
# Destabilizing Iris

SIMON SPIES, MPI-SWS, Germany
NIKLAS MÜCK, MPI-SWS, Germany
HAOYI ZENG, Saarland University, Germany
MICHAEL SAMMLER, ETH Zurich, Switzerland and ISTA, Austria
ANDREA LATTUADA, MPI-SWS, Germany
PETER MÜLLER, ETH Zurich, Switzerland
DEREK DREYER, MPI-SWS, Germany

The separation logic framework Iris has been built on the premise that all assertions are *stable*, meaning they unconditionally enjoy the famous *frame rule*. This gives Iris—and the numerous program logics that build on it—very modular reasoning principles. But stability also comes at a cost. It excludes a core feature of the Viper verifier family, *heap-dependent expression assertions*, which lift program expressions to the assertion level in order to reduce redundancy between code and specifications and better facilitate SMT-based automation.

In this paper, we bring heap-dependent expression assertions to Iris with **Daenerys**. To do so, we must first revisit the very core of Iris, extending it with a new form of *unstable resources* (and adapting the frame rule accordingly). On top, we then build a program logic with heap-dependent expression assertions and lay the foundations for connecting Iris to SMT solvers. We apply Daenerys to several case studies, including some that go beyond what Viper and Iris can do individually and others that benefit from the connection to SMT.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: verification, separation logic, implicit dynamic frames, Iris, Rocq

**ACM Reference Format:**
Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. 2025. Destabilizing Iris. *Proc. ACM Program. Lang.* 9, PLDI, Article 181 (June 2025), 26 pages. https://doi.org/10.1145/3729284

## 1 Introduction

*Separation logic (SL)* [44, 50] is one of the most highly influential developments in programming language foundations in the past 25 years. It serves not only as the basis of many cutting-edge verification tools (*e.g.,* Verifast [26], Gillian [38, 53], CN [46], and Verus [36]), but also as the ancestor of a wide variety of program logics and logical frameworks, especially for concurrent programs [2, 11, 13, 16, 21, 58, 59, 62]. The latter line of work has culminated most recently in *Iris* [29, 31, 34], a Rocq-based framework that enables users to develop their own customized separation logics for different verification problems, programming languages, and application domains. Iris has been used to support high-confidence, machine-checked separation logic proofs in many recent research projects [12, 28, 48, 55, 60] (for a full list, see https://iris-project.org).

Authors' Contact Information: Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Niklas Mück, MPI-SWS, Saarland Informatics Campus, Germany, mueck@mpi-sws.org; Haoyi Zeng, Saarland University, Saarland Informatics Campus, Germany, haze00001@stud.uni-saarland.de; Michael Sammler, ETH Zurich, Zurich, Switzerland and ISTA, Klosterneuburg, Austria, michael.sammler@ist.ac.at; Andrea Lattuada, MPI-SWS, Saarland Informatics Campus, Germany, andrea@mpi-sws.org; Peter Müller, ETH Zurich, Department of Computer Science, Zurich, Switzerland, peter.mueller@inf.ethz.ch; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.


ACM 2475-1421/2025/6-ART181
https://doi.org/10.1145/3729284

The forte of SL—and the reason why frameworks like Iris build on it—is that it enables a modular style of reasoning for complex imperative programs via the concept of *ownership*. That is, assertions in SL not only talk about the current state of the program but also convey ownership for accessing or modifying it. The canonical example of this is the *points-to assertion*, $\ell \mapsto v$, which asserts not only that the pointer $\ell$ currently points to the value $v$ but also that the function being verified—call it $f$—"owns" the memory location $\ell$. As a result, $f$ does not have to worry about other parts of the program "interfering" w.r.t. $\ell$; rather, the verification of $f$ can rely on $\ell$ continuing to store $v$ unless (1) $f$ itself mutates $\ell$ or (2) $f$ transfers ownership of $\ell$ to another part of the program. By validating this kind of *local reasoning*, SL enables large programs to be verified compositionally.

However, separation logic is not the only formal foundation for ownership reasoning. Another closely related, yet decidedly different, foundation is that of *implicit dynamic frames (IDF)* [54]. Deployed most extensively as the foundation of the Viper verification framework [5, 8, 23, 42, 65], IDF is similar to SL in that assertions can talk both about the state of the program and ownership of that state. But unlike in SL, IDF assertions do not have to talk about these two things simultaneously—rather, they are disentangled. For example, instead of the single SL assertion $\ell \mapsto v$, which combines ownership of $\ell$ with the fact that $\ell$ points to $v$, IDF has two kinds of assertions: (1) an *access assertion* $\mathrm{acc}(\ell)$, which conveys ownership of the location $\ell$ (entailing the right to access and update $\ell$) but does not say what $\ell$ points to, and (2) *heap-dependent expression assertions (HDEAs)* such as $!\,\ell =_{\mathrm{IDF}} v$, which says that $\ell$ points to $v$ but does not assert ownership of $\ell$. (HDEAs, as we will see below, are not limited to assertions about a single memory location $\ell$; they may also, for example, include assertions about the results of function calls.)

Separation logic's $\ell \mapsto v$ can be expressed in IDF as $\mathrm{acc}(\ell) * !\,\ell =_{\mathrm{IDF}} v$. Conversely, IDF's access assertion $\mathrm{acc}(\ell)$ can be expressed in SL as $\mathrm{acc}(\ell) \triangleq \exists v.\, \ell \mapsto v$. But how can one encode HDEAs in SL? It is not so simple. For example, take Iris: suppose that $\mathrm{acc}(\ell)$ and $!\,\ell =_{\mathrm{IDF}} v$ could be expressed as separate Iris assertions (conjoined by the separating conjunction $*$). Using $\mathrm{acc}(\ell)$, we could update $\ell$ to $w$, thereby obtaining $!\,\ell =_{\mathrm{IDF}} w$. But by the famous *frame rule* [44, 50], we would be able to frame the assertion $!\,\ell =_{\mathrm{IDF}} v$ around the update, thus leading to a contradiction!

Nevertheless, in this paper, we will show that in fact HDEAs (and more generally IDF-style reasoning) *can* be soundly incorporated into SL[1]—and Iris in particular—and that they constitute a demonstrably useful extension to the Iris toolbox. To do so, we will need to revisit the foundations of Iris in order to support *unstable resources*, a new type of resources that do not unconditionally enjoy the frame rule. But before we get into more details about our contributions (§1.2), let us begin by reviewing why we want to bring HDEAs to Iris in the first place (§1.1).

### 1.1 Why Heap-Dependent Expression Assertions are Useful

To illustrate the utility of HDEAs, let us consider a concrete example (using OCaml-like notation):

```
1 let buf = produce_buffer() in     ■ {buf ↦ u⃗}
2 let chk1 = checksum(buf)   in     ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗)}
3 read_only_client(buf);            ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗)}
4 let chk2 = checksum(buf)   in     ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗) ∗ chk2 = cs(u⃗)}
5 assert(chk1 == chk2)
```

In this example, we allocate a buffer of 64-bit unsigned integers on the heap using the function `produce_buffer` (Line 1) and then pass it to a client (Line 3). The client is only allowed to read from the buffer. Thus, we can compute a checksum of the buffer before (Line 2) and after the

[1] Parkinson and Summers [45] show how to encode SL in IDF with $\ell \mapsto v \triangleq \mathrm{acc}(\ell) \wedge !\,\ell =_{\mathrm{IDF}} v$. In this paper, we consider the *reverse direction*: we want to bring IDF to SL, and we do so for one of the most expressive separation logics out there, Iris.

$$\{\mathsf{True}\}\ \texttt{produce\_buffer}()\ \{v.\ \exists b, \vec{u}.\ v = b * b \mapsto \vec{u}\}$$

$$\{b \mapsto \vec{u}\}\ \texttt{read\_only\_client}(b)\ \{\_.\ b \mapsto \vec{u}\} \qquad \{b \mapsto \vec{u}\}\ \texttt{checksum}(b)\ \{v.\ v = cs(\vec{u}) * b \mapsto \vec{u}\}$$

Fig. 1. Separation logic specifications of the operations in the motivating example.

read-only client (Line 4) and assert that they are the same (Line 5). (The exact algorithm by which the checksum is computed does not matter, as long as it is deterministic.)

Let us first sketch how one could show that the assert always succeeds *in separation logic*. We have annotated intermediate proof states (in ■ gray), and we use the specifications depicted in Fig. 1. First, we allocate the buffer and obtain the ownership of `buf` currently storing a sequence of 64-bit unsigned integers $\vec{u}$ (Line 1). Then, we use it to compute `checksum(buf)`. The result is $cs(\vec{u})$, where $cs$ is a mathematical version of `checksum` operating on the contents $\vec{u}$ (in Line 2). Next, we pass the buffer `buf` to the read-only client, which does not change its contents $\vec{u}$. Thus, when we recompute the checksum (Line 4), it is still $cs(\vec{u})$ and the assert—comparing $cs(\vec{u})$ and $cs(\vec{u})$—succeeds (Line 5).

This verification works. But it is more laborious than it appears at first glance, because to complete it, we must additionally verify the Hoare triples in Fig. 1. This involves a non-trivial amount of work: (1) *reformulating the implementation* of `checksum` as a mathematical function $cs$ and (2) *proving full functional correctness* of `checksum` by showing that it implements $cs$. If `checksum` is small and simple, proving its functional correctness is not a big burden—but if it is a nontrivial recursive function, verifying it becomes tedious quickly. Moreover, functional correctness of `checksum` is a much stronger property than we actually need! The assert succeeds so long as (1) the result of `checksum` depends only on the buffer, (2) `checksum` does not modify the buffer, and (3) the read-only client does not modify the buffer. (Yet, we cannot weaken the specification to say that `checksum` returns just *some integer*, because it must be the same one in Line 2 and Line 4.)

*HDEAs* offer a simpler way to handle such examples. They enrich the assertion language with the ability to describe the current result of program expressions in the deterministic, read-only fragment of the programming language. In particular, they permit $\texttt{checksum(buf)} =_{\text{IDF}} \texttt{chk1}$ (note the use of `checksum` instead of $cs$) as a logic-level assertion, which says that the current value of `checksum(buf)` is `chk1`. Since the client does not modify `buf`, one can then frame $\texttt{checksum(buf)} =_{\text{IDF}} \texttt{chk1}$ around it using an ownership argument, which we will spell out in §2.1. Thus, when we reach Line 5, we know $\texttt{checksum(buf)} =_{\text{IDF}} \texttt{chk1}$ and $\texttt{checksum(buf)} =_{\text{IDF}} \texttt{chk2}$, from which we can deduce `chk1` = `chk2`. In short, HDEAs let us validate our assert while avoiding a needless detour through functional correctness of `checksum`.

**Beyond this example.** The above example is an instance of a more general pattern that arises in SL verification: *redundancy between specifications and implementations*. A typical SL specification abstracts data structures to some mathematical representation (*e.g.,* abstracting a buffer to a sequence of values) and implementation functions to mathematical functions (*e.g.,* abstracting `checksum` to $cs$ above). These are then used to specify the concrete implementation. While data abstraction is undoubtedly useful in program verification, requiring a mathematical counterpart for each implementation function is especially tedious and redundant for cases where the specification ends up more or less just mirroring the implementation (prominent examples include getter functions, comparison operations, and mathematical computations). In those cases, HDEAs shine: they enable one to simply talk about the result of running some code at the assertion level (so long as that result is well-defined) *without* having to develop a mathematical abstraction of it first.

Reducing redundancy is not the only strength of HDEAs. For example, as we will see later in the paper, HDEAs also facilitate the iterative development of proofs, whereby specifications are

strengthened step by step in order to incrementally model more complex aspects of a program's behavior (see §6). Moreover, they unlock a new kind of automation for Iris, namely SMT solvers. In particular, as mentioned above, Iris is a foundational separation logic framework embedded into Rocq, but a downside of its embedding within Rocq is lackluster automation for, *e.g.*, mathematical equality in the presence of theories such as bitvectors and uninterpreted functions. In this paper, we will develop the foundations for connecting *HDEAs in Iris* to *formulas in first-order logic*. This connection then allows one to benefit from the automation of an SMT solver for theories like integers, bitvectors, and uninterpreted functions when reasoning about HDEAs in Iris (see §2.3).

### 1.2 Daenerys: Adding Heap-Dependent Expression Assertions to Iris

We introduce **Daenerys**. The main theoretical contribution of Daenerys is bringing HDEAs to Iris by extending its resource model with unstable resources. In doing so, we combine the benefits of HDEAs (described above) with the expressivity of Iris, including step-indexing [1, 3], persistent propositions [22], impredicative invariants [58], fine-grained concurrency, user-defined ghost state, and a Rocq implementation. The essence of this contribution is a new Iris assertion for HDEAs (§2):

$$e \Downarrow v, \quad \textit{meaning "if we execute } e \textit{ in the current heap, then it terminates in the value } v\textit{"}.$$

The assertion $e \Downarrow v$ (read "$e$ evaluates to $v$") asserts the result of evaluating any deterministic, terminating, read-only expression $e$. It allows us to express HDEAs of traditional formulations of IDF such as "$\texttt{checksum(buf)} =_{\text{IDF}} \texttt{chk1}$" (from §1.1) as $\texttt{checksum(buf)} \Downarrow \texttt{chk1}$.

To introduce $e \Downarrow v$ to Iris and make effective use of it, we make several technical contributions:

**Unstable resources in Iris (§3).** In order to define $e \Downarrow v$, we first have to generalize the underlying model of Iris. We extend the notion of resource algebras of Iris to include *unstable resources*, and we revisit the definition of frame-preserving updates that is central to Iris.[2] The key new unstable resource that we define is the *unstable points-to* $\ell \mapsto_{\mathsf{u}} v$. We use it in the definition of evaluation $e \Downarrow v$ to temporarily capture information about the memory that $e$ accesses (see §3.1). Like a regular, stable points-to $\ell \mapsto v$, the unstable points-to $\ell \mapsto_{\mathsf{u}} v$ allows reading from location $\ell$. However, unlike $\ell \mapsto v$, it can be freely duplicated (*i.e.,* $\ell \mapsto_{\mathsf{u}} v \vdash \ell \mapsto_{\mathsf{u}} v * \ell \mapsto_{\mathsf{u}} v$), and it can co-exist with the regular points-to at the same time (*i.e.,* $\ell \mapsto v \dashv\vdash \ell \mapsto v * \ell \mapsto_{\mathsf{u}} v$).

**Program logic with HDEAs (§4).** On top of the adapted resource model, we then build a program logic $\{P\}\ e\ \{v.\ Q(v)\}$. In typical Iris fashion, it is an expressive, higher-order program logic with impredicative invariants, step-indexing, *etc.* However, we have to be careful! The catch of unstable resources like $\ell \mapsto_{\mathsf{u}} v$ is that not all assertions can be framed anymore (see §2). That is, traditionally in Iris (and SL in general), one uses the rule FRAME-IRIS:

$$\textsc{frame-iris}\ \frac{\{P\}\ e\ \{v.\ Q(v)\}}{\{P * R\}\ e\ \{v.\ Q(v) * R\}} \qquad \textsc{frame-daenerys}\ \frac{\{P\}\ e\ \{v.\ Q(v)\}}{\{\boxplus P * \boxplus R\}\ e\ \{v.\ \boxplus Q(v) * \boxplus R\}}$$

For a Hoare-triple $\{P\}\ e\ \{v.\ Q(v)\}$, it allows one to frame an arbitrary assertion $R$, knowing that it will not be invalidated by $e$. The frame rule is baked into Iris at its very core (see §3.1). However, in the presence of unstable resources like $\ell \mapsto_{\mathsf{u}} v$, this rule is no longer sound. For example, it would be unsound to frame $\ell \mapsto_{\mathsf{u}} 5$ around $\{\ell \mapsto 5\}\ \ell := 42\ \{\_.\ \ell \mapsto 42\}$, since $\ell \mapsto 42 * \ell \mapsto_{\mathsf{u}} 5$ is absurd. To recover framing, we introduce a new modality, the *frame modality* $\boxplus P$, and replace the frame

[2] Daenerys is not the first SL with unstable resources. In other logics [16, 19, 21], typically the update is taken as the primitive and stability is derived from it. We do the opposite: stability is the primitive and updates are derived. See §7 for a comparison.

EVAL-VAL
$v \Downarrow v$

EVAL-PURE
$e \to_{\text{pure}} e' * e' \Downarrow v \vdash e \Downarrow v$

EVAL-LOAD
$\ell \mapsto_q v \vdash !\,\ell \Downarrow v$

EVAL-CTX
$e \Downarrow v * K[v] \Downarrow w \vdash K[e] \Downarrow w$

EVAL-DET
$e \Downarrow v * e \Downarrow w \vdash v = w$

HOARE-EVAL
$\{P * e \Downarrow \_\}\ e\ \{v.\ P * e \Downarrow v\}$

PTS-FRAME
$\ell \mapsto_q v \vdash \boxplus \ell \mapsto_q v$

FRAME-ELIM
$\boxplus P \vdash P$

EVAL-DUPL
$$\frac{P \vdash e \Downarrow v \qquad P \vdash Q}{P \vdash e \Downarrow v * Q}$$

FRAME-EVALS
$$\frac{P \vdash e \Downarrow \_}{\boxplus P * e \Downarrow v \vdash \boxplus(P * e \Downarrow v)}$$

HOARE-FRAME
$$\frac{\{P\}\ e\ \{v.\ Q(v)\}}{\{\boxplus P * \boxplus R\}\ e\ \{v.\ \boxplus Q(v) * \boxplus R\}}$$

Fig. 2. A selection of core proof rules of Daenerys.

rule with FRAME-DAENERYS above. The frame modality $\boxplus P$ acts as a "gate keeper": it makes sure that we only frame assertions that do not depend on ownership of unstable resources.[3]

**Automation via almost-pure assertions (§5).** Having introduced $e \Downarrow v$ as a new Iris assertion, we then carve out a very useful fragment of *almost-pure assertions* out of Iris's propositions:

$$F, G : \mathit{hProp} ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathsf{u}} v \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid \exists x.\, F\,x \mid \forall x.\, F\,x \mid \cdots$$

It contains actually pure assertions $\phi$ and is closed under standard logical connectives. It also contains connectives such as $e \Downarrow v$ and $\ell \mapsto_{\mathsf{u}} v$ that can implicitly refer to the current heap.

We use this fragment to lay the groundwork for new automation for Iris: we show a correspondence between *hProp*-assertions (using HDEAs like $e \Downarrow v$) and standard first-order logic (agnostic about heaps and state). This connection allows us, for the first time, to automate parts of an Iris proof using an SMT solver. For example, we consider a polymorphic hashmap with an equality function $\mathsf{eq}$ and a hash-function $\mathsf{hash}$ (in §6). We express the key relationship between them as:

$$\forall x, y.\, \mathsf{eq}(x, y) \equiv \mathsf{true} \Rightarrow \mathsf{hash}(x) \equiv \mathsf{hash}(y) \quad \textit{where } e_1 \equiv e_2 \triangleq \exists v.\, e_1 \Downarrow v \wedge e_2 \Downarrow v \qquad (\textsc{eq-hash})$$

We then encode this condition for concrete instantiations of $\mathsf{eq}$ and $\mathsf{hash}$ into a first-order logic formula, (manually) query the SMT solver Z3 [18] on this formula, and assume in Rocq that it holds.

The use of an SMT solver means verification is not completely foundational: we trust Z3 to be sound w.r.t. standard first-order logic semantics. We do, however, show foundationally (Theorem 5.5 in §5.2) that our connection between *almost-pure assertions in Iris* (which use HDEAs to reason about memory) and *first-order logic* (which is agnostic about the heap) is sound.

**Case studies (§6).** We have applied Daenerys to several interesting case studies, demonstrating the benefits of combining IDF and Iris and of the SMT-based automation enabled by HDEAs.

Daenerys is fully mechanized in Rocq [51], extending the implementation of Iris [30, 31, 34] and the Iris Proof Mode [33, 35]. The Rocq development is provided as supplementary material [57].

## 2 Heap-Dependent Expression Assertions in Daenerys

In this section, we focus on the main HDEA of Daenerys, the evaluation assertion $e \Downarrow v$. We explain how it works (§2.1), how it integrates into the program logic (§2.2), and how it unlocks new automation for Iris by connecting to first-order logic (§2.3). Throughout, we use the rules in Fig. 2.

### 2.1 The Evaluation Assertion

Before we explain $e \Downarrow v$, let us first introduce the language $\lambda_{\mathsf{dyn}}$ that we will be working with.

[3] In IDF, soundness of the frame rule is typically ensured by requiring the assertions to be "self-framing". The frame modality $\boxplus P$ internalizes this notion of "self-framingness" into the logic in the form of a modality.

$$
\begin{array}{rrl}
\text{Values} & v, w & ::= () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \ell \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid (v, w) \mid \mathbf{rec}\, f(x) := e \mid \#[v_1, ..., v_n] \mid u \\
\text{Expressions} & e & ::= x \mid v \mid \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid e_1\, e_2 \mid (e_1, e_2) \mid e_1 \odot e_2 \mid \ominus e \mid \mathsf{CAS}(e_1, e_2, e_3) \\
& & \quad \mid \mathsf{ref}(e) \mid !\, e \mid e_1 := e_2 \mid \mathsf{free}(e) \mid \mathsf{fork}\, \{e\} \mid e_1[e_2] \mid e_1[e_2 \leftarrow e_3] \mid |e| \mid \cdots \\
\text{Eval. Contexts} & K & ::= \bullet \mid \mathbf{if}\, K\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid e_1\, K \mid K\, v_2 \mid (e_1, \bullet) \mid (\bullet, v_2) \mid e_1 \odot K \mid K \odot v_2 \mid \cdots
\end{array}
$$

Fig. 3. An excerpt of the language $\lambda_{\mathsf{dyn}}$.

**The language $\lambda_{\mathsf{dyn}}$.** We use an ML-like language $\lambda_{\mathsf{dyn}}$ (Fig. 3) with Booleans, integers, references, sums (via the constructors inl and inr), pairs, recursive functions, and fork-join concurrency. It extends Iris's default language HeapLang with vectors $\#[v_1, \ldots, v_n]$ (*i.e.,* immutable sequences of values for, *e.g.,* strings) and bitvectors $u$ (*i.e.,* fixed-size integers such as unsigned 64-bit integers). Notably, $\lambda_{\mathsf{dyn}}$ does not distinguish between commands and expressions: everything is an expression, including recursive functions $\mathbf{rec}\, f(x) := e$ and function application $e_1\, e_2$. For example, the iteration combinator below—applying a function f to integers in the range [n, m)—is a value expression:

$$\mathsf{iter} \triangleq \mathbf{rec}\, \mathsf{it}(\mathsf{n}, \mathsf{m}, \mathsf{s}, \mathsf{f}) := \mathbf{if}\, \mathsf{m} \le \mathsf{n}\, \mathbf{then}\, \mathsf{s}\, \mathbf{else}\, \mathsf{it}(\mathsf{n} + 1, \mathsf{m}, \mathsf{f}\, \mathsf{n}\, \mathsf{s}, \mathsf{f})$$

**Evaluation $e \Downarrow v$.** Let us now explain the evaluation assertion $e \Downarrow v$. It is a simple judgment for reasoning about terminating, deterministic, read-only expressions. Intuitively, $e \Downarrow v$ means "if we execute $e$ in a fragment[4] of the current heap, then it terminates in the value $v$". For example,

$$\ell \mapsto v_{\mathrm{vec}} \vdash e_{\mathrm{add}} \Downarrow 42 \quad \textit{where}\ \ e_{\mathrm{add}} \triangleq \mathsf{iter}(0, |!\,\ell|, 0, \lambda \mathsf{i}, \mathsf{s}.\ !\ell[\mathsf{i}] + \mathsf{s}) \ \ \textit{and}\ \ v_{\mathrm{vec}} \triangleq \#[13, 11, 6, 12],$$

means if $\ell$ stores the vector $v_{\mathrm{vec}}$ in memory, then $e_{\mathrm{add}}$ (which adds up the elements of the vector stored in $\ell$) evaluates to 42. To develop an intuition for $e \Downarrow v$ and illustrate how it works, we prove the entailment $\ell \mapsto v_{\mathrm{vec}} \vdash e_{\mathrm{add}} \Downarrow 42$ using the rules in Fig. 2: First, we focus on the subexpression $!\,\ell$ in evaluation position with EVAL-CTX for $K \triangleq \mathsf{iter}(0, |\bullet|, 0, \lambda \mathsf{i}, \mathsf{s}.\ !\ell[\mathsf{i}] + \mathsf{s})$. We can then justify the load with EVAL-LOAD, leaving us with $K[v_{\mathrm{vec}}] \Downarrow 42$. (We ask the reader for now to ignore the subscript on points-to assertions $\ell \mapsto_q v$ such as in EVAL-LOAD.) Next, we compute the vector length with a pure step $|v_{\mathrm{vec}}| \to_{\mathrm{pure}} 4$ with EVAL-PURE, leaving $\mathsf{iter}(0, 4, 0, \lambda \mathsf{i}, \mathsf{s}.\ !\ell[\mathsf{i}] + \mathsf{s}) \Downarrow 42$ to prove. We continue with pure steps and dereferencing $\ell$ until we reach $42 \Downarrow 42$, which holds by EVAL-VAL. (We will develop more automated approaches for reasoning about $e \Downarrow v$ in §5.)

What makes $e \Downarrow v$ special—particularly from an SL perspective—is that it does not consume any ownership of the locations that $e$ accesses. Traditionally, the separating conjunction $P * Q$ enforces that $P$ and $Q$ access *disjoint* parts of the heap (or more generally disjoint resources). However, for $e \Downarrow v$, we have $\ell \mapsto v_{\mathrm{vec}} \vdash \ell \mapsto v_{\mathrm{vec}} * e_{\mathrm{add}} \Downarrow 42$, yet clearly $e_{\mathrm{add}}$ accesses $\ell$. The key rule is EVAL-DUPL: when we prove $P \vdash e \Downarrow v * Q$, we do not have to split up the ownership of $P$ between $e \Downarrow v$ and $Q$—as would usually be the case. Instead, we can use $P$ for proving $e \Downarrow v$ *and* $Q$. More broadly, this means that $e \Downarrow v$ escapes the usually linear (or affine in Iris) resource management of separation logic (which makes it easier to reason about). We will see in §3 that the underlying reason why $e \Downarrow v$ enjoys this rule is that it is defined using *unstable points-tos* $\ell \mapsto_{\mathsf{u}} v$.

## 2.2 Evaluation and the Program Logic

We verify (effectful) programs using a program logic with Hoare triples $\{P\}\ e\ \{v.\ Q(v)\}$ (see §4). Let us now discuss how $e \Downarrow v$ integrates into it. For this, we return to the `checksum` example (from §1.1). Recall that our goal in this example is to avoid defining a mathematical representation *cs* of `checksum`. We start with a high-level proof sketch, this time using $e \Downarrow v$:

[4]As we will see in §3.1, $e \Downarrow v$ only depends on the fragment of the heap that is accessed by $e$, not the entire heap. As a result, $e \Downarrow v$ is not invalidated by modifications of the heap that are independent of the memory locations read by $e$.

```
6  let buf = produce_buffer() in   ■ {buf ↦ u⃗}
7  let chk1 = checksum(buf)   in   ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
8  read_only_client(buf);          ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
9  let chk2 = checksum(buf)   in   ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1 ∗ checksum(buf) ⇓ chk2}
10 assert(chk1 == chk2)
```

We obtain checksum(buf) ⇓ chk1 in Line 7 and checksum(buf) ⇓ chk2 in Line 9, and then thread them through to the assert in Line 10. We can prove that the assert succeeds, because $e \Downarrow v$ is deterministic (EVAL-DET), so chk1 and chk2 are equal at this point.

Two of these steps warrant a closer look. We discuss how we can obtain $e \Downarrow v$ for checksum (Lines 6–7) and how we can frame it past the read-only client (Lines 7–8).

**Connecting evaluation and the program logic.** To get from Line 6 to Line 7, we prove the Hoare triple $\{b \mapsto \vec{u}\}\ \texttt{checksum}(b)\ \{v.\ b \mapsto \vec{u} * \texttt{checksum}(b) \Downarrow v\}$. In general, we connect evaluation to Hoare triples with the rule HOARE-EVAL (Fig. 2). It allows one to prove a Hoare triple for $e$ if $e$ evaluates, written $e \Downarrow \_ \triangleq \exists v.\ e \Downarrow v$. To apply HOARE-EVAL here, it suffices to prove

$$b \mapsto_q \vec{u} \vdash \texttt{checksum}(b) \Downarrow \_ \qquad \text{(CHECKSUM-EVAL)}$$

meaning that if the buffer $b$ currently stores $\vec{u}$, then checksum($b$) will terminate in some value.

At first glance, this may seem like it requires us to verify checksum after all, even though the assertion (in Line 10) only requires that checksum is deterministic. Recall that providing a mathematical specification like *cs* for checksum is exactly the overhead that we are trying to avoid by using HDEAs. Fortunately, showing that a function deterministically computes *some result* is a weaker requirement than showing that it computes *a specific result*. It suffices for checksum to be safe, deterministic, and terminating—but a mathematical function *cs* on the buffer contents is not needed. We will see in §5.1 how we can exploit this relaxation by introducing a semantic type system, which will give us (in many cases) a simple way of proving $e \Downarrow \_$ via "typechecking" $e$.

**Framing.** As the final piece of the proof (Lines 7–8), let us turn our attention to *framing*. Typically, in Iris, once we proved a Hoare triple $\{P\}\ e\ \{v.\ Q(v)\}$, we can frame any assertion $R$ around it (see FRAME-IRIS in §1.2). However, the assertion $e \Downarrow v$ is special in that it *cannot* be framed on its own. For example, it would be unsound to frame $e_{\text{add}} \Downarrow 42$ around $\{\ell \mapsto v_{\text{vec}}\}\ \ell := \#[]\ \{\_.\ \ell \mapsto \#[]\}$, since the contents of $\ell$ change. Instead, to frame $e \Downarrow v$, we have to frame enough ownership alongside it to ensure that the result of $e$ does not change. For example, the ownership of $\ell \mapsto v_{\text{vec}}$ ensures that $e_{\text{add}}$ does not change, so we can frame $R \triangleq \ell \mapsto v_{\text{vec}} * e_{\text{add}} \Downarrow 42$ around every Hoare triple.

In our example (Lines 7–8), we must be careful not to frame buf $\mapsto \vec{u}$ around the read-only client, since the client *also* needs ownership of the buffer buf to justify reading from it. Thus, we *split* the the ownership of buf $\mapsto \vec{u}$ using *fractional permissions* [9, 10]: The *fractional points-to* $\ell \mapsto_q v$ generalizes $\ell \mapsto v$. For $q = 1$, it is the same as $\ell \mapsto v$, and for any fraction $0 < q < 1$, it allows reading but not writing. One can split and combine the points-to assertions based on their fractions (*i.e.*, $\ell \mapsto_{q+q'} v \dashv\vdash \ell \mapsto_q v * \ell \mapsto_{q'} v$), and two points-to assertions for the same location agree on the value stored there (*i.e.*, $\ell \mapsto_q v * \ell \mapsto_{q'} w \vdash v = w$). We use one half buf $\mapsto_{1/2} \vec{u}$ to frame checksum(buf) ⇓ chk1, and we give the other to the read-only client (still allowing reading), $\{\texttt{buf} \mapsto_{1/2} \vec{u}\}$ read_only_client(buf) $\{\_.\ \texttt{buf} \mapsto_{1/2} \_\}$.

**The frame modality $\boxplus P$.** IDF ensures soundness of the frame rule by ensuring that the framed assertion is "self-framing" (*i.e.*, contains non-zero ownership for each memory location it depends on). We internalize this notion into Daenerys with a revised frame rule, HOARE-FRAME, and a new modality, the *frame modality* $\boxplus P$. The latter is the "gate keeper" that ensures that we frame enough ownership such that $P$ will not be invalidated. To explain both, let us zoom in on the proof step around the read-only client (with two intermediate proof states added in ▸orange):

```
11 ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
12 ▶ {⊞buf ↦1/2 u⃗ ∗ ⊞(buf ↦1/2 u⃗ ∗ checksum(buf) ⇓ chk1)}
13 read_only_client(buf);
14 ▶ {⊞buf ↦1/2 _ ∗ ⊞(buf ↦1/2 u⃗ ∗ checksum(buf) ⇓ chk1)}
15 ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
```

We want to pass $P \triangleq \texttt{buf} \mapsto_{1/2} \vec{u}$ to the read-only client and frame $R \triangleq \texttt{buf} \mapsto_{1/2} \vec{u} * \texttt{checksum(buf)} \Downarrow \texttt{chk1}$. To do so, HOARE-FRAME asks us to split our precondition into two parts $P$ and $R$ and put both into a frame modality "$\boxplus$".[5] We do so in Lines 11–12: For $P$, we use that fractional ownership of a points-to assertion $\ell \mapsto_q v$ can always be put it into a frame modality (PTS-FRAME), because it precludes others from modifying $\ell$. For $R$, we exploit that we can frame $e \Downarrow v$ if we combine it with enough ownership to prove that $e$ evaluates (FRAME-EVALS). Since $R$ contains $\texttt{buf} \mapsto_{1/2} \vec{u}$, we can use CHECKSUM-EVAL to prove that `checksum(buf)` evaluates and obtain $\boxplus R$.

Finally, after the read-only client, in Lines 14–15, we have to establish the original postcondition. But this is easy! The frame modality $\boxplus P$ tells us, in particular, that $P$ holds, so we can just eliminate it (FRAME-ELIM). We can then re-assemble the full points-to assertion for the buffer from the two halves. This completes the last missing step in our example from the start of this subsection.

### 2.3 Connecting Evaluation and First-Order Logic via Almost-Pure Assertions

We will now show how to connect evaluation $e \Downarrow v$ to first-order logic to enable new automation via SMT solvers. As motivation, consider a small variation of the `checksum` example:

```
16 let buf = produce_buffer() in ■ {buf ↦ u⃗}
17 let chk = checksum(buf)    in ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk}
18 read_only_client(buf);       ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk}
19 assert(validate(buf, chk))   ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk ∗ validate(buf,chk) ⇓ true}
```

In this version, we call a function `validate` to validate the buffer `buf` against the checksum `chk`. We consider this version, because—unlike for `checksum` where it was enough to know that it computes *some* value—for `validate`, it actually matters *what* the function does. That is, to show that the assert in Line 19 succeeds, `validate` cannot be just any function. Instead, we need that *for any buffer* $b$, $\texttt{validate}(b, c)$ *returns true if* $c$ *is the result* $\texttt{checksum}(b)$.

**Almost-pure assertions.** Suppose `checksum` returns a 64-bit unsigned integer. Then we can make the desired relationship between `validate` and `checksum` formal in Daenerys as:

$$\forall b, c.\, \texttt{buffer}(b) \Rightarrow \texttt{u64}(c) \Rightarrow \texttt{checksum}(b) \Downarrow c \Rightarrow \texttt{validate}(b, c) \Downarrow \texttt{true} \qquad \text{(CHECK-VAL)}$$

where buffer is defined below and u64 ensures that $c$ is a 64-bit unsigned integer.

To state and prove properties relating HDEAs like CHECK-VAL, we use *almost-pure assertions* in Daenerys—assertions that largely behave pure (*i.e.,* non-linear), yet can also refer to the heap:

$$F, G : \mathit{hProp} ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathsf{u}} v \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid \exists x.\, F\,x \mid \forall x.\, F\,x \mid \cdots \quad \subseteq \quad \mathit{iProp}$$

They are a fragment of (our version of) Iris's propositions *iProp*, containing actually-pure assertions $\phi$,[6] evaluation assertions $e \Downarrow v$, *unstable* points-to assertions $\ell \mapsto_{\mathsf{u}} v$ (see §3), and standard logical connectives including impredicative, higher-order quantification (which gives rise to least- and greatest fixpoints). The unstable points-to allows one (together with $e \Downarrow v$) to constrain the current memory. For example, we define $\texttt{buffer}(b) \triangleq \exists \vec{u}.\, b \mapsto_{\mathsf{u}} \vec{u}$ to ensure that $b$ is a buffer in memory.

[5] The reader may wonder whether both frame modalities in the precondition of HOARE-FRAME are needed. The frame modality around $R$ ensures that $e$ does not invalidate $R$. We discuss the one around $P$ in §4.1.

[6] In traditional IDF terminology, $e \Downarrow v$ would be called pure. Here, we follow the Iris convention of only calling assertions $\phi$ pure if they are meta-level assertions (*i.e., Prop* in Rocq) and, hence, do not depend on the heap (unlike $e \Downarrow v$; see §3.1).

The *hProp*-fragment is quite expressive (see also §5): For example, we can use it to reason about evaluation $e \Downarrow v$ by restating the proof rules from Fig. 2 in *hProp* with $\ell \mapsto_{\mathsf{u}} v$ in place of $\ell \mapsto_q v$, conjunction in place of separating conjunction, and implication in place of entailment:

$$e \to_{\text{pure}} e' \land e' \Downarrow v \Rrightarrow e \Downarrow v \qquad \ell \mapsto_{\mathsf{u}} v \Rrightarrow !\,\ell \Downarrow v \qquad e \Downarrow v \land K[v] \Downarrow w \Rrightarrow K[e] \Downarrow w$$

Hence, one can prove, *e.g.*, $\ell \mapsto_{\mathsf{u}} v_{\text{vec}} \Rrightarrow e_{\text{add}} \Downarrow 42$ in *hProp* analogously to §2.1.

**First-order logic.** Suppose we want to prove CHECK-VAL for a concrete implementation of `validate`, such as `validate(buf,chk)=(checksum(buf) xor chk == 0)`. Does it satisfy the property CHECK-VAL (and, consequently, does the assert in Line 19 succeed)? The answer is yes!

To prove it, one option would be to roll up our sleeves and use the rules for $e \Downarrow v$ above. (In Rocq, we have instantiated the Iris proof mode [35] with *hProp* for such cases.) However, Daenerys also provides a second option: *solve the problem automatically in first-order logic.* If one squints a little, CHECK-VAL looks a lot like a formula in first-order logic: the functions correspond to first-order function symbols, the predicates to first-order sorts, and the evaluation $e \Downarrow v$ to equality. Following this analogy, CHECK-VAL could be restated in first-order logic (indicated in blue) as:

$$\dot\forall(\mathsf{b} : \mathsf{buffer}), (\mathsf{c} : \mathsf{bv}\,64).\ \mathsf{checksum}(\mathsf{b}) \doteq_{\mathsf{bv}\,64} \mathsf{c} \doteRightarrow \mathsf{validate}(\mathsf{b}, \mathsf{c}) \doteq_{\mathsf{bool}} \mathsf{true} \qquad \text{(CHECK-VAL-FO)}$$

This is basically how IDF-based verifiers like Viper [42] reason about HDEAs (although the details differ substantially; see §5.2). In Daenerys, we develop a *foundational* justification for this correspondence: we show that one can translate a first-order logic formula $\pi$ such as CHECK-VAL-FO (which has no concept of "memory") to an *hProp*-assertion $\langle \pi \rangle_{\mathsf{F}}$ such as CHECK-VAL (which refers to the current memory via $\ell \mapsto_{\mathsf{u}} v$ and $e \Downarrow v$) such that if $\pi$ holds, then we can assume $\langle \pi \rangle_{\mathsf{F}}$ in Iris.

**SMT solvers.** The main use-case for this connection is laying the groundwork for connecting Iris to SMT solvers such as Z3 [18] or CVC5 [6] to benefit from their built-in automation. Of course, SMT solvers are not foundational, and we by no means attempt to verify an SMT solver. Instead, Daenerys provides the assurance that *if* $\pi$ holds in first-order logic—the language of SMT solvers—then the *hProp*-assertion $\langle \pi \rangle_{\mathsf{F}}$—indirectly referring to the heap—can be soundly used in Iris. For example, for assert, we derive the following proof rule (from our generic result, Theorem 5.5):

$$\frac{\text{HOARE-ASSERT}\qquad \vDash \pi \qquad P \vdash e \Downarrow \_ \qquad P * \langle \pi \rangle_{\mathsf{F}} \vdash e \Downarrow \mathsf{true}}{\{P\}\ \mathsf{assert}(e)\ \{\_.\ P\}}$$

where $\vDash \pi$ means $\pi$ is provable in first-order logic with knowledge of, *e.g.*, numbers and bitvectors. HOARE-ASSERT means that an assert $e$ succeeds if (1) $\pi$ holds in first-order logic, (2) $P$ suffices to prove that $e$ will evaluate, and (3) *assuming* $\langle \pi \rangle_{\mathsf{F}}$, one can prove that $e$ evaluates to true.

For example, given the implementation of `validate`, Z3 can prove CHECK-VAL-FO automatically, because it knows that $u$ `xor` $u = 0$ for any 64-bit unsigned integer $u$. Thus, if we trust Z3 to be sound w.r.t. to standard first-order logic semantics, we can verify the assert in Line 19—even though it uses heap-accessing functions like `validate`—without ever (1) specifying `checksum` as a mathematical function *cs* and (2) reasoning about bitvector arithmetic in Iris.

## 3 Destabilizing the Foundations of Iris

Having used the evaluation assertion $e \Downarrow v$ (in §2), let us now define it. To do so, we have to go down to the core of Iris and change its underlying notion of *resources*. Famously almost everything in Iris boils down to resources, including *the heap* with $\ell \mapsto_q v$, but also *state transition systems* [31], *invariants* [29], *refinements* [25, 62], *time complexity* [39], and even to some extent *step-indexing* [56]. In Daenerys, we generalize Iris's resources one step further by introducing *unstable resources*.

### 3.1 Unstable Resources

To define $e \Downarrow v$, we need a new resource assertion, the *unstable points-to* $\ell \mapsto_{\mathsf{u}} v$. Like a normal points-to $\ell \mapsto_q v$, it asserts the value of $\ell$ in the current heap. But unlike $\ell \mapsto_q v$, (1) it can co-exist with the full ownership of $\ell$ in the sense that $\ell \mapsto_q v * v = w \dashv\vdash \ell \mapsto_q v * \ell \mapsto_{\mathsf{u}} w$ holds, including for $q = 1$ and (2) it does not prevent updates to $\ell$. As we will see below, this means it goes beyond the resource model of Iris. But before we get there, let us first use it to define $e \Downarrow v$:

$$e \Downarrow v \triangleq \exists h.\, (e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h) * (*_{\ell \mapsto w \in h}\, \ell \mapsto_{\mathsf{u}} w) \qquad \text{(EVAL-DEF)}$$

That is, $e$ evaluates to $v$ if (1) there is a heap fragment $h$ in which $e$ deterministically steps to $v$ in the operational semantics,[7] written $(e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h)$, and (2) we have unstable points-to assertions $\ell \mapsto_{\mathsf{u}} w$ for all entries in $h$, which ensures that $e$ evaluates to $v$ in (a fragment of) *the current heap*.

**Framing by construction.** Let us now see why $\ell \mapsto_{\mathsf{u}} v$ requires us to modify the model of Iris. The issue is that $\ell \mapsto_{\mathsf{u}} v$ cannot soundly be framed around code that modifies $\ell$, yet—usually—Iris "bakes in" framing at its very core. More specifically, as part of its design philosophy [27, §7.1], Iris makes framing *the defining feature* of which resource updates it permits via its so-called "frame preserving update" $a \rightsquigarrow b$. We will now explore the essence of the problem and how we resolve it.

In Iris, resource assertions desugar into the *ownership assertion* $\mathsf{Own}\,(a : R)$. It carries a *resource* $a$ drawn from a *resource algebra* $R$. The resource describes "what" we own, and the algebra determines the logic-level rules for the resource (*e.g.,* how separating conjunction affects it). To illustrate the issue with Iris's original resources, we focus on the *exclusive resource algebra* $\mathit{Ex}(\mathbb{N})$. It essentially has only a single element, $\mathsf{ex}(n)$, carrying full ownership of the number $n$. It adheres to the rules:

| EX-VALID | EX-EXCL | EX-UPD |
|---|---|---|
| $\mathsf{ex}(n) \in \mathcal{V}$ | $\mathsf{ex}(n) \cdot \mathsf{ex}(m) \notin \mathcal{V}$ | $\mathsf{ex}(n) \rightsquigarrow \mathsf{ex}(m)$ |

They use three resource algebra connectives, which we have not yet introduced. The *validity predicate* $\mathcal{V}$ rules out invalid combinations of resources. For example, the resource $\mathsf{ex}(n)$ on its own is valid (EX-VALID), but if we compose it with another copy of $\mathsf{ex}(\_)$ it becomes invalid (EX-EXCL). The *resource composition* $a \cdot b$ underlies the separating conjunction (*i.e.,* $\mathsf{Own}\,(a_1) * \mathsf{Own}\,(a_2) \dashv\vdash \mathsf{Own}\,(a_1 \cdot a_2)$). Thus, the rule EX-EXCL tells us that $\mathsf{ex}(n)$ carries full ownership: it cannot exist at the same time as another copy $\mathsf{ex}(m)$. The *frame preserving update* $a \rightsquigarrow b$ tells us how we can update resources: $\mathsf{ex}(n)$ carries full ownership, so we can update it to $\mathsf{ex}(m)$ for any $m$ (EX-UPD). As an analogy, the reader can think of $\mathsf{ex}(n)$ as the resource underlying $\ell \mapsto_1 n$ for a fixed $\ell$.

To illustrate why we need to update the foundations of Iris, we will now show that unstable points-to assertions are incompatible with its model. Consider an extension of the exclusive resource algebra with a resource $\mathsf{tmp}(n)$, the analog of $\ell \mapsto_{\mathsf{u}} n$. This resource co-exists with $\mathsf{ex}(n)$ yet—crucially—should still allow updating ex via EX-UPD. Formally, we want that $\mathsf{ex}(n) = \mathsf{ex}(n) \cdot \mathsf{tmp}(n)$ (*i.e.,* we can always create a temporary copy) and $\mathsf{tmp}(n) \cdot \mathsf{ex}(m) \in \mathcal{V} \Rightarrow n = m$ (*i.e.,* the two agree on the current value). Unfortunately, if these two hold, then EX-UPD *is no longer true.* To understand why, we have to take a closer look at the frame-preserving update $a \rightsquigarrow b$. Its defining characteristic is that it preserves all valid frames $a_f$:

$$a \rightsquigarrow b \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow (b \cdot a_f) \in \mathcal{V}$$

This breaks EX-UPD in the presence of $\mathsf{tmp}(n)$. In EX-UPD, we have $a = \mathsf{ex}(n)$. This makes the resource $a_f = \mathsf{tmp}(n)$ a valid frame of $a$. But for $n \neq m$, the resource $a_f$ is not a valid frame of $b = \mathsf{ex}(m)$ (*i.e.,* $\mathsf{tmp}(n) \cdot \mathsf{ex}(m) \notin \mathcal{V}$ for $n \neq m$). Thus, there are no more updates of ex as soon as we add $\mathsf{tmp}(n)$.

[7] Concretely, the relation $(e, h) \rightsquigarrow_{\mathrm{det}} (e', h)$ restricts the operational semantics of $\lambda_{\mathrm{dyn}}$ $(e, h) \rightsquigarrow (e', h', es)$ to those steps that (1) do not change the heap $h$ and (2) do not fork any additional threads. They are deterministic in $\lambda_{\mathrm{dyn}}$ (and HeapLang).

UPD-OWN
$a \rightsquigarrow_{\mathsf{st}} b * \mathsf{Own}(a) \vdash \Rrightarrow_{\mathsf{st}} \mathsf{Own}(b)$

UPD-FRAME
$(\boxplus P) * (\Rrightarrow_{\mathsf{st}} Q) \vdash \Rrightarrow_{\mathsf{st}} (\boxplus P) * Q$

FRAME-PERS
$\square P \vdash \boxplus P$

FRAME-OWN
$\mathsf{Own}(a) \vdash \boxplus \mathsf{Own}(|a|_{\mathsf{st}})$

UNSTABLE-DUPL
$(\circledast P) \wedge Q \dashv\vdash (\circledast P) * Q$

UNSTABLE-IMPL
$\circledast P \Rightarrow \circledast Q \vdash \circledast(\circledast P \Rightarrow Q)$

Fig. 4. A selection of rules for the update modality $\Rrightarrow_{\mathsf{st}} P$, frame modality $\boxplus P$, and unstable modality $\circledast P$.

**Unstable resources and stable updates.** This puts us in a pickle! How can we have both $\mathsf{tmp}(n)$ and EX-UPD? The issue is that $a \rightsquigarrow b$ preserves *too many* frames. Intuitively, we should be allowed to update $\mathsf{ex}(n)$ to $\mathsf{ex}(m)$, and the update should invalidate any temporary copies $\mathsf{tmp}(n)$ rather than preserve them. To realize this intuition, we extend Iris's resource algebras with *unstable resources*. More specifically, we add two projections $|a|_{\mathsf{st}}$ and $|a|_{\mathsf{unst}}$ such that $a = |a|_{\mathsf{st}} \cdot |a|_{\mathsf{unst}}$. The first projection, $|a|_{\mathsf{st}}$, yields the stable part of a resource. (Usually, in Iris, all resources are stable.) The second projection, $|a|_{\mathsf{unst}}$, yields the unstable part. We then define a new *stable update* $a \rightsquigarrow_{\mathsf{st}} b$, which only preserves the stable parts of the frame resource $a_f$ (difference highlighted in violet):

$$a \rightsquigarrow_{\mathsf{st}} b \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow (b \cdot |a_f|_{\mathsf{st}}) \in \mathcal{V}$$

In the context of our example, $\mathsf{tmp}(n)$ represents temporary information about the value of $\mathsf{ex}(n)$. Thus, we define $|\mathsf{tmp}(n)|_{\mathsf{st}} \triangleq \epsilon$ (*i.e.,* as the unit of $\cdot$), making it an unstable resource. Hence, for $a_f = \mathsf{tmp}(n)$, the stable update $\rightsquigarrow_{\mathsf{st}}$ *erases* $\mathsf{tmp}(n)$ *from the frame*, such that EX-UPD holds for $\rightsquigarrow_{\mathsf{st}}$.

The frame preserving update $a \rightsquigarrow b$ is a corner stone of resource reasoning in Iris: it underlies Iris's *update modality* $\Rrightarrow P$, which is used pervasively (*e.g.,* to define invariants, later credits, and the weakest precondition). Thus, the introduction of unstable resources (and of $a \rightsquigarrow_{\mathsf{st}} b$ specifically) ripples through all layers of Iris. We will now discuss how unstable resources give rise to new modalities (§3.2), affect resource algebras (§3.3), and alter the program logic (§4).

### 3.2 Extending the Base Logic

We start by lifting the new resource algebra operations (*i.e.,* $a \rightsquigarrow_{\mathsf{st}} b$ and the projections $|a|_{\mathsf{st}}$ and $|a|_{\mathsf{unst}}$) to the assertion level. We define three new modalities in Daenerys:

$$(\boxplus P)(a, i) \triangleq P(|a|_{\mathsf{st}}, i) \qquad (\circledast P)(a, i) \triangleq P(|a|_{\mathsf{unst}}, i)$$

$$(\Rrightarrow_{\mathsf{st}} P)(a, i) \triangleq \forall a_f.\, a \cdot a_f \in \mathcal{V} \Rightarrow \exists b.\, b \cdot |a_f|_{\mathsf{st}} \in \mathcal{V} \wedge P(b, i)$$

In the model of Iris, every proposition $P$ is a predicate over a resource $a$ and a step-index $i$ (which the reader can ignore).[8] In short, the *frame modality* $\boxplus P$ must be proven using only the stable parts of the current resource, and analogously the *unstable modality* $\circledast P$ only using the unstable parts. The *stable update modality* reflects $a \rightsquigarrow_{\mathsf{st}} b$ into the logic (analogously to how $\Rrightarrow P$ reflects $a \rightsquigarrow b$ in Iris). We take a closer look at each one and discuss their key proof rules, depicted in Fig. 4.

The *stable update modality* $\Rrightarrow_{\mathsf{st}}$ allows us to perform stable updates $a \rightsquigarrow_{\mathsf{st}} b$ on resources (UPD-OWN). For $\Rrightarrow_{\mathsf{st}}$ to be a usable update modality, it is important that the definition of $a \rightsquigarrow_{\mathsf{st}} b$ ensures that $\Rrightarrow_{\mathsf{st}}$ retains (almost; explained below) the same compositionality as the normal update $\Rrightarrow$ (*e.g.,* it is also a monad). We even have $\Rrightarrow P \vdash \Rrightarrow_{\mathsf{st}} P$ (since $a \rightsquigarrow b$ implies $a \rightsquigarrow_{\mathsf{st}} b$). Thus, existing resource algebra constructions such as user-defined ghost state can still be reused.

The *frame modality* $\boxplus$ illustrates the key difference between the original update $\Rrightarrow$ and the new update $\Rrightarrow_{\mathsf{st}}$, namely we can only *frame* assertions that are guarded by a frame modality (UPD-FRAME).

[8] Technically, we extend the notion of Iris's step-indexed resources, so-called "cameras". For the sake of simplicity, in this presentation, we focus on simpler resources without step-indexing.

RA-DECOMPOSE
$|a|_{\mathrm{st}} \cdot |a|_{\mathrm{unst}} = a$

RA-STABLE-IDEMP
$||a|_{\mathrm{st}}|_{\mathrm{st}} = |a|_{\mathrm{st}}$

RA-STABLE-DISTR
$|a \cdot b|_{\mathrm{st}} = |a|_{\mathrm{st}} \cdot |b|_{\mathrm{st}}$

RA-CORE-STABLE
$||a|_{\mathrm{core}}|_{\mathrm{st}} = |a|_{\mathrm{core}}$

RA-UNSTABLE-DUPL
$|a|_{\mathrm{unst}} \cdot a = a$

RA-UNSTABLE-IDEMP
$||a|_{\mathrm{unst}}|_{\mathrm{unst}} = |a|_{\mathrm{unst}}$

RA-UNSTABLE-MONO
$a \preccurlyeq b \Rightarrow |a|_{\mathrm{unst}} \preccurlyeq |b|_{\mathrm{unst}}$

RA-UNSTABLE-FLIP
$|a \cdot |b|_{\mathrm{unst}}|_{\mathrm{unst}} = ||a|_{\mathrm{unst}} \cdot b|_{\mathrm{unst}}$

RA-UNSTABLE-EXTENSION
$a \in \mathcal{V} \Rightarrow |a|_{\mathrm{unst}} \cdot b \in \mathcal{V} \Rightarrow a \cdot |b|_{\mathrm{unst}} \in \mathcal{V}$

Fig. 5. The additional axioms of partially stable resource algebras $R = (M, \mathcal{V}, \cdot, \varepsilon, |\_|_{\mathrm{core}}, |\_|_{\mathrm{st}}, |\_|_{\mathrm{unst}})$, where we abbreviate $a \preccurlyeq b \triangleq \exists c.\, a \cdot c = b$.

This rule holds without $\boxplus$ for $\Mapsto$ in Iris and is the basis for Iris's frame rule (FRAME-IRIS in §1.2). By adding frame modalities to it here, we ensure that only stable resources can be framed around $\Mapsto_{\mathtt{st}}$ (which erases the unstable parts of the frame). Iris's persistent assertions $\Box P$ are always frameable (FRAME-PERS), and if we own a resource, we frameably own its stable part (FRAME-OWN).

Lastly, the *unstable modality* $\divideontimes$ reflects a key property of unstable resources into the logic that we have not discussed yet: *they are duplicable*. More specifically, we have $a = a \cdot |a|_{\mathrm{unst}}$. The result is that when we prove an unstable assertion $\divideontimes P$, intuitively, we do not have to give up any ownership to do so. Formally, it means that the rule UNSTABLE-DUPL holds, which says that ordinary conjunction and separating conjunction coincide for unstable assertions.[9] It is the basis for $e \Downarrow v$ seemingly not consuming ownership (EVAL-DUPL in Fig. 2), and it holds for all *almost-pure assertions* (see §2.2):

Lemma 3.1. *For every almost-pure assertion $F$, we have $F \vdash \divideontimes F$, and hence, $F \wedge P \dashv\vdash F * P$*

Thus, whenever we prove an *hProp*-assertion, we can keep all of our ownership. For most logical connectives in *hProp*, proving this property is straightforward. However, for implication $F \Rightarrow G$, it is actually nontrivial (and yet implication is essential for the first-order logic connection in §5). The trick to get it for implication is the rule UNSTABLE-IMPL (which we justify with two dedicated axioms about resources in our definition of resource algebras; see §3.3).

### 3.3 Resource Algebras with Unstable Elements

Let us now turn to the resource algebras underlying the logic. In Daenerys, we introduce *partially stable resource algebras* $R = (M, \cdot, \varepsilon, |\_|_{\mathrm{core}}, \mathcal{V}, |\_|_{\mathrm{st}}, |\_|_{\mathrm{unst}})$, consisting of (1) a carrier set $M$; (2) a composition $a \cdot b$ for $a, b \in M$; (3) a unit $\varepsilon$; (4) a core projection $|a|_{\mathrm{core}}$ (for persistency in Iris [30]); (5) a validity predicate $\mathcal{V}$; (6) a stable projection $|\_|_{\mathrm{st}}$, and (7) an unstable projection $|\_|_{\mathrm{unst}}$. They extend Iris's *unital resource algebras* $(M, \cdot, \varepsilon, |\_|_{\mathrm{core}}, \mathcal{V})$ by the projections $|\_|_{\mathrm{st}}$ and $|\_|_{\mathrm{unst}}$.[10]

All existing rules still apply. We focus on the new rules governing the new projections, depicted in Fig. 5. A resource decomposes into its stable and unstable part (RA-DECOMPOSE). The stable projection is idempotent (RA-STABLE-IDEMP) and distributes over composition (RA-STABLE-DISTR). The core is always stable (RA-CORE-STABLE), which ensures that persistent assertions are frameable (FRAME-PERS). The unstable projection yields a duplicable part of a resource (RA-UNSTABLE-DUPL), justifying the key rule UNSTABLE-DUPL. The projection is idempotent (RA-UNSTABLE-IDEMP) and monotone (RA-UNSTABLE-MONO). Finally, we have RA-UNSTABLE-FLIP and RA-UNSTABLE-EXTENSION to support the

[9] Readers familiar with Iris might wonder about the relationship between persistency $\Box P$ (backed by $|\_|_{\mathrm{core}}$) and unstable propositions $\divideontimes P$ (backed by $|\_|_{\mathrm{unst}}$): unstable resources can describe larger parts of a resource, since they are only temporary whereas persistent propositions are stable (FRAME-PERS). For example, we have $|\mathrm{ex}(n)|_{\mathrm{core}} = \epsilon$ and $|\mathrm{ex}(n)|_{\mathrm{unst}} = \mathrm{tmp}(n)$.
[10] Restricting to *unitial* resource algebras is not a problem in practice as non-unitial resource algebras can be turned into unitial resource algebras via the standard option resource algebra.

HOARE-LOAD
$\{P * \ell \mapsto_{\mathsf{u}} v\}\ !\ell\ \{w.\ P * v = w\}$

HOARE-STORE
$\{\ell \mapsto_1 v\}\ \ell \leftarrow w\ \{\_.\ \ell \mapsto_1 w\}$

HOARE-EVAL
$\{P * e \Downarrow \_\}\ e\ \{v.\ P * e \Downarrow v\}$

HOARE-LET
$$\frac{\{P\}\ e_1\ \{v.\ Q(v)\} \qquad \forall v.\ \{Q(v)\}\ e_2[v/x]\ \{w.\ R(w)\}}{\{P\}\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \{w.\ R(w)\}}$$

HOARE-FRAME
$$\frac{\{P\}\ e\ \{v.\ Q(v)\}}{\{\boxplus P * \boxplus R\}\ e\ \{v.\ \boxplus Q(v) * \boxplus R\}}$$

HOARE-FORK
$$\frac{\{P\}\ e\ \{\_.\ \mathsf{True}\}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{\_.\ \mathsf{True}\}}$$

HOARE-INV
$$\frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * \boxplus Q(v)\} \qquad R \vdash \boxplus R \qquad \mathrm{atomic}(e)}{\{P * \boxed{R}\}\ e\ \{v.\ Q(v)\}}$$

Fig. 6. A selection of rules for the program logic of $\lambda_{\mathsf{dyn}}$.

"unstable implication rule" UNSTABLE-IMPL. They arise, because Iris's implication $P \Rightarrow Q$ is up-closed with respect to larger resources, which complicates commuting $\divideontimes$ in the rule UNSTABLE-IMPL.

The points-tos $\ell \mapsto_{\mathsf{u}} v$ and $\ell \mapsto_q v$ are effectively pointwise per-location liftings of $\mathsf{tmp}(n)$ and $\mathsf{ex}(n)$ (generalized to fractions). As usual for Iris, we factor their definition through several reusable combinators—including new ones for unstable resources—discussed in the appendix [57, §A]. Moreover, this extension of resource algebras is backwards compatible: Regular unital resource algebras from Iris can be embedded into partially stable resource algebras by picking the stable projection as the identity and the unstable projection as unit.

## 4 The Program Logic

Having generalized the resource model of Iris (§3), let us now use it to obtain a program logic. We first focus on the concrete program logic for $\lambda_{\mathsf{dyn}}$ from §2 (§4.1), and then we show how one can obtain such program logics via our adaptation of Iris's language-generic *weakest precondition* (§4.2).

### 4.1 The $\lambda_{\mathsf{dyn}}$ Program Logic

A selection of the rules of the $\lambda_{\mathsf{dyn}}$-program logic are depicted in Fig. 6. (We omit standard Iris rules, *e.g.,* for evaluating pure expressions, consequence, binding subexpressions, and recursion.) The rule for storing (HOARE-STORE) is standard. However, the rule for loads (HOARE-LOAD) uses the unstable points-to $\ell \mapsto_{\mathsf{u}} v$ instead of the fractional points-to $\ell \mapsto_q v$. This makes HOARE-LOAD strictly stronger, because one can obtain the unstable points-to from the regular one (see §3). It also allows one to frame arbitrary assertions $P$ without the frame modality, since $!\,\ell$ does not modify memory.

**Heap-dependent expression assertions.** The main effect of supporting HDEAs are the rules HOARE-FRAME and HOARE-EVAL, which we have already encountered in §2. Let us now discuss the frame modalities in HOARE-FRAME. As we have seen in §2.2, $R$ must be under a frame modality since otherwise one could frame unstable assertions around an expression that invalidates these assertions. The frame modalities in the postcondition only strengthen HOARE-FRAME and can always be eliminated with FRAME-ELIM (in Fig. 2). To understand why $P$ must be under a frame modality, let us first look at HOARE-LET and HOARE-FORK. These rules look like the standard Hoare rules and do *not* contain any frame modalities. This might be surprising since a concurrent thread could potentially invalidate their preconditions. The reason that this cannot happen (and thus no frame modalities are required in these rules) is that Hoare triples implicitly maintain that their pre- and postconditions are always frameable. The price we have to pay for this is that we need to prove that $P$ in HOARE-FRAME is frameable such that we can use it as the precondition of $e$ in the premise.

**Step-indexing and invariants.** Iris provides several advanced features that make it very expressive: *step-indexing* [1, 3] with the later modality $\triangleright P$ [4], *impredicative invariants* $\boxed{P}^{\mathcal{N}}$ [58], *later credits* [56], and *user-defined resources* [30], among others. Daenerys supports all of these as well and thus inherits the expressivity of Iris. Some features like step-indexing or later credits are largely orthogonal to the introduction of unstable assertions. However, supporting impredicative invariants is more subtle. (User-defined resources boil down to resource algebras, discussed in §3.)

Impredicative invariants $\boxed{P}^{\mathcal{N}}$ are how Iris shares resources between threads. What makes them *impredicative*—and special compared to most other separation logics—is that they can contain an arbitrary assertion, including other invariants or Hoare triples themselves. To access such an invariant, we can open it with HOARE-INV[11] around an atomic expression $e$ (*e.g.,* a load or a store). We get to assume the contents of the invariant guarded by a later modality, and we have to restore the invariant again after executing $e$. So far, this is standard for Iris. The only user-facing effect that the presence of *unstable resources* in the underlying model of Daenerys has on this rule is that (1) we must prove that the invariant $R$ is frameable (*i.e.,* we prove $R \vdash \boxplus R$) and (2) we may use only stable resources to prove the postcondition $Q$. The former is important because an invariant storing an unstable assertion without the ownership to stabilize it could be broken when the program invalidates the unstable assertion. The latter is important since the postcondition must not be allowed to take unstable assertions out of the invariant without the ownership that stabilizes them.

### 4.2 The Language-Generic Weakest Precondition

Let us now turn to how we define the Hoare triples in §4.1. In typical Iris fashion, we do so via a weakest precondition $\mathrm{wp}\ e\ \{v.\ Q(v)\}$ (adapted to Daenerys). Concretely, we define:

$$\{P\}\ e\ \{v.\ Q(v)\} \triangleq \square(\boxplus P \mathrel{-\!\!*} \mathrm{wp}\ e\ \{v.\ \boxplus Q(v)\})$$

That is, $\{P\}\ e\ \{v.\ Q(v)\}$ holds if we can show that the pre $P$ implies the weakest precondition of $e$ for post $Q$. (Iris's persistency modality $\square$ allows one to reuse a proven Hoare-triple multiple times.) The two frame modalities ensure that the pre $P$ and post $Q$ are frameable (as discussed in §4.1).

The weakest precondition is then defined in a language-generic fashion over a small-step relation $(e, h) \to (e', h', es)$ as follows:[12]

$$\begin{aligned}
&\mathrm{wp}\ v\ \{v.\ Q(v)\} \triangleq \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} \Rrightarrow_{\mathtt{st}} \mathrm{SI}(h) * Q(v)\\
&\mathrm{wp}\ e\ \{v.\ Q(v)\} \triangleq \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} \Rrightarrow_{\mathtt{st}} (\exists e', h', es.\ (e, h) \to (e', h', es)) \qquad \text{if } e \notin \mathit{Val}\\
&\quad * \ \forall e', h', es.\ (e, h) \to (e', h', es) \mathrel{-\!\!*} \triangleright \Rrightarrow_{\mathtt{st}} (\mathrm{SI}(h') * \boxplus \mathrm{wp}\ e'\ \{v.\ Q(v)\} * *_{e \in es} \boxplus \mathrm{wp}\ e\ \{_.\ \mathrm{True}\})
\end{aligned}$$

This definition is a variation of Iris's standard weakest precondition. It has two cases: In the value case, we assume the state interpretation $\mathrm{SI}(h)$ (tying the current heap $h$ to resources like $\ell \mapsto_q w$) and prove the postcondition $Q$ (after an update). In the case where $e$ is not a value, we assume the state interpretation and, after an update, prove that $e$ can make progress in the current heap. Then, we show that for any successor expression $e'$, heap $h'$, and forked-off threads $es$, we can reestablish the state interpretation and prove weakest preconditions for $e'$ and the forked-off threads.

In this definition, we reap the fruits of our more general model (in §3). We can support HDEAs simply by using the new modalities of Daenerys (highlighted in violet): we use stable updates $\Rrightarrow_{\mathtt{st}}$ in the places where Iris would traditionally use its frame preserving update $\Rrightarrow P$. Moreover, we add the frame modality $\boxplus$ for the successor expression $e'$ and the forked-off threads, because in a

[11] This rule is simplified to omit masks and namespaces $\mathcal{N}$ that Iris and Daenerys use to prevent reentrancy.

[12] To focus on the key parts of this definition (and the changes over Iris), this weakest pre is strongly simplified from the Rocq version, omitting, *e.g.,* fancy updates with masks for invariants, later credits, multiple laters per step, and observations.

$$\textsc{type-lam}\ \frac{\Gamma, x : \tau \vDash e : \sigma}{\Gamma \vDash \lambda x.\, e : \tau \to \sigma} \qquad \textsc{type-app}\ \frac{\Gamma \vDash e_1 : \tau \to \sigma \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash e_1\, e_2 : \sigma} \qquad \textsc{type-var}\ \frac{x : \tau \in \Gamma}{\Gamma \vDash x : \tau} \qquad \textsc{type-bitvec-xor}\ \frac{\Gamma \vDash e_1 : \mathsf{bv}\, n \qquad \Gamma \vDash e_2 : \mathsf{bv}\, n}{\Gamma \vDash e_1\ \mathsf{xor}\ e_2 : \mathsf{bv}\, n}$$

$$\textsc{type-if}\ \frac{\Gamma \vDash e : \mathsf{bool} \qquad \Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau} \qquad \textsc{type-nthopt}\ \emptyset \vDash \mathsf{nthopt} : \mathsf{buf}\, \tau \times \mathsf{int} \to \mathsf{option}\, \tau$$

Fig. 7. A selection of typing rules for $\lambda_{\mathsf{dyn}}$.

concurrent setting, they can be executed concurrently, so we must ensure that, *e.g.,* the verification of $e'$ does not rely on any unstable knowledge that could be invalidated by another thread.

## 5 Unlocking Automation with Almost-Pure Assertions and SMT Solvers

Recall the fragment of *almost-pure assertions* (from §2.3), containing pure assertions, evaluation, the unstable points-to assertion, and standard logical connectives:

$$F, G : \mathit{hProp} ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathsf{u}} v \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid \exists x.\, F\, x \mid \forall x.\, F\, x \mid \cdots \quad \subseteq \quad \mathit{iProp}$$

In this section, we use it to develop two automatable proof techniques for core aspects of Daenerys: First, we develop a semantic type system (§5.1) to streamline stabilizing $e \Downarrow v$ (*i.e.,* to move it inside a frame modality $\boxplus$; see §2.2). Then, we develop a correspondence between first-order logic and *hProp*-assertions (§5.2) to automate reasoning about program expressions in *hProp*-assertions using SMT solvers (see §2.3). For the correspondence, we will reuse the type system: it will bridge the gap between program functions $\mathsf{f}$ in $\lambda_{\mathsf{dyn}}$ and well-typed first-order functions $\mathsf{f}$ (see Lemma 5.6).

### 5.1 The Semantic Type System

Recall (from §2.2) that to stabilize $e \Downarrow v$, we must frame enough ownership $P$ alongside it to ensure that the result of $e$ does not change (FRAME-EVALS in Fig. 2). Formally, the side condition that arises is $P \vdash e \Downarrow \_$, where $e \Downarrow \_ \triangleq \exists v.\, e \Downarrow v$. It implicitly means that $P$ constrains enough of the heap to ensure that $e$ safely terminates in some value (considering the definition of $e \Downarrow v$ in EVAL-DEF). To simplify proving it, we now introduce a semantic type system $\Gamma \vDash e : \tau$ that satisfies the following property:

Lemma 5.1. $(\emptyset \vDash e : \tau) \vdash e \Downarrow \_$, *meaning closed, well-typed expressions safely evaluate.*

The type system is an ML-style type system, with typing rules for the simply-typed lambda calculus extended with standard data types (*e.g.,* integers, bitvectors of constant size $n$, sums, pairs, vectors, buffers, *etc.*). A selection of typing rules is depicted in Fig. 7. For example, we can use TYPE-LAM to type $\lambda$-functions, TYPE-APP for function application, and TYPE-VAR for variables.

**Safe termination and limitations of the type system.** The main purpose of the type system $\Gamma \vDash e : \tau$ is to streamline proving $e \Downarrow \_$ via a set of simple, automatable ML-style typing rules. Since $e \Downarrow \_$ holds only for deterministic, side-effect free expressions $e$, the type system has no rules for expressions that cause side effects (*e.g.,* heap updates or forking threads) or for non-deterministic expressions (*e.g.,* allocation). In addition, since the rules are supposed to be easily automatable, there are expressions $e$ that evaluate, meaning $e \Downarrow \_$, but which cannot be type checked by applying typing rules. One such example is $\textbf{if}\ 1 < 2\ \textbf{then}\ 42\ \textbf{else}$ assert (false), because while it is safe and terminates in 42, the typing rule for **if**-expressions, TYPE-IF, does not evaluate the condition.

The practical consequences of this limitation are that the type system has no typing rules for (1) recursive functions $\mathsf{fix}\, f\, x.\, e$, since they potentially do not terminate; (2) direct, unchecked buffer

$$\begin{array}{ll} \mathcal{V}[\![\mathsf{int}]\!] \triangleq \{n \mid n \in \mathbb{Z}\} & \mathcal{V}[\![\tau \times \sigma]\!] \triangleq \{(v, w) \mid v \in \mathcal{V}[\![\tau]\!] \land w \in \mathcal{V}[\![\sigma]\!]\} \\ \mathcal{V}[\![\mathsf{bv}\, n]\!] \triangleq \{u \mid 0 \le u < 2^n\} & \mathcal{V}[\![\mathsf{buf}\, \tau]\!] \triangleq \{b \mid \exists \vec{v}.\, b \mapsto_{\mathsf{u}} \vec{v} \land \forall w \in \vec{v}.\, w \in \mathcal{V}[\![\tau]\!]\} \\ \mathcal{E}[\![\tau]\!] \triangleq \{e \mid \exists v.\, e \Downarrow v \land v \in \mathcal{V}[\![\tau]\!]\} & \mathcal{V}[\![\tau \to \sigma]\!] \triangleq \{v \mid \forall w.\, w \in \mathcal{V}[\![\tau]\!] \Rightarrow v\, w \in \mathcal{E}[\![\sigma]\!]\} \end{array}$$

Fig. 8. Select cases of the logical relation for heap-dependent expressions in the *hProp*-fragment.

accesses $b[i]$, since the index $i$ could be out of bounds making the access unsafe; (3) assertions assert $(e)$, since they are treated as unsafe if the condition $e$ fails. We will explain below how we relax this limitation by using a *semantic type system*, such that functions like `checksum` can be type checked even though they need to iterate over a buffer. (In future work, one could additionally consider supporting syntactically restricted forms of recursion such as, *e.g.*, structural recursion.)

**Semantic typing.** In a *semantic type system* [1, 40, 61]—as opposed to a syntactic type system, which is a fixed collection of typing rules—one defines the typing judgment $\Gamma \vDash e : \tau$ via a logical relation and then proves the typing rules as lemmas about $\Gamma \vDash e : \tau$. The result is that one still obtains easily automatable rules such as the ones in Fig. 7—as with a syntactic type system—but additionally the type system is *open*: it can be extended with additional functions by proving manually that they have the right type (discussed below).

In our case, we define the logical relation—the basis for the typing judgment—in the *hProp*-fragment. The logical relation is depicted in Fig. 8. For each type $\tau$, the *value relation* $\mathcal{V}[\![\tau]\!]$ determines which values are of this type. For integers $\mathcal{V}[\![\mathsf{int}]\!]$, bitvectors $\mathcal{V}[\![\mathsf{bv}\, n]\!]$, and pairs $\mathcal{V}[\![\tau \times \sigma]\!]$, this is straightforward. Where things get more interesting is (1) stateful types like $\mathcal{V}[\![\mathsf{buf}\, \tau]\!]$, because they can use the unstable points-to $\ell \mapsto_{\mathsf{u}} v$ of *hProp* to refer to the current contents of the memory and (2) function types $\mathcal{V}[\![\tau \to \sigma]\!]$, because they can use the implication of *hProp* to say that a value $v$ is semantically a function of type $\tau \to \sigma$ if, applied to an argument $w$ of type $\tau$, it results in a value of type $\sigma$. To express "it results in a value of type $\sigma$", we define the *expression relation* $\mathcal{E}[\![\tau]\!]$, which uses the evaluation $e \Downarrow v$ of *hProp* to evaluate $e$ (and constrain the result).

With both the value and expression relation in hand, we then define the typing judgment as

$$\Gamma \vDash e : \tau \triangleq \forall \gamma.\, \gamma \in \mathcal{G}[\![\Gamma]\!] \Rightarrow \gamma(e) \in \mathcal{E}[\![\tau]\!] \qquad \textit{where} \quad \mathcal{G}[\![\Gamma]\!] \triangleq \{\gamma \mid \forall x : \tau \in \Gamma.\, \gamma(x) \in \mathcal{V}[\![\tau]\!]\}\,.$$

That is, an expression $e$ has type $\tau$ in context $\Gamma$ if for any closing substitution $\gamma \in \mathcal{G}[\![\Gamma]\!]$, the expression after substituting the free variables $\gamma(e)$ is semantically of type $\tau$.

**Extensibility.** Let us now discuss how we can extend the type system with additional functions that, internally, rely on language features without corresponding typing rules. For example, recall that the type system has no rule for recursive functions. However, for specific recursive functions such as the combinator iter (from §2.1), we can still manually *prove* that they are well-typed by unfolding the definition of $\Gamma \vDash e : \tau$ and applying the rules for $e \Downarrow v$ in Fig. 2:

Lemma 5.2. $\vDash \mathsf{iter} : \mathsf{int} \times \mathsf{int} \times \tau \times (\mathsf{int} \to \tau \to \tau) \to \tau$

Proof. By unfolding the logical relation, induction on the distance between the start bound and the end bound, and manually applying the proof rules for $e \Downarrow v$ to evaluate iter. □

As another example, recall that the type system has no typing rule for direct, unchecked buffer accesses $b[i]$. One can, however, define safe abstractions around these accesses such as the function $\mathsf{nthopt}(\mathsf{b}, \mathsf{i}) \triangleq \textbf{if } 0 \le \mathsf{i} < \mathsf{length}(\mathsf{b}) \textbf{ then } \mathsf{Some}(\mathsf{b}[\mathsf{i}]) \textbf{ else } \mathsf{None}$, which dynamically checks whether the index is in bounds. Alternatively, if one wants to avoid the use of options, one can also define the combinator $\mathsf{foldbuf}(\mathsf{b}, \mathsf{s}, \mathsf{f}) \triangleq \mathsf{iter}(0, \mathsf{length}(\mathsf{b}), \mathsf{s}, \lambda \mathsf{i}, \mathsf{a}.\, \mathsf{f}\ (\mathsf{b}[\mathsf{i}])\ \mathsf{a})$ for iterating over the contents of a buffer. Analogously to Lemma 5.2, one can then show that both are well-typed as:

$$\begin{array}{rl} \text{Sorts} & S, T ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{bv}\, n \mid \cdots \quad \text{Predicates} \quad p, q ::= \leq_{\mathsf{int}} \mid <_{\mathsf{int}} \mid \cdots \quad \text{Terms} \quad t, s ::= x \mid f\,\vec{t} \\ \text{Functions} & f, g ::= \mathsf{true} \mid \mathsf{false} \mid n \mid +_{\mathsf{int}} \mid \cdot_{\mathsf{int}} \mid -_{\mathsf{int}} \mid \mathsf{neg} \mid \mathsf{xor}_{\mathsf{bv}\,n} \mid ==_{\mathsf{bv}\,n} \mid \mathsf{if}_S \mid \cdots \\ \text{Formulas} & \pi, \chi ::= \mathsf{True} \mid \mathsf{False} \mid t \doteq_S s \mid p\,\vec{t} \mid \pi_1 \mathbin{\dot\wedge} \pi_2 \mid \pi_1 \mathbin{\dot\vee} \pi_2 \mid \pi_1 \mathrel{\dot\Rightarrow} \pi_2 \mid \dot\exists x : S.\, \pi \mid \dot\forall x : S.\, \pi \end{array}$$

Fig. 9. The syntax of the first-order logic of Daenerys.

Lemma 5.3. $\vDash \mathsf{nthopt} : \mathsf{buf}\,\tau \times \mathsf{int} \to \mathsf{option}\,\tau$ *and* $\vDash \mathsf{foldbuf} : \mathsf{buf}\,\tau \times \sigma \times (\tau \to \sigma \to \sigma) \to \sigma$

**Type checking.** Once we have proven that a function is well-typed, it can be added to the type system (*e.g.,* see type-nthopt in Fig. 7) and used to typecheck functions that call it. For example, using iter for iteration and nthopt for accessing buffers, we can define checksum as

$$\mathsf{checksum}(\mathsf{b}) \triangleq \mathsf{iter}(0, \mathsf{length}(\mathsf{b}), 0, \lambda \mathsf{i}, \mathsf{a}.\ \mathsf{a}\ \mathtt{xor}\ \mathsf{default}(\mathsf{nthopt}(\mathsf{b}, \mathsf{i}), 0))$$

where $\mathsf{default} : \mathsf{option}\,\tau \times \tau \to \tau$ unwraps the optional and returns a default value if it is None. Alternatively, avoiding options, we can also define $\mathsf{checksum}'(\mathsf{b}) \triangleq \mathsf{foldbuf}(\mathsf{b}, 0, \lambda \mathsf{x}, \mathsf{y}.\ \mathsf{x}\ \mathtt{xor}\ \mathsf{y})$. The implementations can then easily be type-checked simply by applying typing rules.

Lemma 5.4. $\vDash \mathsf{checksum} : \mathsf{buf}\,(\mathsf{bv}\,64) \to \mathsf{bv}\,64$ *and* $\vDash \mathsf{checksum}' : \mathsf{buf}\,(\mathsf{bv}\,64) \to \mathsf{bv}\,64$

Proof. By automatically applying typing rules on the implementations. □

Having type checked checksum, the property checksum-eval (from §2.1) follows via Lemma 5.1.

## 5.2 First-Order Logic

Let us now connect $e \Downarrow v$ to first-order logic, which enables using an SMT solver to automate proofs. Our focus is on providing sound foundations for an SMT integration; developing an automated conversion between Rocq and SMT, let alone foundationally verifying an SMT solver, is beyond the scope of this paper. Instead, we justify why proving a formula $\pi$ in first-order logic—oblivious to heaps—means that a corresponding *hProp*-formula $\langle \pi \rangle_{\mathsf{F}}$ over HDEAs can be assumed in Daenerys (Theorem 5.5). The payoff of this result is that, if one trusts an SMT solver like Z3 [18] to be sound w.r.t. standard first-order logic, then one can use it to verify properties with HDEAs (see §6). While IDF-based verifiers [5, 23, 42, 65] rely on similar correspondences, to our knowledge, we are the first to establish one foundationally for HDEAs as rich as ours (*e.g.,* with functions).

We work with a standard multi-sorted first-order logic, depicted in Fig. 9, with (1) *sorts* such as integers int and bitvectors $\mathsf{bv}\,n$, (2) *functions* such as $+_{\mathsf{int}}$ for integer addition, (3) *predicates* such as $\leq_{\mathsf{int}}$ for integer less-or-equal, (4) *terms* consisting of variables x and function applications $f\,\vec{t}$, and (5) *first-order logic formulas* $\pi$ over them. The sorts, terms, and predicates in Fig. 9 are *interpreted*, meaning their semantics corresponds to the intuitive mathematical semantics (*e.g.,* $+_{\mathsf{int}}$ is integer addition and not subtraction). In addition, the logic can be freely extended with *uninterpreted* sorts, functions, and predicates (indicated by "$\cdots$" in Fig. 9), whose semantics is for us to choose.

For example, for the validate-example from §2.3, we use an *uninterpreted sort* buffer and two *uninterpreted functions* $\mathsf{validate} : \mathsf{buffer} \times \mathsf{bv}\,64 \to \mathsf{bool}$ and $\mathsf{checksum} : \mathsf{buffer} \to \mathsf{bv}\,64$ to express the key relationship between checksum and validate as the first-order logic formula $\pi_{\mathsf{chk}}$:

$$\pi_{\mathsf{chk}} \triangleq \begin{array}{l} \left(\dot\forall(\mathsf{b} : \mathsf{buffer}), (\mathsf{w} : \mathsf{bv}\,64).\ \mathsf{validate}(\mathsf{b}, \mathsf{w}) \doteq_{\mathsf{bool}} (\mathsf{checksum}(\mathsf{b})\ \mathsf{xor}_{\mathsf{bv}\,64}\ \mathsf{w} ==_{\mathsf{bv}\,64} 0)\right) \mathrel{\dot\Rightarrow} \\ \dot\forall(\mathsf{b} : \mathsf{buffer}), (\mathsf{v} : \mathsf{bv}\,64).\ \mathsf{checksum}(\mathsf{b}) \doteq_{\mathsf{bv}\,64} \mathsf{v} \mathrel{\dot\Rightarrow} \mathsf{validate}(\mathsf{b}, \mathsf{v}) \doteq_{\mathsf{bool}} \mathsf{true} \end{array}$$

This is the kind of formula that an SMT solver like Z3 can prove. But note that it does *not* refer to the heap: checksum and validate—to an SMT solver—are simply function symbols. We will now connect it to an *hProp*-assertion about the heap-accessing $\lambda_{\mathsf{dyn}}$-functions checksum and validate.

**The translation.** To relate first-order logic and *hProp*-assertions, we introduce a translation $\langle\_\rangle$ from the former to the latter. It translates sorts S to types $\langle \mathsf{S} \rangle_{\mathsf{S}}$ (*e.g.,* mapping buffer to the type buf (bv 64)), functions f to $\lambda_{\mathsf{dyn}}$-values $\langle \mathsf{f} \rangle_{\mathsf{C}}$ (*e.g.,* mapping checksum to `checksum`), predicates p to *hProp*-predicates $\langle \mathsf{p} \rangle_{\mathsf{P}}$, terms $t$ to $\lambda_{\mathsf{dyn}}$-expressions $\langle t \rangle^{\gamma}_{\mathsf{T}}$, and formulas $\pi$ to *hProp*-assertions $\langle \pi \rangle^{\gamma}_{\mathsf{F}}$. The translation is given in the appendix [57, §D]. We discuss the most interesting cases:

$$\langle t \doteq_{\mathsf{S}} s \rangle^{\gamma}_{\mathsf{F}} \triangleq \langle t \rangle^{\gamma}_{\mathsf{T}} \equiv \langle s \rangle^{\gamma}_{\mathsf{T}} \qquad \langle \dot{\forall} x : \mathsf{S}.\, \pi \rangle^{\gamma}_{\mathsf{F}} \triangleq \forall v.\, v \in \mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!] \Rightarrow \langle \pi \rangle^{\gamma, x \mapsto v}_{\mathsf{F}} \qquad \langle \pi_1 \dot{\Rightarrow} \pi_2 \rangle^{\gamma}_{\mathsf{F}} \triangleq \langle \pi_1 \rangle^{\gamma}_{\mathsf{F}} \Rightarrow \langle \pi_2 \rangle^{\gamma}_{\mathsf{F}}$$

For term equality, we use evaluation $e \Downarrow v$ via $e_1 \equiv e_2 \triangleq \exists v.\, e_1 \Downarrow v \wedge e_2 \Downarrow v$. For quantification, we use the logical relation $\mathcal{V}[\![\tau]\!]$ to constrain the values. For implication, we map the implication of first-order logic $\pi_1 \dot{\Rightarrow} \pi_2$ directly to our *hProp*-implication $F \Rightarrow G$.

For example, translating $\pi_{\mathsf{chk}}$, we obtain $F_{\mathsf{chk}} \triangleq \langle \pi_{\mathsf{chk}} \rangle^{\emptyset}_{\mathsf{F}}$, which is given by:

$$(\forall b, w.\, b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!] \Rightarrow w \in \mathcal{V}[\![\mathsf{bv}\,64]\!] \Rightarrow \mathtt{validate}(b, w) \equiv (\mathtt{checksum}(b)\ \mathtt{xor}\ w == 0)) \Rightarrow$$
$$\forall b, v.\, b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!] \Rightarrow v \in \mathcal{V}[\![\mathsf{bv}\,64]\!] \Rightarrow \mathtt{checksum}(b) \equiv v \Rightarrow \mathtt{validate}(b, v) \equiv \mathsf{true}$$

Superficially, this assertion looks similar to $\pi_{\mathsf{chk}}$ (which is the point of $\langle \pi \rangle^{\gamma}_{\mathsf{F}}$). However, there is one crucial difference: as an *hProp*-assertion, $F_{\mathsf{chk}}$ implicitly *refers to the current heap.* In typical SL fashion, the heap itself is hidden, but it can be constrained via resources such as $\ell \mapsto_{\mathsf{u}} v$. For example, $b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!]$ contains an unstable points-to for the buffer $b$ (see Fig. 8), and $e_1 \equiv e_2$ implicitly evaluates `checksum` and `validate` on the current heap. In contrast, the formula $\pi_{\mathsf{chk}}$ is a first-order logic formula and does not mention a heap—neither explicitly nor implicitly.

Connecting *hProp*-assertions and first-order logic assertions is useful, because—as we will show below—it gives us access to formulas proven by the SMT solver in, *e.g.,* the rule of consequence.

**The correspondence.** Informally, we wish to prove that if $\pi$ holds in first-order logic, then we can get $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$ in Iris. To make this formal, we must specify what it means for a formula $\pi$ to hold. To do so, we define a standard Tarski semantics $\vDash \pi$ for first-order logic (see the appendix [57, §C]), where we make sure that the *interpreted* parts of the logic (*e.g.,* bitvectors and integers) have their standard mathematical semantics. With it, we establish the following result:

Theorem 5.5. *If* $\vDash \pi$ *holds, then* $(\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}$ *holds in Iris.*

In other words, if $\pi$ is true in first-order logic, then we can assume $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$ in Iris when proving a weakest precondition. (The weakest precondition gives us access to the current heap for $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$.) From this result, we can derive rules such as the consequence rule below (and HOARE-ASSERT in §2.3):

$$\textsc{hoare-conseq-fol}\ \frac{\vDash \pi \qquad P * \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \vdash \boxplus Q \qquad \{Q\}\ e\ \{v.\, R(v)\}}{\{P\}\ e\ \{v.\, R(v)\}}$$

If $\pi$ (*e.g.,* $\pi_{\mathsf{chk}}$) holds in first-order logic, then we can assume $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$ (*e.g.,* $F_{\mathsf{chk}}$) and use it to prove any frameable fact $Q$ from it. In our case studies in §6, we use it implicitly to justify solving queries about, *e.g.,* functions manipulating bitvectors and buffers automatically with an SMT solver, which would otherwise involve tedious manual reasoning about these theories.

The proof of Theorem 5.5 is beyond the space limitations of this paper. A detailed discussion can be found in the appendix [57, §D] (including the generalization of Lemma 5.6). The high-level idea is that we construct a model of first-order logic, where we must provide nonempty types for sorts S, meta-level predicates for predicate symbols p, and meta-level functions $f$ for function symbols f. To do so for function symbols f (*e.g.,* checksum), we use a generalization of Lemma 5.6, which turns well-typed $\lambda_{\mathsf{dyn}}$-functions f (*e.g.,* `checksum`) into meta-level functions $f$ (for a fixed heap $h$).

Lemma 5.6. *Let* f *be a* $\lambda_{dyn}$ *function such that* $\vdash \mathsf{f} \in \mathcal{V}[\![\mathsf{int} \to \mathsf{int}]\!]$. *For any heap* $h$, *there exists a function* $f : Val \to Val$ *such that for all integers* $n \in \mathbb{Z}$, *we have* $(\mathsf{f}\ n, h) \rightsquigarrow^{*}_{\mathsf{det}} (f(n), h)$ *and* $f(n) \in \mathbb{Z}$.

| Group | Case Study | Iris [30] | ViperCore [17] | Viper [42] | Daenerys |
|---|---|---|---|---|---|
| #1 | Channel Library | ● | ○ | ○ | ● |
| | Checksum Exchange | ○ | ○ | ● | ● |
| #2 | Popcount 32-bit Integer | ◐ | ● | ● | ● |
| | Popcount Buffer *à la* Redis [49] | ◐ | ● | ● | ● |
| | Priority Bit Map *à la* RefinedC [52] | ● | ● | ● | ● |
| #3 | Iterative Linked-List | ○ | ○ | ● | ● |
| #4 | Polymorphic Hashmap | ○ | ○ | ● | ● |
| #5 | Iris Concurrent Logical Relation [61] | ● | ○ | ○ | ● |
| | Barrier [29], Reader-Writer Lock, Spinlock | ● | ○ | ○ | ● |
| **Foundational Model** | | ● | ● | ○ | ● |

Fig. 10. Evaluation of Daenerys. We compare with the other approaches Iris, ViperCore, and Viper-based verifiers and mark whether they support the case study. We write ● for *yes*, ○ for *no*, and ◐ for case studies that *could conceivably be done* but require significant manual effort in Iris.

## 6 Case Studies

We apply Daenerys to several case studies, depicted in Fig. 10. A more detailed discussion and the code of the case studies can be found in the appendix [57, §E]. We discuss them below in five groups. To give an impression how Daenerys compares to other approaches, Fig. 10 provides a comparison with Iris [30]; tools based on Viper [5, 23, 42, 65] as expressive representatives of SL and IDF; and ViperCore [17], a recent foundational formalization of IDF and a subset of Viper.

**#1 The best of both worlds.** Group #1 illustrates how the marriage of Iris and IDF goes beyond what either approach typically provides in isolation. It consists of two parts: (1) a concurrent channel library and (2) a variation of the `checksum`-example (§2.2) using the channels to send a buffer and its checksum from a worker-thread to a client.

For the channel library, Daenerys benefits from its Iris roots. Due to its elaborate, step-indexed model, Iris supports *impredicative invariants* [29, 58], which can contain arbitrary Iris assertions. They allow us to prove general and modular specifications for the channel operations, where one can pick an arbitrary Iris predicate $\Phi$ to be exchanged over a channel:

$$\{\mathsf{True}\}\ \mathsf{chan}()\ \{c.\ \mathsf{ischan}(c,\Phi)\} \quad \mathsf{persistent}(\mathsf{ischan}(c,\Phi))$$

$$\{\mathsf{ischan}(c,\Phi) * \boxplus\Phi(v)\}\ \mathsf{send}(c,v)\ \{\_.\ \mathsf{True}\} \quad \{\mathsf{ischan}(c,\Phi)\}\ \mathsf{recv}(c)\ \{v.\ \boxplus\Phi(v)\}$$

When we create a channel with chan, we get an abstract predicate $\mathsf{ischan}(c,\Phi)$ that can be shared freely between threads (*i.e.,* is persistent). We can send a value $v$ satisfying $\Phi$ with send and receive a value satisfying $\Phi$ on the other end with recv. When we send a value, the predicate $\Phi$ should not depend on any unstable resources, which is ensured by the frame modality $\boxplus$.

We apply the channels to exchange a buffer and its checksum between a worker and a client:

$$\begin{aligned}
&\mathsf{client}() \triangleq \mathbf{let}\ (\mathsf{i},\mathsf{o}) = (\mathsf{chan}(),\mathsf{chan}())\ \mathbf{in}\ \ \mathsf{fork}\ \{\mathsf{wrk}(\mathsf{i},\mathsf{o})\}\ ;\\
&\qquad \mathsf{send}(\mathsf{i},(\mathtt{produceA},\mathtt{checksumA}));\mathbf{let}\ (\mathsf{b_A},\mathsf{s_A}) = \mathsf{recv}(\mathsf{o})\ \mathbf{in}\ \mathsf{assert}(\mathsf{s_A} == \mathtt{checksumA}(\mathsf{b_A}));\\
&\qquad \mathsf{send}(\mathsf{i},(\mathtt{produceB},\mathtt{checksumB}));\mathbf{let}\ (\mathsf{b_B},\mathsf{s_B}) = \mathsf{recv}(\mathsf{o})\ \mathbf{in}\ \mathsf{assert}(\mathsf{s_B} == \mathtt{checksumB}(\mathsf{b_B}))\\
&\mathsf{wrk}(\mathsf{i},\mathsf{o}) \triangleq \mathbf{let}\ (\mathsf{p},\mathsf{c}) = \mathsf{recv}(\mathsf{i})\ \mathbf{in}\ \mathbf{let}\ \mathsf{b} = \mathsf{p}()\ \mathbf{in}\ \mathbf{let}\ \mathsf{s} = \mathsf{c}(\mathsf{b})\ \mathbf{in}\ \mathsf{send}(\mathsf{o},(\mathsf{b},\mathsf{s}));\mathsf{wrk}(\mathsf{i},\mathsf{o})
\end{aligned}$$

The client creates an input channel i and an output channel o, spawns the worker-thread, and then sends the worker two different workloads: First, it sends `produceA` to produce a buffer with checksum function `checksumA`, receives the result $b_A$ and $s_A$, and ensures that $s_A$ matches `checksumA` of $b_A$. Then, it repeats the same process with a *different workload and checksum implementation.* The worker (1) receives on the input channel i a workload p and a checksum function c, (2) produces the buffer b by executing p, (3) computes the checksum s of b, (4) sends both b and s back via the output channel o, and (5) repeats the entire process.

Similar to the example in §2.2, HDEAs allow us here to avoid proving functional correctness of (two) checksum implementations. Instead, we can simply send the assertion $c(b) \Downarrow s$ from the worker to the client. We get the best of both worlds. The verification of the channels is beyond the scope of Viper (due to the higher-order abstract predicate), and reasoning about checksum as an HDEA is beyond Iris. The details of the exchange are described in the appendix [57, §E].

**#2 Leveraging SMT solvers.** Group #2 contains case studies that illustrate the benefit of the connection to first-order logic (§5.2). Specifically, we consider case studies where SMT solvers provide automation that otherwise, in a regular Iris proof, would require tedious manual reasoning about, *e.g.,* bitvectors. A poster child example in this category is the function

$$\begin{aligned}\mathsf{pc32}(\mathsf{x}) \triangleq\ & \textbf{let } \mathsf{y} = \mathsf{x} - (\mathsf{x} \gg 1 \,\&\, \texttt{0x55555555}) \textbf{ in let } \mathsf{z} = (\mathsf{y} \,\&\, \texttt{0x33333333}) + ((\mathsf{y} \gg 2) \,\&\, \texttt{0x33333333}) \textbf{ in} \\ & (((\mathsf{z} + (\mathsf{z} \gg 4)) \,\&\, \texttt{0x0F0F0F0F}) * \texttt{0x01010101}) \gg 24\end{aligned}$$

It counts the number of *ones* in the 32-bit bitvector x via an intricate combination of shifting, masking, addition, and multiplication. However, the SMT solver Z3 can show in an instant that pc32 behaves the same as $\mathsf{ones}(\mathsf{x}) \triangleq (\mathsf{x} \gg 31) \,\&\, \texttt{0x1} + \cdots + (\mathsf{x} \gg 0) \,\&\, \texttt{0x1}$. Thus, we encode the desired relationship between the two functions into a first-order logic formula $\pi_{\mathrm{pc32}}$ and use our first-order connection (Theorem 5.5) to show:

$$\vDash \pi_{\mathrm{pc32}} \quad \textit{implies} \quad \{0 \le u < 2^{32}\}\ \mathsf{pc32}(u)\ \{v.\, v \equiv \mathsf{ones}(u)\}, \quad \text{and then ask Z3 to prove } \pi_{\mathrm{pc32}}.$$

This case study ("Popcount 32-bit Integer" in Fig. 10) is a toy version of "Popcount Buffer", which is inspired by a popcount implementation from Redis [49] and works on a buffer of integers, processing *seven 32-bit integers at a time* using a scaled up version of pc32.

Furthermore, we verify a version of a bit map implementation previously verified in RefinedC [52], an Iris-based verification tool for C. Since Rocq provides little to no automation for bitvectors, the RefinedC-version requires around 300 lines of manual reasoning about bitvector arithmetic. In contrast, in Daenerys, we prove *not a single lemma* about bitvector arithmetic manually thanks to its connection to SMT solvers. (This connection requires a significant amount of boilerplate code at the moment that would be straightforward to automatically generate.) Note that Viper-based tools can verify all case studies in this category, but in contrast to ours, their encoding of assertions into first-order logic is not foundational.

**#3 Incremental Verification.** In Group #3, we incrementally verify a linked-list with increasingly stronger specifications—one of the strengths of IDF. In each step, we use an additional function operating on the linked-list to expose additional information about the data structure. For example, for the set-function, we prove:

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l,i,x)\ \{\_.\ \mathsf{list}(l)\} \qquad \{\mathsf{list}(l)\}\ \mathsf{set}(l,i,x)\ \{\_.\ \mathsf{list}(l) * \mathsf{len}(l) \Downarrow \textbf{old}\,\{\mathsf{len}(l)\}\}$$

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l,i,x)\ \{\_.\ \mathsf{list}(l) * \mathsf{len}(l) \Downarrow \textbf{old}\,\{\mathsf{len}(l)\} * (0 \le_{\mathsf{hp}} i <_{\mathsf{hp}} \mathsf{len}(l) \Rightarrow \mathsf{nth}(l,i) \Downarrow \mathsf{Some}\, x) * \cdots\}$$

where $e_1 \sim_{\mathsf{hp}} e_2 \triangleq \exists n_1, n_2.\, e_1 \Downarrow n_1 \wedge e_2 \Downarrow n_2 \wedge n_1 \sim n_2$ for $\sim \in \{<, \le\}$. The first specification expresses memory safety. The second one establishes that the length of the list, len, does not change. It uses

an "old expression" **old** $\{e\}$, which refers to the value of $e$ in the precondition. (**old** $\{e\}$ is described in the appendix [57, §E].) The third specification additionally uses an nth-function to say that the $i$-th value becomes $x$ and (omitted here) that all others are unchanged. Each step only involves strengthening the postcondition (thereby leaving proofs of clients intact) and only requires adding to the original proofs. This form of incremental verification is also supported in Viper-based tools (although functions such as len are not formalized in ViperCore).

In separation logics like Iris, in contrast, this form of incremental verification is not supported, because when choosing a representation predicate such as list, one simultaneously decides on an abstraction of the data structure (*e.g.,* the length, the contents, the underlying heap fragment, *etc.*). If one later wants to track a richer abstraction of the data structure, one has to define a new predicate and adapt the verification. (While one could start with a rich abstraction from the beginning but not expose all information to clients yet, this means already reasoning about the rich abstraction, *e.g.,* the list contents, when proving simpler properties like memory safety.) With HDEAs, the predicate stays the same and each function on the data structure exposes an additional abstraction.

**#4 Polymorphic Hashmap.** Group #4 verifies a polymorphic hashmap implementation to illustrate how we can use HDEAs to *relate different program expressions* without abstracting them to mathematical functions. This hashmap takes a user-provided equality function eq and a hash function hash and relies on the following property of these functions:

$$\forall x, y.\ \mathsf{eq}(x, y) \equiv \mathsf{true} \Rightarrow \mathsf{hash}(x) \equiv \mathsf{hash}(y) \qquad \text{(EQ-HASH-REL)}$$

What is interesting about EQ-HASH-REL is that we can state the relationship directly on the code of eq and hash using an almost-pure assertion. In a traditional Iris proof, one would first model eq and hash as mathematical functions *eq* and *hash* in order to state EQ-HASH-REL as a pure property. Furthermore, we can use an SMT solver to prove EQ-HASH-REL for concrete instantiations.

**#5 Iris Examples.** To show that Daenerys retains the expressiveness of Iris, we ported existing Iris proofs to Daenerys (Group #5). We ported the rich logical relation of Timany et al. [61] and several fine-grained concurrency examples, including the challenging Barrier example of Jung et al. [29]. In all cases, the effort was at most a few hours, and the delta over the original proofs is negligible (mostly adding frame modalities and introducing them in the goal). Viper-based tools do not support the expressive Iris features needed for these examples.

## 7 Related Work

**Implicit dynamic frames.** There is a long line of work on building automated verifiers based on implicit dynamic frames [37, 54, 64] culminating in the work around Viper [5, 23, 42, 65]. This work aims to build practical verification tools with a focus on automation. In contrast, our work aims to provide a foundational account for the meta-theory underlying IDF based on Iris with a focus on expressivity (*e.g.,* supporting higher-order quantification; see also the comparison in §6).

We discuss the two most closely related pieces of work that *do* focus on the meta-theory of IDF. Parkinson and Summers [45] show how to encode SL in IDF. They define an umbrella logic with a total-heap semantics—similar to a standard model for IDF—and then show that it has the intended meaning for SL assertions (stopping short of an SL program logic). We consider the opposite direction: we integrate IDF into a very expressive SL, Iris. In doing so, we define an encoding of IDF based on SL resources, which could be understood as a partial-heap semantics for IDF.

Dardinier et al. [17] provide a foundational approach for showing the soundness of verifiers based on intermediate verification languages. They extend a variant of resource algebras with stable and unstable projections and instantiate it with a subset of Viper called ViperCore. However, their notion of resource algebras is less expressive than our Iris-based resource algebras (*e.g.,*

no step-indexing and no persistency) and their work does not address advanced features (*e.g.,* frame preserving updates, heap-dependent functions, predicates, or impredicative invariants). Thus, they do not cover most of the examples described in §6 (see Fig. 10). As future work, it would be interesting to explore whether Daenerys could serve as a basis to model a larger fragment of Viper.

**Separation logic with unstable resources.** There is a long line of work on separation logics with unstable resources focused on fine-grained concurrency verification [16, 19–21, 43, 47, 58, 59]. None of them use unstable assertions about heaps or program expressions (*i.e.,* no HDEAs). Instead, they use unstable resources to unstably assert the current *logical state* $s$ of a concurrent program. At a technical level, an interesting difference is that they take the transition system $s \to s'$ as the primitive and then derive their notion of "stable resources" w.r.t. it. In contrast, for us, stability is a primitive notion of the resource algebras (see §3) and then we derive our updates $a \rightsquigarrow_{\mathsf{st}} b$ from it.

Charguéraud and Pottier [15] develop a read-only modality $\mathsf{RO}(P)$ that temporarily gives read-only access to the memory described by $P$. Like our unstable points-to $\ell \mapsto_{\mathsf{u}} v$, their read-only modality is freely duplicable, *i.e.,* $\mathsf{RO}(P) \vdash \mathsf{RO}(P) * \mathsf{RO}(P)$. However, read-only access and read-write access are temporally disjoint: in their work, one can either own $\mathsf{RO}(\ell \mapsto v)$ or $\ell \mapsto v$, but not both at the same time. Thus, their approach cannot support HDEAs: For HDEAs, it is crucial that $\ell \mapsto_{\mathsf{u}} v$ holds at the same time as $\ell \mapsto v$ (*e.g.,* for $\ell \mapsto v_{\mathsf{vec}} \vdash \ell \mapsto v_{\mathsf{vec}} * e_{\mathsf{add}} \Downarrow 42$ in §2.1).

Building on the work of Charguéraud and Pottier, Gospel [14] is a separation logic-based specification language for OCaml that allows leaving the mathematical model of an abstract predicate implicit and thus enables a form of incremental verification. However, Gospel still requires one to fix a final mathematical abstraction up-front, providing a different kind of incrementality than HDEAs. In particular, as demonstrated in case study #3 in §6, HDEAs do *not* require fixing a mathematical abstraction up-front. They allow one to incrementally add abstractions by considering additional functions (*e.g.,* len, nth, *etc.*) that expose different information about the data structure.

**Iris.** Daenerys alters Iris at a fundamental level by introducing unstable resources into its resource model and altering the frame rule. However, as mentioned in §6, for many traditional Iris proofs with only stable resources, the presence of instability should not introduce a noteworthy proof overhead (*e.g.,* we have backported several existing Iris proofs by changing only a few lines). It would be interesting to explore further use cases of unstable resources in Iris beyond our HDEAs.

Vindum et al. [63] develop a nextgen update modality $\Rightarrow^t P$ in Iris that—like our update $\Rrightarrow_{\mathsf{st}} -$ does not preserve (all) frames. Their modality—unlike Iris's $\Rrightarrow P$ and our $\Rrightarrow_{\mathsf{st}} P$—is not centered around the concept of (stable) frame preservation. Instead, it applies a user-specified function $t$ to the current resource (without being concerned with frame preservation). Also unlike us, they do not modify the notion of resources in Iris and do not consider HDEAs.

One goal of Daenerys is to improve automation for Iris via HDEAs, laying the foundation for integrating SMT solvers. This is orthogonal to other lines of work on automation for Iris [32, 41, 52, 66], which focus on different directions for automation (*e.g.,* automating resource manipulation in concurrent programs). BFF [66] and Katamaran [32] provide specialized bitvector solvers. While these solvers provide end-to-end foundational proofs, they are not as powerful as the bitvector support of state-of-the-art SMT solvers. For example, they cannot handle the Popcount 32-bit integer case study in group #2 from §6.

**SMT solvers and proof assistants.** There are several approaches that aim to integrate the automation of SMT solvers into foundational proof assistants [7, 24]. These approaches address a problem that is orthogonal to the results of this paper: They focus on reflecting proof artifacts of an SMT solver into a proof assistant, while Daenerys shows how one can leverage (potentially reflected) results from an SMT solver to reason about heap-accessing programs. In future work, it would be interesting to combine these techniques and obtain a foundational end-to-end result.

## Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback and Alex Summers for insightful discussions. This work was funded in part by a Google PhD Fellowship for the first author.

## Data Availability Statement

The Coq development and appendix for this paper can be found in Spies et al. [57]. The current development version of Daenerys is linked from the project webpage at https://plv.mpi-sws.org/iris-daenerys/.

## References

[1] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

[2] Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods (LNCS, Vol. 7226)*. Springer, 2. https://doi.org/10.1007/978-3-642-28891-3_2

[3] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[4] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. ACM, 109–122. https://doi.org/10.1145/1190216.1190235

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (1) (LNCS, Vol. 13243)*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[7] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.* 51, 1 (2013), 109–128. https://doi.org/10.1007/S10817-013-9278-5

[8] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *IFM (LNCS, Vol. 10510)*. Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

[9] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*. ACM, 259–270. https://doi.org/10.1145/1040305.1040327

[10] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

[12] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. ACM, 243–258. https://doi.org/10.1145/3341301.3359632

[13] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *ICFP*. ACM, 418–430. https://doi.org/10.1145/2034773.2034828

[14] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. 2019. GOSPEL - Providing OCaml with a Formal Specification Language. In *FM (LNCS, Vol. 11800)*. Springer, 484–501. https://doi.org/10.1007/978-3-030-30942-8_29

[15] Arthur Charguéraud and François Pottier. 2017. Temporary Read-Only Permissions for Separation Logic. In *ESOP (LNCS, Vol. 10201)*. Springer, 260–286. https://doi.org/10.1007/978-3-662-54434-1_10

[16] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (LNCS, Vol. 8586)*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[17] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2024. Formal Foundations for Translational Separation Logic Verifiers (extended version). arXiv:2407.20002 [cs.PL] https://arxiv.org/abs/2407.20002

[18] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[19] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*. ACM, 287–300. https://doi.org/10.1145/2429069.2429104
[20] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (LNCS, Vol. 6183)*. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24
[21] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 16:1–16:134. https://doi.org/10.1145/3477082
[22] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. ACM, 185–198. https://doi.org/10.1145/1706299.1706323
[23] Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *CAV (1) (LNCS, Vol. 10981)*. Springer, 596–603. https://doi.org/10.1007/978-3-319-96145-3_33
[24] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *CAV (2) (LNCS, Vol. 10427)*. Springer, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7
[25] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. ACM, 442–451. https://doi.org/10.1145/3209108.3209174
[26] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS, Vol. 6617)*. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4
[27] Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language*. Ph. D. Dissertation. Saarland University.
[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154
[29] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269. https://doi.org/10.1145/2951913.2951943
[30] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151
[31] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980
[32] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.* 6, ICFP (2022), 194–224. https://doi.org/10.1145/3547628
[33] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772
[34] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26
[35] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM, 205–217. https://doi.org/10.1145/3009837.3009855
[36] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037
[37] K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *ESOP (LNCS, Vol. 5502)*. Springer, 378–393. https://doi.org/10.1007/978-3-642-00590-9_27
[38] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38
[39] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP (LNCS, Vol. 11423)*. Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1
[40] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4
[41] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *PLDI*. ACM, 809–824. https://doi.org/10.1145/3519939.3523432
[42] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[43] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP (LNCS, Vol. 8410)*. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[44] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[45] Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP (LNCS, Vol. 6602)*. Springer, 439–458. https://doi.org/10.1007/978-3-642-19718-5_23

[46] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. https://doi.org/10.1145/3571194

[47] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP (LNCS, Vol. 9032)*. Springer, 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

[48] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1096–1120. https://doi.org/10.1145/3591265

[49] Redis. 2024. Redis Pocount Implementation for Potentially Large Buffers. https://github.com/redis/redis/blob/3fac869f02657d94dc89fab23acb8ef188889c96/src/bitops.c#L40.

[50] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[51] Rocq. 2025. The Rocq Prover. https://rocq-prover.org.

[52] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[53] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A multi-language platform for symbolic execution. In *PLDI*. ACM, 927–942. https://doi.org/10.1145/3385412.3386014

[54] Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP (LNCS, Vol. 5653)*. Springer, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

[55] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*. ACM, 80–95. https://doi.org/10.1145/3453483.3454031

[56] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later Credits: Resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[57] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. 2025. Destabilizing Iris (Rocq development and appendix). https://doi.org/10.5281/zenodo.15041581 Project webpage: https://plv.mpi-sws.org/iris-daenerys/.

[58] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[59] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP (LNCS, Vol. 7792)*. Springer, 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

[60] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL (2024), 241–272. https://doi.org/10.1145/3632851

[61] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* (July 2024). https://doi.org/10.1145/3676954

[62] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM, 377–390. https://doi.org/10.1145/2500365.2500600

[63] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2025. The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 83–97. https://doi.org/10.1145/3703595.3705876

[64] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 228:1–228:28. https://doi.org/10.1145/3428296

[65] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *CAV (1) (LNCS, Vol. 12759)*. Springer, 367–379. https://doi.org/10.1007/978-3-030-81685-8_17

[66] Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. 2022. BFF: foundational and automated verification of bitfield-manipulating programs. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1613–1638. https://doi.org/10.1145/3563345

Received 2024-11-14; accepted 2025-03-06