JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

VeriFast's separation logic: a logic without laters for modular verification of fine-grained concurrent programs

Bart Jacobs
KU Leuven, Department of Computer Science, DistriNet Research Group, Leuven, Belgium

ABSTRACT VeriFast is one of the leading tools for semi-automated modular formal program verification. A central feature of VeriFast is its support for *higher-order ghost code*, which enables its support for expressively specifying fine-grained concurrent modules, without the need for the *later* modality. We present the first formalization and soundness proof for this aspect of VeriFast's logic, and we compare it both to Iris, a state-of-the-art logic for fine-grained concurrency which features the later modality, as well as to some recent proposals for Iris-like reasoning without the later modality.

KEYWORDS Separation Logic, Fine-Grained Concurrency, Modular Verification.

1. Introduction

VeriFast (Vogels et al. 2015) is one of the leading tools for semi-automated modular formal verification of single-threaded and multithreaded C, Java, and Rust programs. It symbolically executes each function/method of the program, using a separation logic (O'Hearn et al. 2001; Reynolds 2002) representation of memory. It requires programs to be *annotated* with function/method preconditions and postconditions and loop invariants, as well as *ghost declarations*, such as definitions of separation logic predicates that specify the layout of data structures, and *ghost commands* for folding and unfolding predicates as well as invoking *lemma functions*, functions consisting entirely of ghost code. For expressive specification of fine-grained concurrent modules, it supports *higher-order ghost code*, in the form of *lemma function pointers* and *lemma function pointer type assertions*. While the general ideas underlying this specification approach have been described earlier (Jacobs & Piessens 2011), as have some examples of their use for solving verification challenges (Jacobs et al. 2015; Jacobs 2016), in this paper we present the first formalization and soundness proof for this aspect of VeriFast's logic. We define the programming language and introduce the running example in §2, define the syntax of annotations in §3, formalize the program logic implemented by VeriFast's symbolic execution algorithm in §4, and prove its soundness in §5. We discuss related work in §6.

JOT reference format:
Bart Jacobs. *VeriFast's separation logic: a logic without laters for modular verification of fine-grained concurrent programs.* Journal of Object Technology. Vol. 25, No. 1, 2026. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2026.25.1.a4

$$
\begin{array}{l}
\textbf{let}\ \mathsf{x} = \textbf{cons}(0)\ \textbf{in} \\
(\ \textbf{FAA}(\mathsf{x}, 1)\ ||\ \textbf{FAA}(\mathsf{x}, 1)\ ); \\
\textbf{let}\ \mathsf{v} = *\mathsf{x}\ \textbf{in} \\
\textbf{assert}\ \mathsf{v} = 2
\end{array}
$$

Figure 1 An example program. $\textbf{cons}(0)$ allocates a memory cell, initializes it to 0, and returns its address. The **FAA** command performs a sequentially consistent atomic fetch-and-add operation. $c_1 \,||\, c_2$ is the parallel composition of commands c_1 and c_2. $*\mathsf{x}$ returns the value stored at address x.

2. Programming language

In order to focus on the complexities of the logic rather than those of the programming language, we present VeriFast's separation logic in the context of a trivial concurrent programming language whose syntax is given in Fig. 2 and whose small-step operational semantics is given in Fig. 3. An example program that allocates a memory cell, increments it twice in parallel, and then asserts that the cell's value equals two is shown in Fig. 1.

$$
\begin{array}{rl}
 & z \in \mathbb{Z}, x \in \mathcal{X} \\
e ::= & z \mid x \\
i ::= & \mathbf{cons}(e) \mid \mathbf{FAA}(e,e) \mid *e \mid \mathbf{assert}\ e = e \\
c ::= & e \mid i \mid \mathbf{let}\ x = c\ \mathbf{in}\ c \mid (c \mid\mid c)
\end{array}
$$

Figure 2 Syntax of the expressions e, instructions i, and commands c of the programming language. We assume a set $\mathcal{X}$ of program variable names. $c; c'$ is a shorthand for $\mathbf{let}\ \_ = c\ \mathbf{in}\ c'$, where $\_$ is a designated element of $\mathcal{X}$

$$\frac{\ell \notin \operatorname{dom} h}{(h, \mathbf{cons}(v)) \to (h[\ell := v], \ell)}$$

$$\frac{\ell \in \operatorname{dom} h}{(h, \mathbf{FAA}(\ell, z) \to (h[\ell := h(\ell) + z], h(\ell))}$$

$$\frac{\ell \in \operatorname{dom} h}{(h, *\ell) \to (h, h(\ell))} \qquad (h, \mathbf{assert}\ v = v) \to (h, 0)$$

$$(h, \mathbf{let}\ x = v\ \mathbf{in}\ c) \to (h, c[v/x])$$

$$\frac{(h, c) \to (h', c')}{(h, \mathbf{let}\ x = c\ \mathbf{in}\ c'') \to (h', \mathbf{let}\ x = c'\ \mathbf{in}\ c'')}$$

$$\frac{(h, c) \to (h', c')}{(h, (c \mid\mid c'')) \to (h', (c' \mid\mid c''))}$$

$$\frac{(h, c) \to (h', c')}{(h, (c'' \mid\mid c)) \to (h', (c'' \mid\mid c'))} \qquad (h, v \mid\mid v') \to (h, 0)$$

Figure 3 Small-step operational semantics of the programming language

We define the multiset of threads of a command c as follows:

$$\mathsf{thrds}(c) = \begin{cases} \mathsf{thrds}(c_1) & \text{if } c = \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \\ \mathsf{thrds}(c_1) \uplus \mathsf{thrds}(c_2) & \text{if } c = (c_1 \mid\mid c_2) \\ \{\!\{c\}\!\} & \text{otherwise} \end{cases}$$

We say a configuration (h, c) is *reducible* if it can make a step:

$$\frac{(h, c) \to (h', c')}{\mathsf{red}\,(h, c)}$$

We say a configuration is *finished* if its command is a value.

$$\mathsf{finished}\,(h, z)$$

We say a configuration is *okay* if each thread is either reducible or finished.

$$\frac{\forall c_\mathsf{t} \in \mathsf{thrds}(c).\ \mathsf{finished}\,(h, c_\mathsf{t}) \vee \mathsf{red}\,(h, c_\mathsf{t})}{\mathsf{ok}\,(h, c)}$$

We say a configuration is *safe* if each configuration reachable from it is okay.

$$\frac{\forall h', c'.\ (h, c) \to^* (h', c') \Rightarrow \mathsf{ok}\,(h', c')}{\mathsf{safe}\,(h, c)}$$

We say a program c is safe if $(\emptyset, c)$ is safe. The goal of the logic that we present here is to prove that a given program is safe. This implies that it does not access unallocated memory and that there are no assertion failures.[1]

3. Annotated programs

When verifying a program with VeriFast, the user must first insert *annotations*, specifically *ghost declarations* and *ghost commands*, to obtain an *annotated program*. The syntax of ghost declarations and ghost commands is shown in Fig. 4. An annotated version of the example program is shown in Fig. 5. An annotated program may refer to ghost constructs declared in the *VeriFast prelude*, shown in Fig. 6.

There are two kinds of ghost declarations: *lemma type declarations* and *predicate constructor declarations*. These give meaning to *lemma type names* $t \in \mathcal{T}$ and *predicate constructor names* $p \in \mathcal{P}$. Conceptually, a lemma type is a predicate over a *lemma value* $\lambda\overline{g}.\ C$, a parameterized ghost command. A predicate constructor is a named, parameterized assertion. Applying a predicate constructor to an argument list produces a *predicate value* $p(\overline{V})$.

Besides integers, lemma values, and predicate values, ghost values may be pairs of ghost values, unit values $()$, and finite sets of ghost values.

Resources may be shared among threads using *atomic spaces* (analogous to Iris *invariants* (Jung et al. 2015, 2018)). An atomic space is (non-uniquely) identified by a *name* (any ghost value) and an *invariant* (a predicate value) (but there may be multiple atomic spaces with the same name and invariant at any

[1] In fact, the logic also proves that there are no data races, but for simplicity we do not consider data races here.

$$
\begin{array}{rll}
 & t \in \mathcal{T} & \text{lemma type names} \\
 & p \in \mathcal{P} & \text{predicate constructor names} \\
 & g \in \mathcal{G} & \text{ghost variable names} \\
 & \pi \in \mathbb{R}^+ & \text{fractions} \\
\text{ghost values } V ::= & z \mid (V, V) \mid () \mid \{\overline{V}\} & \\
 \mid & p(\overline{V}) & \text{predicate values} \\
 \mid & \lambda \overline{g}.\ G & \text{lemma values} \\
\text{ghost expressions } E ::= & V \mid x \mid g \mid E + E & \\
 \mid & p(\overline{E}) & \text{predicate constructor applications} \\
 \mid & (E, E) \mid () & \text{pair expressions, empty tuple} \\
 \mid & \emptyset \mid \{E\} \mid E \cup E \mid E \setminus E & \text{set expressions} \\
\text{assertions } a ::= & [\pi]E \mapsto E & \text{points-to assertions} \\
 \mid & [\pi]E \mapsto_{\mathsf{g}} E & \text{ghost cell points-to assertions} \\
 \mid & E() & \text{predicate assertions} \\
 \mid & [\pi]\mathbf{atomic\_space}(E, E) & \text{atomic space assertions} \\
 \mid & E : t(\overline{E}) & \text{lemma type assertions} \\
 \mid & \exists g.\ a & \\
 \mid & \mathbf{atomic\_spaces}(E) & \text{atomic spaces assertions} \\
 \mid & \mathbf{heap}(E) & \text{heap chunk assertions} \\
 \mid & a * a & \text{separating conjunctions} \\
gdecl ::= & \mathbf{lem\_type}\ t(\overline{g}) = \mathbf{lem}(\overline{g})\ \mathbf{forall}\ \overline{g}\ \mathbf{req}\ a\ \mathbf{ens}\ a & \\
 \mid & \mathbf{pred\_ctor}\ p(\overline{g})() = a & \\
I ::= & E(\overline{E}) & \\
 \mid & \mathbf{gcons}(E) \mid *E \leftarrow_{\mathsf{g}} E & \\
 \mid & \mathbf{open\_atomic\_space}(E, E) \mid \mathbf{close\_atomic\_space}(E, E) & \\
 \mid & E \leftarrow_{\mathsf{h}} E & \text{heap chunk update} \\
G ::= & I \mid \mathbf{glet}_{\mathsf{i}}\ g = G\ \mathbf{in}\ G & \\
C ::= & G \mid \mathbf{produce\_lem\_ptr\_chunk}\ t(\overline{E})(\overline{g})\ \{\ G\ \} & \\
 \mid & \mathbf{create\_atomic\_space}(E, E) \mid \mathbf{destroy\_atomic\_space}(E, E) & \\
\hat{c} ::= & e \mid i \mid \mathbf{let}\ x = \hat{c}\ \mathbf{in}\ \hat{c} \mid \hat{c} \mid\mid \hat{c} \mid \mathbf{glet}\ g = C\ \mathbf{in}\ \hat{c} &
\end{array}
$$

Figure 4 Syntax of ghost declarations *gdecl*, ghost instructions I, inner ghost commands G, outer ghost commands C (collectively called *ghost commands*), and annotated commands $\hat{c}$. Heap chunk assertions and heap chunk update commands are *internal*; they are not accepted by VeriFast in source code and are introduced here only for the sake of the soundness proof.

```
pred_ctor Inv(x, g1, g2)() =
  ∃v1, v2. [1/2]g1 ↦_g v1 ∗ [1/2]g2 ↦_g v2 ∗ x ↦ v1 + v2
pred_ctor pre1(x, g1, g2)() =
quad[1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 0
pred_ctor post1(x, g1, g2)() =
  [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 1
pred_ctor pre2(x, g1, g2)() =
  [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g2 ↦_g 0
pred_ctor post2(x, g1, g2)() =
  [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g2 ↦_g 1

let x = cons(0) in
glet g1 = gcons(0) in
glet g2 = gcons(0) in
create_atomic_space(Nx, Inv(x, g1, g2));
(
  produce_lem_ptr_chunk
    FAA_ghop(x, 1, pre1(x, g1, g2), post1(x, g1, g2))(op) {
    open_atomic_space(Nx, Inv(x, g1, g2));
    op();
    ∗g1 ←_g 1;
    close_atomic_space(Nx, Inv(x, g1, g2))
  };
  FAA(x, 1)
||
  produce_lem_ptr_chunk
    FAA_ghop(x, 1, pre2(x, g1, g2), post2(x, g1, g2))(op) {
    open_atomic_space(Nx, Inv(x, g1, g2));
    op();
    ∗g2 ←_g 1;
    close_atomic_space(Nx, Inv(x, g1, g2))
  };
  FAA(x, 1)
);
destroy_atomic_space(Nx, Inv(x, g1, g2));
let v = ∗x in
assert v = 2
```

Figure 5 VeriFast proof of the example program. $\mathsf{Nx} \triangleq ()$.

```
lem_type FAA_op(x, n, P, Q) = lem()
  forall v
  req x ↦ v ∗ P()
  ens x ↦ v + n ∗ Q()
lem_type FAA_ghop(x, n, pre, post) = lem(op)
  forall P, Q
  req atomic_spaces(∅) ∗ op : FAA_op(x, n, P, Q) ∗ P()
    ∗ pre()
  ens atomic_spaces(∅) ∗ op : FAA_op(x, n, P, Q) ∗ Q()
    ∗ post()
pred_ctor heap_(h)() = heap(h)
```

Figure 6 The ghost prelude (built-in ghost declarations). The declaration of heap_ is *internal*. It is not meant to be used in annotated programs; it is introduced here only for the sake of the soundness proof.

given time). At any point in time, ownership of the stock of logical resources in the system is distributed among the threads and the atomic spaces. That is, at any point, each logical resource is owned either by exactly one thread or by exactly one atomic space, or has been leaked irrecoverably. (More precisely, given that fractional resources are supported, the bundles of resources owned by the threads and the atomic spaces sum up to a logical heap that contains each physical points-to chunk only once and each **atomic_space** chunk only as many times as there are atomic spaces with that name and invariant, etc.) Creating an atomic space transfers a bundle of resources satisfying the atomic space's invariant from the creating thread to the newly created atomic space. Opening an atomic space transfers the resources owned by the atomic space to the opening thread; closing an atomic space again transfers a bundle of resources satisfying the atomic space's invariant from the closing thread to the atomic space. Destroying an atomic space transfers ownership of the resources owned by the atomic space to the destroying thread. To destroy an atomic space, the destroying thread must have full ownership of the atomic space. To open it, only partial ownership is required. (To close it, no ownership is required. If no such atomic space exists, the resources are leaked.) To prevent the same atomic space from being opened when it is already open, the set of opened atomic spaces is tracked using an **atomic_spaces**(S) chunk, where S is a set of the name-invariant pairs of the atomic spaces that are currently open.[2]

Lemma type assertions $V : t(\overline{V})$ assert that a given lemma value V is of a given lemma type t, applied to a given lemma type argument list $\overline{V}$. Such assertions are *linear*. To call a lemma, a full lemma type chunk for that lemma must be available, and it becomes unavailable for the duration of the call. A lemma type chunk is produced by the **produce_lem_ptr_chunk** ghost command. Since that command is not allowed inside lemmas, the stock of lemma type

[2] This means it is not possible to open two atomic spaces with the same name-invariant pair at the same time, even if multiple such atomic spaces exist.

chunks in the system only decreases as the lemma call stack grows; absence of infinite lemma recursion follows trivially.[3]

Intermediate results produced by ghost commands can be stored in *ghost variables*, which are like program variables except that they are in a separate namespace and can therefore never hide a program variable.[4] To facilitate reasoning about concurrent programs, annotated programs can furthermore allocate *ghost cells*; these are like physical memory locations except that they are allocated in a separate *ghost heap* and mutated using separate *ghost cell mutation commands*.

Points-to chunks, ghost points-to chunks, and atomic spaces can be owned *fractionally*, which allows them to be shared temporarily or permanently among multiple threads. A *fractional chunk* has a *coefficient* which is a positive real number.

4. Verification of annotated programs

In this section we formalize the program logic implemented by VeriFast's symbolic execution algorithm. We abstract over the mechanics of symbolic execution, the essence of which is described in Featherweight VeriFast (Vogels et al. 2015). In particular, the tool generally requires **open** and **close** ghost commands to unfold and fold predicates. Instead, here we use *semantic assertions*; predicates are fully unfolded during the interpretation of syntactic assertions as semantic assertions.

Core to VeriFast's verification approach is the concept of a *chunk* α:

$$\alpha ::= V \mapsto V \mid V \mapsto_{\mathrm{g}} V \mid \mathbf{atomic\_space}(V, V) \mid V : t(\overline{V}) \\ \mid \mathbf{atomic\_spaces}(V) \mid \mathbf{heap}(V)$$

A *logical heap* H is a function from chunks to nonnegative real numbers:

$$H \in \mathit{LogicalHeaps} = \mathit{Chunks} \to \mathbb{R}^+$$

We say a logical heap is *weakly consistent*, denoted $\mathsf{wok}\ H$ if no points-to chunk or ghost points-to chunk is present with a coefficient greater than 1, and no two (fractions of) points-to chunks or two (fractions of) ghost points-to chunks are present with the same left-hand side (address) but a different right-hand side (stored value).

[3] This is a simplification with respect to the actual VeriFast tool, which does support production of lemma type chunks inside lemmas, using a variant of the **produce_lem_ptr_chunk** syntax that additionally takes a block of ghost code. The chunk is available only until the end of that block. Now, suppose there is an infinite lemma call stack. Since the program text contains only finitely many **produce_lem_ptr_chunk** commands, among the lemmas that appear infinitely often in that call stack, there is one that is syntactically maximal, i.e. that is not itself contained within another lemma that also appears infinitely often. It follows that from some point on, the call stack contains no lemmas bigger than this maximal one. Since a lemma type chunk for a given lemma can only be produced by a bigger lemma (since the latter's body must contain a **produce_lem_ptr_chunk** command producing the former's), the stock of lemma type chunks for this maximal lemma will, from that point on, only decrease, which leads to a contradiction. (Note: for measuring the size of a lemma, the size of contained lemma *values* is not taken into account. It follows that substitution of values for ghost variables never affects the size of a lemma.)

[4] In the actual VeriFast tool, they are in the same namespace, but VeriFast checks that real code never uses a ghost variable.

We define *satisfaction* of an assertion a by a logical heap H, denoted $H \vDash a$, inductively as follows:

$$\frac{H(\alpha) \geq \pi}{H \vDash [\pi]\alpha}$$

$$\frac{\mathbf{pred\_ctor}\ p(\overline{g})() = a \qquad |\overline{V}| = |\overline{g}| \qquad H \vDash a[\overline{V}/\overline{g}]}{H \vDash p(\overline{V})()}$$

$$\frac{H \vDash a[V/g]}{H \vDash \exists g.\ a} \qquad \frac{H \vDash a \qquad H' \vDash a'}{H + H' \vDash a * a'}$$

A *semantic assertion* is a set of logical heaps. We define the *interpretation* $[\![a]\!]$ of an assertion as a semantic assertion as $[\![a]\!] = \{H \mid H \vDash a\}$.

We define *correctness* of an annotated command or ghost command $\dot{c}$ with respect to a precondition P and a postcondition Q (both semantic assertions), denoted $\{P\}\ \dot{c}\ \{Q\}$, inductively in Fig. 7. We define implication of semantic assertions as follows:

$$P \Rightarrow Q \triangleq \forall H \in P.\ \mathsf{wok}\ H \Rightarrow H \in Q$$

Note: nesting **produce_lem_ptr_chunk** commands is not allowed.

A correctness proof outline for the example annotated program is shown in Fig. 8.

We say an annotated program $\hat{c}$ is *correct* if $\{\mathsf{True}\}\ \hat{c}\ \{\mathsf{True}\}$.

We define the erasure of an annotated command $\hat{c}$ to a command $c = \mathsf{erasure}(\hat{c})$ as follows:

$$\begin{aligned} \mathsf{erasure}(c) &= c \\ \mathsf{erasure}(\mathbf{let}\ x = \hat{c}\ \mathbf{in}\ \hat{c}') &= \mathbf{let}\ x = \mathsf{erasure}(\hat{c})\ \mathbf{in}\ \mathsf{erasure}(\hat{c}') \\ \mathsf{erasure}(\hat{c} \mid\mid \hat{c}') &= \mathsf{erasure}(\hat{c}) \mid\mid \mathsf{erasure}(\hat{c}') \\ \mathsf{erasure}(\mathbf{glet}\ g = C\ \mathbf{in}\ \hat{c}) &= \mathsf{erasure}(\hat{c}) \end{aligned}$$

Theorem 1. *If an annotated program $\hat{c}$ is correct, then its erasure $\mathsf{erasure}(\hat{c})$ is safe.*

5. Soundness

We say a logical heap is *strongly consistent*, denoted $\mathsf{sok}\ H$, if, for every $V : t(\overline{V})$ such that $H(V : t(\overline{V})) > 0$, we have that V semantically is of type $t(\overline{V})$, denoted $\vDash V : t(\overline{V})$, defined as follows:

$$\frac{\begin{array}{c}\mathbf{lem\_type}\ t(\overline{g}')(\overline{g}'')\ \mathbf{req}\ a\ \mathbf{ens}\ a'' \\ |\overline{V}| = |\overline{g}'| \qquad |\overline{g}| = |\overline{g}''| \\ \forall \overline{V}'.\quad |\overline{V}'| = |\overline{g}| \ \Rightarrow \\ \{[\![a[\overline{V}/\overline{g}', \overline{V}'/\overline{g}'']]\!]\}\ G[\overline{V}'/\overline{g}]\ \{[\![a'[\overline{V}/\overline{g}', \overline{V}'/\overline{g}'']]\!]\}\end{array}}{\vDash \lambda \overline{g}.\ G : t(\overline{V})}$$

A *ghost heap* $\hat{h}$ is a partial function from integers to ghost values.

An *atomic spaces bag* A is a multiset of pairs $((V, V), H)$ of name-invariant pairs and logical heaps, such that for each element $((\_, V), H)$ we have $H \vDash V()$. We define the atomic

$$\{\mathsf{True}\}\ \mathbf{cons}(V)\ \{\mathsf{res} \mapsto V\} \qquad \{[\pi]\ell \mapsto V\} *\ell\ \{[\pi]\ell \mapsto V \wedge \mathsf{res} = V\} \qquad \{\ell \mapsto V\}\ \ell \leftarrow V'\ \{\ell \mapsto V'\}$$

$$\frac{\{P\}\ \hat{c}\ \{R\} \qquad \forall v.\ \{R[v/\mathsf{res}]\}\ \hat{c}'[v/x]\ \{Q\}}{\{P\}\ \mathbf{let}\ x = \hat{c}\ \mathbf{in}\ \hat{c}'\ \{Q\}} \qquad \begin{array}{l}\{V : \mathsf{FAA\_ghop}(\ell, z, V', V'') * [\![V'()]\!]\}\\ \mathbf{FAA}(\ell, z)\\ \{V : \mathsf{FAA\_ghop}(\ell, z, V', V'') * [\![V''()]\!]\}\end{array} \qquad \frac{\{P\}\ \hat{c}\ \{Q\} \qquad \{P'\}\ \hat{c}'\ \{Q'\}}{\{P * P'\}\ \hat{c}\ ||\ \hat{c}'\ \{Q * Q'\}}$$

$$\{\mathsf{True}\}\ \mathbf{gcons}(V)\ \{\mathsf{res} \mapsto_{\mathsf{g}} V\} \qquad \{\ell \mapsto_{\mathsf{g}} V\}\ \ell \leftarrow_{\mathsf{g}} V'\ \{\ell \mapsto_{\mathsf{g}} V'\}$$

$$\{[\![V'()]\!]\}\ \mathbf{create\_atomic\_space}(V, V')\ \{\mathbf{atomic\_space}(V, V')\}$$

$$\frac{(V, V') \notin S}{\begin{array}{l}\{\mathbf{atomic\_spaces}(S) * [\pi]\mathbf{atomic\_space}(V, V')\}\\ \mathbf{open\_atomic\_space}(V, V')\\ \{\mathbf{atomic\_spaces}(S \cup \{(V, V')\}) * [\pi]\mathbf{atomic\_space}(V, V') * [\![V'()]\!]\}\end{array}} \qquad \begin{array}{l}\{\mathbf{atomic\_spaces}(S) * [\![V'()]\!]\}\\ \mathbf{close\_atomic\_space}(V, V')\\ \{\mathbf{atomic\_spaces}(S \setminus \{(V, V')\})\}\end{array}$$

$$\{\mathbf{atomic\_space}(V, V')\}\ \mathbf{destroy\_atomic\_space}(V, V')\ \{[\![V'()]\!]\}$$

$$\frac{\begin{array}{c}\mathbf{lem\_type}\ t(\overline{g}) = \mathbf{lem}(\overline{g}')\ \mathbf{req}\ a\ \mathbf{ens}\ a'\\ |\overline{V}| = |\overline{g}| \qquad |\overline{g}''| = |\overline{g}'| \qquad \forall \overline{V}'.\ |\overline{V}'| = |\overline{g}'| \Rightarrow \{[\![a[\overline{V}/\overline{g}, \overline{V}'/\overline{g}']]\!]\}\ G[\overline{V}'/\overline{g}'']\ \{[\![a'[\overline{V}/\overline{g}, \overline{V}'/\overline{g}']]\!]\}\end{array}}{\{\mathsf{True}\}\ \mathbf{produce\_lem\_ptr\_chunk}\ t(\overline{V})(\overline{g}'')\ \{\ G\ \}\ \{\mathsf{res} : t(\overline{V})\}}$$

$$\frac{\mathbf{lem\_type}\ t(\overline{g}) = \mathbf{lem}(\overline{g}')\ \mathbf{req}\ a\ \mathbf{ens}\ a' \qquad |\overline{V}'| = |\overline{g}'|}{\{V : t(\overline{V}) * [\![a[\overline{V}/\overline{g}, \overline{V}'/\overline{g}']]\!]\}\ V(\overline{V}')\ \{V : t(\overline{V}) * [\![a'[\overline{V}/\overline{g}, \overline{V}'/\overline{g}']]\!]\}}$$

$$\{\mathbf{heap}(h) * \ell \mapsto \_\}\ \ell \leftarrow_{\mathsf{h}} v\ \{\mathsf{heap}(h[\ell := v]) * \ell \mapsto v\} \qquad \frac{\{P\}\ \dot{c}\ \{Q\}}{\{P * R\}\ \dot{c}\ \{Q * R\}} \qquad \frac{\forall V.\ \{P[V/g]\}\ \dot{c}\ \{Q\}}{\{\exists g.\ P\}\ \dot{c}\ \{Q\}}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ \dot{c}\ \{Q\} \qquad Q \Rightarrow Q'}{\{P\}\ \dot{c}\ \{Q'\}}$$

Figure 7 Correctness of annotated commands and ghost commands. We use $\dot{c}$ to range over both annotated commands and ghost commands.

```
{emp}
let x = cons(0) in glet g1 = gcons(0) in glet g2 = gcons(0) in
{x ↦ 0 ∗ g1 ↦_g 0 ∗ g2 ↦_g 0}
close Inv(x, g1, g2)();
{Inv(x, g1, g2)() ∗ [1/2]g1 ↦_g 0 ∗ [1/2]g2 ↦_g 0}
create_atomic_space(Nx, Inv(x, g1, g2));
{atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 0 ∗ [1/2]g2 ↦_g 0}
(
  {[1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 0}
  glet lem = produce_lem_ptr_chunk FAA_ghop(x, 1, pre1(x, g1, g2), post1(x, g1, g2))(op) {
    For all P, Q,
    {atomic_spaces(∅) ∗ op : FAA_op(x, 1, P, Q) ∗ P() ∗ pre1(x, g1, g2)()}
    open pre1(x, g1, g2)();
    { atomic_spaces(∅) ∗ op : FAA_op(x, 1, P, Q) ∗ P() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 0 }
    open_atomic_space(Nx, Inv(x, g1, g2)); open Inv(x, g1, g2)();
    { ∃v2. atomic_spaces({(Nx, Inv(x, g1, g2))}) ∗ op : FAA_op(x, 1, P, Q) ∗ P() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ g1 ↦_g 0 ∗ [1/2]g2 ↦_g v2 ∗ x ↦ v2 }
    For all v2,
    { atomic_spaces({(Nx, Inv(x, g1, g2))}) ∗ op : FAA_op(x, 1, P, Q) ∗ P() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ g1 ↦_g 0 ∗ [1/2]g2 ↦_g v2 ∗ x ↦ v2 }
    op();
    { atomic_spaces({(Nx, Inv(x, g1, g2))}) ∗ op : FAA_op(x, 1, P, Q) ∗ Q() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ g1 ↦_g 0 ∗ [1/2]g2 ↦_g v2 ∗ x ↦ 1 + v2 }
    ∗g1 ←_g 1;
    { atomic_spaces({(Nx, Inv(x, g1, g2))}) ∗ op : FAA_op(x, 1, P, Q) ∗ Q() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ g1 ↦_g 1 ∗ [1/2]g2 ↦_g v2 ∗ x ↦ 1 + v2 }
    close Inv(x, g1, g2)(); close_atomic_space(Nx, Inv(x, g1, g2));
    { atomic_spaces(∅) ∗ op : FAA_op(x, 1, P, Q) ∗ Q() ∗
      [1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 1 }
    close post1(x, g1, g2)()
    {atomic_spaces(∅) ∗ op : FAA_op(x, 1, P, Q) ∗ Q() ∗ post1(x, g1, g2)()}
  } in
  {[1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 0 ∗ lem : FAA_ghop(x, 1, pre1(x, g1, g2), post1(x, g1, g2))}
  close pre1(x, g1, g2)();
  {pre1(x, g1, g2)() ∗ lem : FAA_ghop(x, 1, pre1(x, g1, g2), post1(x, g1, g2))}
  FAA(x, 1);
  {post1(x, g1, g2)() ∗ lem : FAA_ghop(x, 1, pre1(x, g1, g2), post1(x, g1, g2))}
  open post1(x, g1, g2)()
  {[1/2]atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 1}
||
  ...
);
{atomic_space(Nx, Inv(x, g1, g2)) ∗ [1/2]g1 ↦_g 1 ∗ [1/2]g2 ↦_g 1}
destroy_atomic_space(Nx, Inv(x, g1, g2)); open Inv(x, g1, g2)();
{g1 ↦_g 1 ∗ g2 ↦_g 1 ∗ x ↦ 2}
let v = ∗x in
assert v = 2
```

Figure 8 Proof outline for the example proof

space chunks $\mathsf{chunks}(A)$ and the atomic spaces total owned heap $\mathsf{heap}(A)$ as follows:

$$\begin{array}{rl} \mathsf{chunks}(A) = & \{\!|\mathbf{atomic\_space}(V,V') \mid ((V,V'),\_) \in A|\!\} \\ \mathsf{heap}(A) = & \uplus_{(\_,H)\in A} H \end{array}$$

A *stock of lemma type chunks* Σ is a multiset of $(V,t,\overline{V})$ tuples. We say such a stock is *consistent* if for each $(V,t,\overline{V})$ in Σ, V is semantically of type $t(\overline{V})$.

We say a heap h and logical heap H are *consistent*, denoted $h \sim H$, if there exists a ghost heap $\hat{h}$, an atomic spaces bag A, and a consistent stock of lemma type chunks Σ such that $h + \hat{h} + \mathsf{chunks}(A) + \Sigma \geq \mathsf{heap}(A) + H$, where a heap is interpreted as a set of $\mapsto$ chunks and a ghost heap is interpreted as a set of $\mapsto_{\mathsf{g}}$ chunks. Notice: if $h \sim H$, it follows that H is strongly consistent.

We define the *weakest precondition* for n steps of a command c with respect to postcondition Q, denoted $\mathsf{wp}_n(c,Q)$, as the semantic assertion that is true for a logical heap H if either c is a value and $H \in Q$ or $n = 0$ or for each heap h and *frame* H' such that $h \sim H + H'$, all threads of c are either finished or reducible and for each step that (h,c) can make to some configuration (h',c'), there exists a logical heap H'' such that $h' \sim H'' + H'$ and H'' satisfies the weakest precondition of c' with respect to Q for $n-1$ steps:

$$\begin{array}{l} H \in \mathsf{wp}_n(c,Q) \Leftrightarrow \\ \quad \mathsf{finished}(\emptyset,c) \wedge H \in Q \vee n = 0 \vee \\ \quad \forall h, H'.\ h \sim H + H' \Rightarrow (h,c)\,\mathsf{ok} \wedge \\ \qquad \forall h', c'.\ (h,c) \rightarrow (h',c') \ \Rightarrow \\ \qquad\quad \exists H''.\ h' \sim H'' + H' \wedge H'' \in \mathsf{wp}_{n-1}(c',Q) \end{array}$$

We define the atomic space chunks $\mathsf{chunks}(S)$ for a set S of opened atomic spaces as follows:

$$\mathsf{chunks}(S) = \{\!|\mathbf{atomic\_space}(V,V') \mid (V,V') \in S|\!\}$$

We say a logical heap H is *self-consistent with depth bound* k, denoted $H\,\mathsf{ok}_k$, if there exists a heap h, a ghost heap $\hat{h}$, an atomic spaces bag A, a set of opened atomic spaces S, and a consistent stock of lemma type chunks Σ of size at most k such that $\{\!|\mathbf{heap}(h)|\!\} + h + \hat{h} + \mathsf{chunks}(A) + \mathsf{chunks}(S) + \{\!|\mathbf{atomic\_spaces}(S)|\!\} + \Sigma \geq \mathsf{heap}(A) + H$, where a heap is interpreted as a set of $\mapsto$ chunks and a ghost heap is interpreted as a set of $\mapsto_{\mathsf{g}}$ chunks. Notice: if $H\,\mathsf{ok}_k$, it follows that H is strongly consistent.

Notice that $h \sim H$ if and only if $\exists k, (H + \{\!|\mathbf{heap}(h), \mathbf{atomic\_spaces}(\emptyset)|\!\})\,\mathsf{ok}_k$.

Lemma 1 (Soundness of inner ghost command correctness)**.**

$$\begin{array}{l} \{P\}\ G\ \{Q\} \wedge H \in P \wedge (H + H')\,\mathsf{ok}_k \ \Rightarrow \\ \quad \exists H'' \in Q.\ (H'' + H')\,\mathsf{ok}_k \end{array}$$

Proof. By induction on k and nested induction on the size of G. The outer induction hypothesis is used to deal with lemma calls. □

Lemma 2. *If an annotated command $\hat{c}$ is correct with respect to precondition P and postcondition Q, then, for all n, P implies the weakest precondition of the erasure of $\hat{c}$ with respect to Q for n steps:*

$$\{P\}\ \hat{c}\ \{Q\} \Rightarrow \forall n.\ P \Rightarrow \mathsf{wp}_n(\mathsf{erasure}(\hat{c}), Q)$$

Proof. By induction on the derivation of the correctness judgment. The most interesting case is $\hat{c} = \mathbf{FAA}(\ell,z)$. Fix an n and a logical heap $H \in P$. Fix a heap h, a ghost heap $\hat{h}$, an atomic spaces bag A, a consistent stock of lemma type chunks Σ, and a frame H_{F} such that $h + \hat{h} + \mathsf{chunks}(A) + \Sigma = \mathsf{heap}(A) + H + H_{\mathsf{F}}$. By $H \in P$ and H strongly consistent we can fix a g, a G and an H' such that $H = \{\!|\lambda g.\ G : \mathsf{FAA\_ghop}(\ell,z,V_{\mathsf{pre}},V_{\mathsf{post}})|\!\} + H'$ and $H' \vDash V_{\mathsf{pre}}()$. By strong consistency of H, we have $\forall op, V_{\mathsf{P}}, V_{\mathsf{Q}}.\ \{\mathbf{atomic\_spaces}(\emptyset) * op : \mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}}) * [\![V_{\mathsf{P}}()]\!] * [\![V_{\mathsf{pre}}()]\!]\}\ G[op/g]\ \{\mathbf{atomic\_spaces}(\emptyset) * op : \mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}}) * [\![V_{\mathsf{Q}}()]\!] * [\![V_{\mathsf{post}}()]\!]\}$. We take $op = \lambda.\ \ell \leftarrow_{\mathsf{h}} h(\ell) + z$, $V_{\mathsf{P}} = \mathsf{heap\_}(h)$, and $V_{\mathsf{Q}} = \mathsf{heap\_}(h[\ell := h(\ell) + z])$. We have that semantically, op is of type $\mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}})$, so $\Sigma' = \Sigma - \{\!|\lambda g.\ G : \mathsf{FAA\_ghop}(\ell,z,V_{\mathsf{pre}},V_{\mathsf{post}})|\!\} + \{\!|\mathsf{op} : \mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}})|\!\}$ is consistent. We apply Lemma 1 to G using $H' + \{\!|\mathbf{atomic\_spaces}(\emptyset), \mathsf{op} : \mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}}), \mathbf{heap}(h)|\!\}$ for H, H_{F} for H' and the size of Σ' for k to obtain that there exists an $H'' \in [\![V_{\mathsf{post}}()]\!]$ such that $(H'' + \{\!|\mathbf{atomic\_spaces}(\emptyset), \mathsf{op} : \mathsf{FAA\_op}(\ell,z,V_{\mathsf{P}},V_{\mathsf{Q}}), \mathbf{heap}(h[\ell := h(\ell) + z])|\!\} + H_{\mathsf{F}})\,\mathsf{ok}_k$ and therefore $\{\!|\lambda g.\ G : \mathsf{FAA\_ghop}(\ell,z,V_{\mathsf{pre}},V_{\mathsf{post}})|\!\} + H'' \in Q$ and $h[\ell := h(\ell) + z] \sim H'' + \{\!|\lambda g.\ G : \mathsf{FAA\_ghop}(\ell,z,V_{\mathsf{pre}},V_{\mathsf{post}})|\!\} + H_{\mathsf{F}}$. □

Lemma 3. *If $h \sim H$ and $H \in \mathsf{wp}_n(c,\mathsf{True})$, then any configuration reached by (h,c) in at most n steps is okay.*

Proof. By induction on n. □

Theorem 1. *If an annotated program $\hat{c}$ is correct, then its erasure* $\mathsf{erasure}(\hat{c})$ *is safe.*

Proof. We first apply Lemma 2. Then, since the empty heap is consistent with the empty logical heap, we can finish the proof by applying Lemma 3 with $H = \emptyset$. □

6. Related work and discussion

Iris In contrast to the state-of-the-art logic for fine-grained concurrency verification Iris (Jung et al. 2015, 2018), the presented logic does not require the *later* modality. This is because atomic space invariants are stored in the logical heap in a *syntactic* form, rather than as propositions over logical heaps. As a result, no recursive domain equations are involved.

A downside of our approach compared to Iris, however, is that our logic does not directly support separating implications (a.k.a. magic wands), viewshifts, or other logical connectives in which some operand assertions appear in *non-positive* positions, i.e. whose truth is not monotonic in the truth of some of the operand assertions. This is because we define the meaning of predicate values using a least fixpoint construction.

We recover the functionality of separating implications and viewshifts to some extent by means of lemma values, with the major limitation that lemma type assertions are *linear*, which makes them more awkward to work with than the Iris constructs, although in practice this has not hindered us significantly so far; in fact, while we do vaguely remember encountering cases where this was inconvenient (or worse), we have trouble recalling the specific circumstances.

Having said that, we use VeriFast as a tool for verifying particular programs, not for metatheory development. It is very likely that the limitations of our logic would become prohibitive if we attempted to replicate deep metatheory developments such as RustBelt's lifetime logic (Jung 2020) in VeriFast. We do, however, make use of the *results* of such developments in VeriFast, through axiomatisation. The soundness of such axiomatisations, however, is a nontrivial question. While our axiomatisation of the lifetime logic *appears* sound, it is future work to build a formal argument of that, perhaps by connecting a Rocq mechanisation of the development of the present paper with that of the lifetime logic. Very important related work in this regard is Nola, which proposes a version of the lifetime logic, machine-checked in Rocq, without the later modality (see below).

Nola, Lilo VeriFast's logic is by no means the only later-less logic for fine-grained concurrency. Nola (Matsushita 2023; Matsushita & Tsukada 2025) is an Iris library that generalizes Iris's support for invariants (the Iris construct analogous to our atomic spaces) by parameterizing it over the type *Fml* of *formulas* that describe the contents of invariants, and the *semantics* $[\![\,]\!]$ *of formulas*, that maps a formula to an Iris proposition. Classical Iris invariants are obtained by taking $Fml = \blacktriangleright\mathsf{iProp}$, the type of Iris propositions but guarded by the later functor, and $[\![\mathsf{next}(P)]\!] = \triangleright P$. However, an alternative is to take as *Fml* a type of *syntactic* separation logic formulae. For most syntactic constructs, the semantics can be defined without the need for the later modality, thus enabling later-less invariant reasoning for a wide class of invariants. Matsushita & Tsukada (2025) were even able to implement a later-less version of RustBelt's lifetime logic this way. Furthermore, they show how to handle an even wider class of invariants without laters by applying *stratification*, where formulae at a layer $k+1$ can quantify over formulae at layer k. Lilo (Lee et al. 2025) applies Nola's idea of stratification to enable later-less invariant reasoning in a logic for verifying termination of busy-waiting programs under fair scheduling. Lilo was used to build the first modular total correctness proof of an elimination stack. Nola and Lilo are fully mechanized in Rocq.

Acknowledgments

We thank Justus Fasse for proofreading. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.

References

Jacobs, B. (2016). Partial solutions to VerifyThis 2016 challenges 2 and 3 with VeriFast. In V. Klebanov (Ed.), *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2016, Rome, Italy, July 17-22, 2016* (p. 7). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2955818

Jacobs, B., & Piessens, F. (2011). Expressive modular fine-grained concurrency specification. In T. Ball & M. Sagiv (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (pp. 271–282). ACM. Retrieved from https://doi.org/10.1145/1926385.1926417 doi: 10.1145/1926385.1926417

Jacobs, B., Smans, J., & Piessens, F. (2015). Solving the VerifyThis 2012 challenges with VeriFast. *Int. J. Softw. Tools Technol. Transf.*, *17*(6), 659–676. Retrieved from https://doi.org/10.1007/s10009-014-0310-9 doi: 10.1007/S10009-014-0310-9

Jung, R. (2020). *Understanding and evolving the Rust programming language* (Doctoral dissertation, Saarland University, Saarbrücken, Germany). Retrieved from https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., & Dreyer, D. (2018). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, *28*, e20. Retrieved from https://doi.org/10.1017/S0956796818000151 doi: 10.1017/S0956796818000151

Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., & Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani & D. Walker (Eds.), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (pp. 637–650). ACM. Retrieved from https://doi.org/10.1145/2676726.2676980 doi: 10.1145/2676726.2676980

Lee, D., Lee, J., Yoon, T., Cho, M., Kang, J., & Hur, C.-K. (2025, April). Lilo: A higher-order, relational concurrent separation logic for liveness. *Proc. ACM Program. Lang.*, *9*(OOPSLA1). Retrieved from https://doi.org/10.1145/3720525 doi: 10.1145/3720525

Matsushita, Y. (2023). *Non-step-indexed separation logic with invariants and Rust-style borrows* (Doctoral dissertation, University of Tokyo). doi: 10.15083/0002013242

Matsushita, Y., & Tsukada, T. (2025, June). Nola: Later-free ghost state for verifying termination in Iris. *Proc. ACM Program. Lang.*, *9*(PLDI). Retrieved from https://doi.org/10.1145/3729250 doi: 10.1145/3729250

O'Hearn, P. W., Reynolds, J. C., & Yang, H. (2001). Local reasoning about programs that alter data structures. In L. Fribourg (Ed.), *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings* (Vol. 2142, pp. 1–19). Springer. Retrieved from https://doi.org/10.1007/3-540-44802-0_1 doi: 10.1007/3-540-44802-0_1

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings* (pp. 55–74). IEEE Computer

Society. Retrieved from https://doi.org/10.1109/LICS.2002.1029817 doi: 10.1109/LICS.2002.1029817

Vogels, F., Jacobs, B., & Piessens, F. (2015). Featherweight VeriFast. *Log. Methods Comput. Sci.*, *11*(3). Retrieved from https://doi.org/10.2168/LMCS-11(3:19)2015 doi: 10.2168/LMCS-11(3:19)2015

About the author

Bart Jacobs is an associate professor at the DistriNet research group at the department of Computer Science at KU Leuven (Belgium). His main research interest is in modular formal verification of concurrent programs. You can contact the author at bart.jacobs@kuleuven.be or visit https://distrinet.cs.kuleuven.be/people/BartJacobs/.