Received 2024-04-06; accepted 2024-08-18