# The Logical Essence of Well-Bracketed Control Flow

AMIN TIMANY, Aarhus University, Denmark
ARMAËL GUÉNEAU, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, France
LARS BIRKEDAL, Aarhus University, Denmark

A program is said to be well-bracketed if every called function must return before its caller can resume execution. This is often the case. Well-bracketedness has been captured semantically as a condition on strategies in fully abstract games models and multiple prior works have studied well-bracketedness by showing correctness/security properties of programs where such properties depend on the well-bracketed nature of control flow. The latter category of prior works have all used involved relational models with explicit state-transition systems capturing the relevant parts of the control flow of the program. In this paper we present the first Hoare-style *program logic* based on separation logic for reasoning about well-bracketedness and use it to show correctness of well-bracketed programs both directly and also through defining unary and binary logical relations models based on this program logic. All results presented in this paper are formalized on top of the Iris framework and mechanized in the Coq proof assistant.

CCS Concepts: • **Theory of computation** → **Program specifications**; **Program verification**; **Abstraction**; **Pre- and post-conditions**; **Logic and verification**; **Programming logic**; **Separation logic**; **Hoare logic**.

Additional Key Words and Phrases: well-bracketedness, stack discipline, program logics, program verification, logical relations, semantic typing

**ACM Reference Format:**
Amin Timany, Armaël Guéneau, and Lars Birkedal. 2024. The Logical Essence of Well-Bracketed Control Flow. *Proc. ACM Program. Lang.* 8, POPL, Article 20 (January 2024), 29 pages. https://doi.org/10.1145/3632862

## 1 INTRODUCTION

The control flow of a program is said to be well-bracketed if in every function call the caller can only resume execution after the callee has returned. In other words, function calls take place in a stack-like fashion which is the reason why they are usually implemented using a *call stack* at the hardware level. Note that in a programming language with continuations and/or concurrency, programs are not guaranteed to be well-bracketed.

Well-bracketedness has been captured semantically as a condition on strategies in fully abstract games models, e.g., Hyland and Ong [2000], and multiple prior works have studied well-bracketedness by showing correctness, e.g., Dreyer et al. [2010], Hur et al. [2012], Jaber and Murawski [2021], and security properties, e.g., Skorstengaard et al. [2019], Georges et al. [2021], of programs where such properties depend on the well-bracketed nature of control flow.

Prior work on reasoning about well-bracketed control flow [Dreyer et al. 2010] has epitomized the following *very awkward example* (VAE) as the key example of a program, written in an ML-like

Authors' addresses: Amin Timany, timany@cs.au.dk, Aarhus University, Aarhus, Denmark; Armaël Guéneau, armael.gueneau@inria.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Paris, France; Lars Birkedal, Birkedal@cs.au.dk, Aarhus University, Aarhus, Denmark.


ACM 2475-1421/2024/1-ART20
https://doi.org/10.1145/3632862

programming language, whose correctness depends on well-bracketedness of control flow:

$$\mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\, r \leftarrow 0; f\ (); r \leftarrow 1; f\ (); !\, r \qquad \text{(Very Awkward)}$$

The VAE program above first allocates a reference $r$ with value 0, and then returns a closure which takes a function argument $f$. The returned closure first resets the value of $r$ to 0 and subsequently calls $f$; here () is the unit value, *i.e.*, the only value in the unit type. After $f$ returns, the program continues by setting $r$ to 1, followed by another call to $f$. Finally, the program returns the value stored in $r$. VAE must always return 1 because the reference $r$ is a local reference, only accessible in the returned closure, which only reads $r$ after it has set it to 1. However, the argument for why this is the case is quite subtle, because the operation of reading $r$ and setting $r$ to 1 are separated by a call to $f$. One might hastily dismiss this call to $f$ since, after all, the reference $r$ is a local reference only accessible by the returned closure. Hence, one might fallaciously argue that the call to $f$ should in no way affect $r$. However, the function $f$ might itself include *calls to the closure*, and thus it *can* affect $r$! Nonetheless, as long as the execution is well-bracketed, VAE always returns 1. This is because, in a well-bracketed setting, any calls to the closure within $f$ must all end before $f$ returns, and hence the part of the code setting $r$ to 0 cannot take place after $f$ returns. This is in contrast to settings where control flow is not well-bracketed, *e.g.*, in the presence of continuations (see Dreyer et al. [2010] for an example where continuations break well-bracketedness of VAE) or concurrency (see Section 5 for an example of concurrency breaking well-bracketedness of VAE). Note: the existing literature sometimes refers to VAE also as the awkward example.

***A Historical Note.*** VAE was first mentioned by Ahmed et al. [2009] as a variant, attributed to Jacob Thamsborg, of an example by Pitts and Stark [1999]. The original example was called the "awkward" example by Pitts and Stark [1999] because they could not handle it. The Awkward example is as follows:

$$\mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\, r \leftarrow 1; f\ (); !\, r \qquad \text{(Awkward)}$$

The approach proposed by Ahmed et al. [2009] can reason about the Awkward example but not the VAE variant. The core difficulty of the Awkward example is that the value of the reference undergoes an *irreversible* change. The value is 0 prior to the assignment *only if* this is the first time the closure is being called. To quote Dreyer et al. [2010]: "One of the key contributions of the ADR [Ahmed et al. [2009]] model was to combine the machinery of step-indexed logical relations with that of Kripke logical relations in order to model higher-order state." and "The other key contribution of the ADR [Ahmed et al. [2009]] model was to provide an enhanced notion of possible world, which has the potential to express properties of local state that *evolve* over time."

Dreyer et al. [2010] state that they "recast the ADR [Ahmed et al. [2009]] model in the more familiar terms of state transition systems" which enables them to include so-called *private transitions* (see below) which are necessary in their approach to reason about well-bracketedness. Using these private transitions, Dreyer et al. [2010] presented the first formal proof establishing that VAE always returns 1. The use of private transitions is the key insight that subsequent works on reasoning about well-bracketedness prior to our work have directly built upon, and in a way, as we will allude to below, it is also what inspired us in developing our approach.

The logical relations model of Dreyer et al. [2010] was a so-called Kripke logical relations model, where the worlds consist of state transition systems, which describe how local state may evolve over time, and the key observation of Dreyer et al. [2010] is that a function can take private transitions as long as it is not observable by its callers. That is, it must restore the state to a publicly reachable

state before it returns. The state transition system for VAE is as follows:


(VAE-sts)

Here the solid arrow is a public transition and the dashed arrow is the private transition. Hence, the program can set the $r$ to 0 even if it is already 1 but it must restore it to 1 before returning, which is precisely the essence of the behavior of VAE.

***Using Separation Logic to Build Logical Relation Models.*** Modern separation logics such as Iris make it possible to build logical relations models at a higher-level of abstraction [Krebbers et al. 2017; Timany et al. 2022]. In particular, the combination of Iris invariants and ghost state provides a more abstract alternative to explicit worlds and state transition systems. The Awkward example can be verified in Iris using this approach: the proof does not involve explicit worlds or even building a logical relations model, as we show in Section 2. However, using Iris's invariants and ghost state has so far only been used to match the reasoning power of state-transition-system-based Kripke logical relations with *public* transitions, which do not allow reasoning about well-bracketedness.

A most relevant line of work is the study by Skorstengaard et al. [2019] and Georges et al. [2021] of security properties for programs at the assembly level. These works both consider well-bracketedness as a correctness aspect of the calling convention they present. To argue well-bracketedness, both works verify that (their assembly version of) VAE returns 1 under their respective calling convention. Georges et al. [2021] define their logical relations model using the Iris program logic, building on the earlier model of Skorstengaard et al. [2019], defined as a "traditional" step-indexed Kripke logical relation indexed over worlds consisting of state transition systems with private transitions, much in the style of Dreyer et al. [2010]. Georges et al. [2021] are able to reap *some* of the benefits of working in Iris, such as abstracting over less-relevant details of the model like step-indexing, and reasoning more abstractly about state. However, in order to reason about well-bracketedness, the model of Georges et al. [2021] is *still indexed over an explicit world* consisting of state transition systems. This makes their model more "invasive" in the sense that one has to consider public/private state transitions when reasoning about *any* program, not only when reasoning about examples relying on well-bracketedness.

***The Central Question.*** This raises a question. Can we reason about well-bracketed control flow without an explicit state transition system, or some such mechanism? In other words, can we develop a program logic that allows us to reason about well-bracketed control flow whenever necessary without complicating the reasoning about other programs whose correctness does not hinge on well-bracketedness of control flow? In this paper we answer this question in the affirmative. We present a program logic that satisfies the following desiderata:

**Conservative Extension** It is a Hoare-style program logic based on separation logic which validates *verbatim* all the usual reasoning principles of separation logic.[1]

**Bespoke Facilities** It at the same time offers *additional* reasoning principles for reasoning about well-bracketedness.

**Minimal Extension** It uses existing reasoning facilities, *e.g.*, Iris's ghost state and invariants, instead of encoding well-bracketedness as a state transition system with public/private transitions.

[1]For concreteness, we focus on the Iris separation logic in this paper, but we believe the ideas we present could also be used in other separation logics, *e.g.*, Nanevski et al. [2014].

**Versatility** It can be used to used to construct unary and binary logical relations models *in the standard way*, *i.e.*, as presented in Krebbers et al. [2017], to obtain a model which enables reasoning about well-bracketedness like in Georges et al. [2021] while hiding the details of the underlying mechanism.

To make the central question more concrete, we now sketch how the Awkward example can be proved in the Iris separation logic.

***The Essence of the Proof of the Awkward Example in Iris.*** To reason about non-trivial programs in separation logic one often uses ghost resources (aka auxiliary or logical state), and invariants, which are used to tie the ghost state to the physical program state. Ghost resources are strategically picked so as to ensure that they may only evolve in certain ways. For instance, a ghost resource could be defined so that it could only be increased and never decreased. In case of the Awkward example (see Section 2) ghost resources are used to define a ghost theory called the one-shot ghost theory. The one-shot ghost theory features a pair of predicates: $\mathsf{pending}(\gamma)$ and $\mathsf{shot}(\gamma)$ — $\gamma$ is the name of the resource instance.[2] The theory only allows the state to start in $\mathsf{pending}(\gamma)$ and evolve to $\mathsf{shot}(\gamma)$ but never the other way around. It is then required that the state of the reference is either 0 in which case we must have $\mathsf{pending}(\gamma)$ or it is 1 in which case we must have $\mathsf{shot}(\gamma)$. In Iris, this requirement can be enforced using an Iris invariant. This approach essentially reflects a state transition system in the ghost theory that is similar to VAE-sts but without the private transition. (This is precisely the state transition system Dreyer et al. [2010] used to prove correctness of the Awkward example.)

In summary, the combination of Iris's notion of invariants and ghost resources naturally supports reasoning about monotonic evolution of state. This makes Iris perfectly adapted to the kind of reasoning supported by *public* transitions in state-transition-systems-based Kripke logical relation models.

***The Well-Bracketed Program Logic.*** Now, given this perspective on Iris's methodology for proving correctness of examples like the Awkward example, we can recast our central question as follows:

> **If Iris's ghost state and invariants capture the public transitions, then which additional mechanism does a program logic need to capture the kind of reasoning supported by private transitions in Kripke logical relations models?**

The answer that we give to this question is (at least to us) surprisingly simple and elegant. We introduce a new program logic, called the *well-bracketed program logic*, which is a new Hoare logic of so-called well-bracketed Hoare triples. ***The additional key ingredient provided by our well-bracketed program logic is a novel logical mechanism of so-called ghost stacks.*** (Ghost stacks are themselves encoded using Iris's ghost resources; see Section 9 and the accompanying technical appendix for further details.) In combination with Iris's existing ghost state and invariants, ghost stacks make it possible to reason about well-bracketed evolution of state and resources, thus achieving the same reasoning power as enabled by private transitions.

Our well-bracketed program logic is *built on top of the existing Hoare logic of Iris*. Hence, just like the existing Hoare logic of Iris, the well-bracketed program logic can be instantiated with different programming languages. The main instantiation of the well-bracketed program logic presented in this paper is Iris's default concurrent higher-order imperative programming language known as HeapLang. The other two instantiations of the well-bracketed program logic presented in this paper are: (1) a variant of HeapLang that enables referring to execution traces based on Birkedal et al.'s

[2] Our one-shot resource is a slight simplification of the one-shot resource of Jung et al. [2016].

approach [Birkedal et al. 2021] (see Section 6), and (2) a variant of System F featuring higher-order references and recursive types for which we develop logical relations models (Section 7).

The HeapLang instantiation of the well-bracketed Hoare triples satisfy verbatim *all* the same reasoning principles as Iris's Hoare triples except for the rule for forking of threads (naturally so, since concurrency breaks well-bracketedness). Additionally, the well-bracketed program logic includes proof rules to allocate new *ghost stacks* (stacks of ghost names), and to access them when necessary. This is similar to how the standard Iris program logic allows the user to allocate new invariants, and to access them when necessary. The core idea of using ghost stacks to reason about well-bracketedness is then as follows. When using an invariant to tie the physical state of the program to a ghost resource, instead of using a fixed instance of a ghost resource, one ties the physical state to the ghost resource instance whose name is *on top of the ghost stack*. This shift, from a fixed instance of ghost state to a stack of ghost states enables encoding private transitions as follows. To make a private transition from state $s$ to state $s'$ we just leave the ghost state associated to state $s$ untouched. Instead, we allocate a fresh instance of ghost state in state $s'$. We then *pushes* the name of this newly allocated ghost state on top of the stack. In order to *roll back* a private transition, we simply *pop* the stack.

As we will demonstrate in Section 3.2, ghost stacks allow us to prove correctness of VAE using a stack of (names of) one-shot resources, *i.e.*, precisely the resource that we use in Section 2 for the proof of the Awkward example. We note that the point of the well-bracketed program logic is to increase the expressivity of the logic. It allows us to prove more expressive specifications for programs by using ghost stacks to reason about programs whose specifications relies on well-bracketedness. When proving specifications for programs that do not rely on well-bracketedness, we do not need ghost stacks. Hence, such proofs can be exactly as in the standard Iris program logic. This means that *one should think of the ghost stacks of the well-bracketed program logic as playing a role similar to the role invariants play for reasoning about concurrent programs — the stacks are a logical mechanism that allow us to easily prove specifications for a wider class of programs.*

***Contributions.***

- We present a well-bracketed Hoare logic on top of Iris's Hoare logic, satisfying the desiderata described above [Conservative Extension, Bespoke Facilities, Minimal Extension, and Versatility]. The key idea is a novel reasoning mechanism of so-called *ghost stacks*.
- We give a proof of VAE demonstrating that the well-bracketed Hoare logic can indeed be used to reason about well-bracketed control flow [Bespoke Facilities].
- We use the well-bracketed Hoare logic (instead of the ordinary Hoare logic of Iris) to construct unary and binary logical relations models for a variant of System F featuring higher-order references and recursive types [Versatility]; essentially a typed variant of the sequential subset of HeapLang. We use both logical relations models to reason about VAE. These logical relations models are the first such defined in Iris that support reasoning about well-bracketedness without explicit indexing over worlds.
- We use the technique of Birkedal et al. [2021] to demonstrate that the proof of VAE in our well-bracketed program logic implies that, in the trace generated by any program that uses VAE, the calls to the VAE closure are indeed well-bracketed.
- We prove that *any* state transition system with public and private transitions, in the sense of Dreyer et al. [2010], can be encoded using ghost stacks.
- We have mechanized all results in the Coq proof assistant on top of the Iris framework.

***The structure of the rest of the paper.*** Section 2 presents a proof of the Awkward example in Iris. We also use this example to introduce basic concepts of the Iris program logic. We present

our well-bracketed program logic in Section 3. In Section 3.2 we show how the well-bracketed program logic can be used to prove correctness, and well-bracketedness of VAE. Section 4 gives the definition of the well-bracketed program logic in terms of the underlying program logic of Iris. This definition is given in terms of what we will call a theory of stack collections. We discuss, in Section 4.2, how we derive the rules of the well-bracketed program logic. Then, in Section 5 we discuss how the well-bracketed program logic can be used, in conjunction with the underlying program logic of Iris, to reason about well-bracketedness of programs even in the presence of what we call innocuous concurrency. In Section 6 we show that well-bracketed Hoare triples enforce well-bracketedness of calls and returns, stated as an intensional trace property. In Section 7 we use our well-bracketed program logic to construct logical relations models capable of reasoning about well-bracketedness similar to prior works [Dreyer et al. 2010; Georges et al. 2021; Skorstengaard et al. 2019]. We demonstrate generality of ghost stacks in Section 8 by showing that we can use them to encode any state transition system with private and public transitions. Section 9 presents the construction of the theory of stack collections and in Section 10 we discuss related and future work. Finally, our concluding remarks are in Section 11.

## 2 THE PROOF OF THE AWKWARD EXAMPLE IN IRIS

In this section we briefly discuss the Iris program logic proof of the fact that the Awkward example always returns 1. The programming language we use to write examples in throughout this paper is the HeapLang language which Iris comes equipped with. HeapLang is an ML-like call-by-value language with general references and shared-memory concurrency — for this paper it suffices to have an intuitive understanding of its operational semantics. We recall the necessary parts of the Iris program logic along the way, and refer to the accompanying Coq development for the full details.

Formally, we wish to prove the following Hoare triple for Awkward:

$$\begin{array}{l}\{\mathsf{True}\}\\ \quad \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\ r \leftarrow 1; f\ (); !\, r\\ \{g.\ \forall f.\ \{\{\mathsf{True}\}\ f\ ()\ \{x.\ x = ()\}\}\ g\ f\ \{x.\ x = 1\}\}\end{array} \tag{awkward-spec}$$

Here, awkward-spec states that Awkward returns a closure $g$ that, when applied to any function $f$ satisfying a simple Hoare triple, returns 1. Note how in Hoare triples the return value is bound to a variable ($g$ and $x$ in the rule above) so that the postcondition can refer to it. We will also sometimes use a predicate as a postcondition and write $\{P\}\ e\ \{\Phi\}$ as a shorthand for $\{P\}\ e\ \{x.\ \Phi(x)\}$.

In Iris, we use a combination of invariants and ghost state to reason about the Awkward example. First, to capture the only-the-first-time evolution of the state in Awkward, we use the one-shot ghost state mentioned earlier. We introduce two Iris propositions $\mathsf{pending}(\gamma)$ and $\mathsf{shot}(\gamma)$, where $\mathsf{pending}(\gamma)$ may be turned into $\mathsf{shot}(\gamma)$ via a ghost state update, but not the other way around. The name $\gamma$ is used to identify a specific instance of these two predicates. Second, we use an invariant to hold these ghost resources and tie them to the physical state of the program (the reference allocated at the start).

To prove awkward-spec, we first allocate a fresh instance named $\gamma$ of the one-shot resource using rule MAKE-ONE-SHOT:

$$\textsc{make-one-shot}\qquad \vdash \Rrightarrow \exists \gamma.\ \mathsf{pending}(\gamma)$$

This rule states that we can always update ghost resources ($\Rrightarrow$ is Iris's so-called update modality) so as to make a new instance of the one-shot resource. Here, $\vdash$ stands for logical entailment between Iris propositions.

After the reference has been allocated by the program (which results in a memory location $\ell$), we use the freshly allocated $\mathsf{pending}(\gamma)$ resource to establish the following invariant:

$$\mathit{AwkwardInv} \triangleq \boxed{(\ell \mapsto 0 * \mathsf{pending}(\gamma)) \vee (\ell \mapsto 1 * \mathsf{shot}(\gamma))}^{\mathcal{N}_{\mathsf{awk}}} \qquad \text{(awkward-inv)}$$

In Iris we write $\boxed{P}^{\mathcal{N}}$ for the *persistent* (and hence duplicable) proposition that asserts that $P$ holds invariantly. The $\mathcal{N}$ in $\boxed{P}^{\mathcal{N}}$ is the name of the invariant. It is used for bookkeeping purposes. Looking inside our invariant, the proposition $\ell \mapsto v$ asserts *exclusive* ownership over memory location $\ell$, and, in addition states that this location stores the value $v$. The connective $*$ is called the *separating* conjunction. It incorporates a notion of separation or disjointness. Intuitively, $P * Q$ holds if we can split resources into two *disjoint* parts, one satisfying $P$ and the other $Q$.

The program logic of Iris (Iris's Hoare logic) enforces that an invariant, once established, is always preserved. This is effectively achieved by the rule INV-OPEN enforcing that invariants can only be accessed for the duration of atomic operations, *i.e.*, an operation like loading from, or storing into a reference.[3] Interestingly, it might appear that this rule is *consuming* the invariant while opening it, but this is in fact not a problem as invariants are always persistent and hence can be duplicated as necessary.

$$\textsc{inv-open}\ \frac{\left\{I * P\right\} e \left\{x.\ I * Q\right\} \qquad e \text{ is atomic}}{\left\{\boxed{I}^{\mathcal{N}} * P\right\} e \left\{x.\ Q\right\}}$$

Opening an invariant allows us to access the resource it contains, as long as the invariant can be re-established afterwards. Furthermore, Iris's Hoare logic is designed to allow us to update resources (formally, to eliminate the update modality) at any point throughout the proof.

The one-shot ghost theory, in addition to the rule MAKE-ONE-SHOT earlier, enjoys the following rules:

| SHOOT | PENDING-NOT-SHOT | SHOT-PERSISTENT |
|---|---|---|
| $\mathsf{pending}(\gamma) \vdash \Rrightarrow \mathsf{shot}(\gamma)$ | $\mathsf{pending}(\gamma) * \mathsf{shot}(\gamma) \vdash \mathsf{False}$ | $\vdash \mathsf{persistent}(\mathsf{shot}(\gamma))$ |

Using the SHOOT rule, any pending resource can be shot by updating ghost resources. Note that $\mathsf{shot}(\gamma)$ is *persistent* (and hence duplicable) as it merely asserts the knowledge that the resource is not pending anymore. On the other hand, $\mathsf{pending}(\gamma)$ asserts *exclusive* ownership indicating that the instance $\gamma$ of the one-shot resource is not shot yet. This is to ensure that each instance of the one-shot resource can only be shot once; hence the name. Note, in particular, that persistence of $\mathsf{shot}(\gamma)$ and exclusivity of $\mathsf{pending}(\gamma)$ mean that we have $\mathsf{shot}(\gamma) \dashv\vdash \mathsf{shot}(\gamma) * \mathsf{shot}(\gamma)$, for $\mathsf{shot}(\gamma)$, but not the analogous for $\mathsf{pending}(\gamma)$. Here, $\dashv\vdash$ is the bi-entailment relation (logical entailment in both directions). Finally, the rule PENDING-NOT-SHOT states that it is not possible for an instance of the one-shot resource to be both shot and pending at the same time. Notice how the behavior of the one-shot resource closely resembles the manner in which the state of the reference in the Awkward example evolves — the reference stores 0 (the writing of 1 is pending) until the store operation takes place (is shot), a change which persists. (The accompanying technical appendix details how this ghost theory is defined using Iris's resource algebras.)

[3] In practice Iris's Hoare triples are equipped with a mask, which is a set of invariant names, to ensure that invariants are not opened in a nested way. For the sake of simplicity we ignore these masks in this paper. We also elide the use of the so-called later modality in this paper which technically should also appear in this rule.

To finish the proof, we only need to show the following Hoare triple for the rest of the program:

$$\left\{\mathit{AwkwardInv} * \left\{\mathsf{True}\right\} f\ ()\ \left\{x.\ x = ()\right\}\right\}\ \ell \leftarrow 1; f\ (); !\,\ell\ \left\{x.\ x = 1\right\} \tag{1}$$

Since, the operation $\ell \leftarrow 1$ is an atomic operation we get to assume the invariant *AwkwardInv* before it, and need to reestablish it afterwards. The essence of the argument here is that we can prove the following Hoare triple:

$$\left\{(\ell \mapsto 0 * \mathsf{pending}(\gamma)) \vee (\ell \mapsto 1 * \mathsf{shot}(\gamma))\right\}\ \ell \leftarrow 1\ \left\{x.\ x = () * \ell \mapsto 1 * \mathsf{shot}(\gamma)\right\} \tag{2}$$

Note how the postcondition above allows us to reestablish the invariant, and at the same time, since $\mathsf{shot}(\gamma)$ is persistent (and thus duplicable), also retain a copy of $\mathsf{shot}(\gamma)$. To prove (2) we simply perform a case analysis on the disjunction in the precondition. In the case of the right disjunct nothing needs to be done. In the case of the left disjunct, after the operation we obtain $\ell \mapsto 1$ in the postcondition as the storing operation has mutated the heap of the program. In addition, from the precondition we have $\mathsf{pending}(\gamma)$. To reconcile the postcondition we only need to produce $\mathsf{shot}(\gamma)$ from $\mathsf{pending}(\gamma)$. This is precisely what the rule SHOOT allows us to do. To finalize the proof, it suffices to show

$$\left\{\mathit{AwkwardInv} * \left\{\mathsf{True}\right\} f\ ()\ \left\{x.\ x = ()\right\} * \mathsf{shot}(\gamma)\right\}\ f\ (); !\,\ell\ \left\{x.\ x = 1\right\} \tag{3}$$

For the call to $f$ we have a Hoare triple provided in the precondition. As for loading from $\ell$, since it is an atomic operation, we can access the invariant again. This time though, since we also have $\mathsf{shot}(\gamma)$ we can discard the left disjunct of the invariant as it is impossible due to the rule PENDING-NOT-SHOT. The fact that we can discard the left disjunct is a key step; it captures the irreversibility of the update to $\ell$.

## 3 WELL-BRACKETED PROGRAM LOGIC

In this section we present the well-bracketed program logic which is centered around the notion of well-bracketed Hoare triples, which we write as follows:

$$\left(\!\left|P\right|\!\right)\ e\ \left(\!\left|x.\ Q\right|\!\right)^{O}$$

Just like with an ordinary Hoare triple, $P$ is the precondition, $e$ is the program, and $Q$ is the postcondition which may also refer to the final return value of the program bound to $x$. Additionally, a well-bracketed Hoare triple is parameterized by a *stack mask* $O$, representing the set of names of stacks that are *out*, *i.e.*, those that are not controlled by the program logic (we will explain this in more detail later). We will omit the stack mask $O$ whenever it is just the empty set.

Intuitively, a well-bracketed Hoare triple is very much like an ordinary triple, except that it has additionally access to *ghost stacks* which allow reasoning about well-bracketed evolution of ghost state. In Iris, an ordinary triple asserts that assuming $P$, $e$ runs safely and produces a result satisfying $Q$ while all invariants are satisfied at each step. A well-bracketed Hoare triple asserts that assuming $P$, $e$ runs safely and produces a result satisfying $Q$ while all invariants are satisfied, *and that ghost state tracked in ghost stacks evolves in a well-bracketed manner* matching the execution of the program.

***Standard rules.*** The well-bracketed program logic satisfies all the rules satisfied by the ordinary Iris program logic for HeapLang, except the fork rule, since concurrency can break well-bracketedness. An excerpt of these is given in Figure 1. Note how these rules never effect the stack mask. The rules WBHOARE-ALLOC, WBHOARE-LOAD and WBHOARE-STORE simply reflect the operational semantics of the language, *e.g.*, WBHOARE-ALLOC expresses that when we allocate a new location, it is *fresh* because we immediately receive *exclusive* ownership over it in terms of the points-to proposition $x \mapsto v$. The rule WBHOARE-CONSEQUENCE and WBHOARE-INV-OPEN need little explanation.

$$\textsc{wbHoare-consequence}\quad \frac{Q \Longrightarrow P \qquad \forall v.\ \Phi(v) \Longrightarrow \Psi(v)}{\left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O} \vdash \left(\!\left|Q\right|\!\right)\ e\ \left(\!\left|\Psi\right|\!\right)^{O}}$$

$$\textsc{wbHoare-inv-open}\quad \frac{\left(\!\left|I * P\right|\!\right)\ e\ \left(\!\left|x.\ I * Q\right|\!\right)^{O} \qquad e \text{ is atomic}}{\left(\!\left|\boxed{I}^{N} * P\right|\!\right)\ e\ \left(\!\left|x.\ Q\right|\!\right)^{O}}$$

$$\textsc{wbHoare-bind}\quad \frac{\left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Psi\right|\!\right)^{O} \qquad \forall v.\ \left(\!\left|\Psi(v)\right|\!\right)\ K[v]\ \left(\!\left|\Phi\right|\!\right)^{O}}{\left(\!\left|P\right|\!\right)\ K[e]\ \left(\!\left|\Phi\right|\!\right)^{O}}$$

$$\textsc{wbHoare-frame}\quad \left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O} \vdash \left(\!\left|P * R\right|\!\right)\ e\ \left(\!\left|x.\ \Phi(x) * R\right|\!\right)^{O}$$

$$\textsc{wbHoare-alloc}\quad \vdash \left(\!\left|\mathsf{True}\right|\!\right)\ \mathsf{ref}(v)\ \left(\!\left|x.\ x \mapsto v\right|\!\right)^{O}$$

$$\textsc{wbHoare-load}\quad \vdash \left(\!\left|\ell \mapsto v\right|\!\right)\ !\ell\ \left(\!\left|x.\ x = v * \ell \mapsto v\right|\!\right)^{O}$$

$$\textsc{wbHoare-store}\quad \vdash \left(\!\left|\ell \mapsto v\right|\!\right)\ \ell \leftarrow w\ \left(\!\left|x.\ x = () * \ell \mapsto w\right|\!\right)^{O}$$

Fig. 1. All the rules satisfied by the ordinay program logic are also satisfied by the well-bracketed program logic — this figure depicts a sample set.

$$\textsc{wbHoare-create-stack}\quad \frac{\forall N.\ \mathit{stack}_{\bullet}(N, []) * \mathit{stack}_{\circ}(N, []) \vdash \Rrightarrow R(N) * \exists s.\ \mathit{stack}_{\bullet}(N, s) \qquad \forall N.\ \left(\!\left|P * R(N)\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O}}{\left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O}}$$

$$\textsc{wbHoare-access-stack}\quad \frac{N \notin O}{\left(\forall s.\ \left(\!\left|P * \mathit{stack}_{\bullet}(N, s)\right|\!\right)\ e\ \left(\!\left|x.\ \Phi(x) * \mathit{stack}_{\bullet}(N, s)\right|\!\right)^{O \cup \{N\}}\right) \vdash \left(\!\left|\mathit{stack}_{\exists}(N) * P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O}}$$

$$\textsc{wbHoare-mend-stack}\quad \left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O \setminus \{N\}} \vdash \left(\!\left|\mathit{stack}_{\bullet}(N, s) * P\right|\!\right)\ e\ \left(\!\left|x.\ \Phi(x) * \mathit{stack}_{\bullet}(N, s)\right|\!\right)^{O}$$

$$\textsc{Hoare-wbHoare}\quad \{P\}\ e\ \{\Phi\} \vdash \left(\!\left|P\right|\!\right)\ e\ \left(\!\left|\Phi\right|\!\right)^{O}$$

Fig. 2. Rules connecting well-bracketed Hoare triples and ghost stacks.

The former is the standard rule of consequence for Hoare logic and the latter is the invariant opening rule INV-OPEN expressed for well-bracketed Hoare triples. The rule WBHOARE-FRAME is the frame rule of separation logic. It states that once a program is proven correct with respect to a certain pre- and postcondition it remains correct in the presence of other resources, be it physical or ghost resources. In rule WBHOARE-FRAME, the use of separating conjunction is key as owning more resources *disjoint* from $P$ can never invalidate $P$. The rule WBHOARE-BIND is an important rule for modular reasoning. It concerns programs of the form $K[e]$ that consist of an expression $e$ under an evaluation context $K$, the part of the program that would run following evaluation of $e$. The rule WBHOARE-BIND states that to show correctness of $K[e]$, it suffices to show that $e$ is correct, and that $K$ is correct when plugging into it any value that satisfies the postcondition of $e$.

***Accessing ghost stacks.*** The distinguishing feature of well-bracketed triples is the ability to access ghost stacks. Concretely, ghost stacks are tracked using three novel predicates of our well-bracketed program logic: $\mathit{stack}_{\bullet}(N, s)$ called system stack, $\mathit{stack}_{\circ}(N, s)$ called user stack, and $\mathit{stack}_{\exists}(N)$ called stack-exists. In these predicates, $N$ is the name of the stack and $s$ is a sequence of ghost resource names which constitutes the contents of the stack. The idea is that $\mathit{stack}_{\bullet}(N, s)$ and

$\mathit{stack}_\circ(N, s)$ must always agree on the contents of the stack. The former represents the "system's view" (read: the program logic's view) of the stack, which is tracked by well-bracketed Hoare triples. The latter is handed to the user, and therefore represents the "user's view" of the stack. The predicate $\mathit{stack}_\exists(N)$ is a *persistent* proposition. It merely expresses the knowledge that there exists a stack by the name of $N$, without asserting any ownership over it.

Figure 2 displays the rules that connect well-bracketed Hoare triples to these stack predicates. The HOARE-WBHOARE rule asserts that *not using* ghost stacks is always an option. Indeed, if one is able to verify a triple without leveraging ghost stacks, then the same triple should also hold in the well-bracketed program logic (for an arbitrary mask)! This means that we can combine reasoning in the ordinary program logic with reasoning in the well-bracketed program logic. This allows us to show that well-bracketed programs remain well-bracketed, even in the presence of *innocuous* uses of non-well-bracketed programs, *e.g.*, programs featuring concurrency — we will discuss this further in Section 5.

If one does want to leverage ghost stacks, one can create new stacks using the rule WBHOARE-CREATE-STACK, at any point in the proof of a well-bracketed Hoare triple. There are a few points of interest here regarding the rule WBHOARE-CREATE-STACK. Note how the user of the rule has to pick a predicate $R$ parameterized by the fresh stack name $N$, which the program logic picks when creating the stack — this is why in both antecedents of the rule the name $N$ is universally quantified. Also, the user obtains both $\mathit{stack}_\circ$ and $\mathit{stack}_\bullet$ predicates where the stack is empty. The user can use these predicates, and any other resources and/or invariants, in order to establish $R$. Finally, along $R$, the user of the rule must return the $\mathit{stack}_\bullet$ predicate for *some* initial stack chosen by the user. This is why the fresh name $N$ is not added to the stack mask, unlike the rule WBHOARE-ACCESS-STACK for accessing $\mathit{stack}_\bullet$.

The rule WBHOARE-ACCESS-STACK allows access to the $\mathit{stack}_\bullet$ part of existing stacks. The crucial point here is that the contents of the stack are expected to be returned untouched in the postcondition; this is exactly the reason why we can guarantee well-bracketedness. Any temporary changes made must be undone by the time stack is returned. (This will become more evident later, when we consider the proof of VAE.) The rule WBHOARE-ACCESS-STACK requires $\mathit{stack}_\exists$ as evidence that a stack with the name $N$ has indeed been created before. Importantly, we must show that the name of the stack, *i.e.*, $N$, is not already in the stack mask $\mathcal{O}$ of stacks that are already accessed.

The final rule for using stacks of ghost names is rule WBHOARE-MEND-STACK and it is the exact dual of WBHOARE-ACCESS-STACK. The soundness of this rule relies on the fact that well-bracketed Hoare triples always respect all stacks. The rule WBHOARE-MEND-STACK allows us to temporarily *put the stack back* so as to give access to it to the rest of the program. However, at the same time, it guarantees that the stack will come back to us unchanged. (See the discussion below regarding the proof of VAE.) Note also that since we have $\mathit{stack}_\bullet(N, s)$ in the precondition of the Hoare triple in the consequent of the rule WBHOARE-MEND-STACK, we know that the name $N$ *must be* in the stack mask $\mathcal{O}$ as it cannot both be tracked by the program logic and outside at the same time.

***Reasoning with ghost stacks.*** Ghost stack predicates come with a straightforward ghost theory that corresponds to the intuition given earlier. Figure 3 lists the formal rules that one may leverage on the stack predicates one gets access to by applying some of the rules of Figure 2.

The propositions $\mathit{stack}_\bullet(N, s)$ and $\mathit{stack}_\circ(N, s)$ express exclusive ownership over respective halfs of the ghost stack named $N$. They consequently agree on the stack contents (STACKS-AGREE) and are unique (STACK-SYSTEM-UNIQUE and STACK-USER-UNIQUE). Given $\mathit{stack}_\circ(N, s)$ one may derive a persistent witness that the stack $N$ exists (STACK-EXISTS). Finally, when having full access to both halves of a ghost stack, one may *update* its contents, either by pushing a new ghost name (STACKS-PUSH), or popping the top of the stack (STACKS-POP).

STACKS-AGREE
$stack_\bullet(N, s) * stack_\circ(N, s') \vdash s = s'$

STACK-SYSTEM-UNIQUE
$stack_\bullet(N, s) * stack_\bullet(N, s') \vdash \mathsf{False}$

STACK-USER-UNIQUE
$stack_\circ(N, s) * stack_\circ(N, s') \vdash \mathsf{False}$

STACK-EXISTS
$stack_\circ(N, s) \vdash stack_\exists(N)$

STACKS-PUSH
$stack_\bullet(N, s) * stack_\circ(N, s) \vdash \Rrightarrow stack_\bullet(N, \gamma :: s) * stack_\circ(N, \gamma :: s)$

STACKS-POP
$stack_\bullet(N, \gamma :: s) * stack_\circ(N, \gamma :: s) \vdash \Rrightarrow stack_\bullet(N, s) * stack_\circ(N, s)$

Fig. 3. Reasoning rules on ghost stacks.

### 3.1 The Adequacy Theorem of the Well-Bracketed Program Logic

The adequacy theorem of the well-bracketed program logic, establishing its soundness, is similar to that of the ordinary program logic of Iris. It states that any program $e$ for which we have Hoare triple is safe, *i.e.*, it does not crash, and whenever it terminates to a value $v$, $v$ satisfies the postcondition. Formally, this is stated as the following theorem.

Theorem 3.1 (Adequacy). *Let $e$ be a program, $\sigma$ be the initial state (usually the empty heap in HeapLang), and $\varphi$ a meta-logic predicate on values, i.e., a predicate that does not involve Iris — a Coq predicate of type Val → Prop. The following holds:*[4]

$$\left(\vdash \Rrightarrow \mathsf{SI}(\sigma) * \left(\!\left|\mathit{True}\right|\!\right) e \left(\!\left|x.\ \ulcorner \varphi(x) \urcorner\right|\!\right)\right) \implies \mathit{Safe}_\varphi(\sigma, e)$$

*where* SI *is the predicate reflecting the state of the programming language into Iris resources. In case of HeapLang it reflects the heap in Iris resources so as to define and validate all the rules governing heap points-to propositions.*

Notice that in the adequacy theorem above the postcondition must be a meta-level proposition $\varphi$ ($\ulcorner\cdot\urcorner$ embeds meta-level propositions, *i.e.*, Prop in Coq into Iris's *iProp*), as otherwise the predicate $\mathit{Safe}_\varphi$ does not make sense at the meta level. We emphasize that the statement of the adequacy theorem above is the same as for the standard Iris program logic — recall that the point of the well-bracketed program logic is to prove a wider class of specifications, not to change the meaning of specifications.

Note how the statement of the adequacy theorem above requires the user of the theorem to provide resources for the initial state of the program (rather trivial when we start in the empty heap). It does not, however, require the initialization resources for ghost stacks. This is because the initialization of ghost stacks, always with the empty collection of stacks, is taken care of as part of the proof of the adequacy theorem. Apart from this initialization of the collection of ghost stacks, the proof of the adequacy theorem above follows directly from the adequacy theorem of the ordinary program logic of Iris. The reason why we always initialize with the empty collection of ghost stacks is because, whereas there are cases where it makes sense to speak of executing a program on a non-empty heap, it makes no sense to start a proof with an existing collection of stacks. That is, proofs should always allocate their required stacks using the rule wbHoare-create-stack.

***The Soundness of the Well-Bracketed Program Logic.*** Note again that both the statement of the adequacy theorem (the implication), and the predicate $\mathit{Safe}_\varphi$ are statements in the meta-theory (*i.e.* Coq in our Coq formalization). It is only the initialization of the resources for the initial state

[4]In the Coq formalization, following Iris's terminology, the Safe predicate is called *adequate*.

and the proof of well-bracketed Hoare triples that take place in Iris. (These facts are syntactically indicated by the Iris turnstile being delimited by the surrounding parentheses.) In other words, to trust that the program is indeed safe, one only needs to trust the soundness of the meta theory (Coq) and nothing about Iris, its construction, the rules of the ordinary program logic of Iris, or the rules of the well-bracketed program logic. Hence, the adequacy theorem above does indeed establish soundness of well-bracketed program logic.

Following the strategy of Iris's ordinary program logic (see our Coq formalization), we in fact prove a stronger version of the adequacy theorem for the well-bracketed program logic which implies, and is used to prove, the adequacy theorem detailed above. This stronger adequacy theorem is also used to prove the adequacy theorem that is used in Section 6 to establish well-bracketedness as a trace property.

### 3.2 Proof of VAE

Now we have all the necessary building blocks to prove correctness and well-bracketedness of VAE. The specification VAE-spec we ascribe to VAE is similar to the specification awkward-spec of the Awkward example we saw earlier, except it is expressed in terms of well-bracketed Hoare triples instead of Iris's ordinary Hoare triples.

$$\begin{array}{l}(\!|\mathsf{True}|\!)\\ \quad \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\ r \leftarrow 0; f\ (); r \leftarrow 1; f\ (); !\, r\\ (\!|g.\ \forall f.\ (\!|(\!|\mathsf{True}|\!)\ f\ ()\ (\!|x.\ x = ()|\!)|\!)\ g\ f\ (\!|x.\ x = 1|\!)|\!)\end{array} \tag{VAE-spec}$$

In the proof we first allocate the location $\ell$, and obtain $\ell \mapsto 0$. We then allocate an instance $\gamma$ of the one-shot resource using the rule MAKE-ONE-SHOT, just as in the proof of the Awkward example. At this point, we use the rule WBHOARE-CREATE-STACK to create a new stack and pick the invariant VAE-inv below, together with $stack_\exists(N)$, as the predicate $R$. We use the resources we have at hand together with the rule STACKS-PUSH to obtain $stack_\circ(N, [\gamma])$ and $stack_\bullet(N, [\gamma])$ in order to establish the invariant VAE-inv below. Subsequently, we return $stack_\bullet(N, [\gamma])$ to the program logic.

$$VAEInv \triangleq \boxed{\exists \gamma, s.\ stack_\circ(N, \gamma :: s) * (\ell \mapsto 0 * \mathsf{pending}(\gamma)) \vee (\ell \mapsto 1 * \mathsf{shot}(\gamma))}^{\mathcal{N}_{\mathsf{vae}}} \tag{VAE-inv}$$

Note that the invariant VAE-inv is similar to the awkward-inv invariant, except for the fact that the name of the one-shot instance is not fixed. Instead, it is the name on the top of the stack $N$. What remains is to show the following for the body of the closure returned by VAE:

$$(\!|VAEInv * stack_\exists(N) * (\!|\mathsf{True}|\!)\ f\ ()\ (\!|x.\ x = ()|\!)|\!)\ \ell \leftarrow 0; f\ (); \ell \leftarrow 1; f\ (); !\, \ell\ (\!|x.\ x = 1|\!) \tag{4}$$

Here, we wish to reason about the well-bracketed nature of the calls to the closure. Hence, we proceed by applying the rule WBHOARE-ACCESS-STACK to obtain $stack_\bullet(N, s')$ for *some stack* $s'$, which leaves us with having to prove:

$$\begin{array}{l}(\!|VAEInv * stack_\bullet(N, s') * (\!|\mathsf{True}|\!)\ f\ ()\ (\!|x.\ x = ()|\!)|\!)\\ \quad \ell \leftarrow 0; f\ (); \ell \leftarrow 1; f\ (); !\, \ell\\ (\!|x.\ x = 1 * stack_\bullet(N, s')|\!)^{\{N\}}\end{array} \tag{5}$$

Here, the stack $s'$ is just some stack for which we only know the following by the fact that the invariant VAE-inv holds: $s'$ is non-empty, and furthermore, the one-shot resource whose name is on top of the stack reflects the value stored in $\ell$. This makes intuitive sense: we have to show the spec (4) without knowing whether or not it is the first call to the VAE closure, or whether it is a nested call, *i.e.*, a call (indirectly) made by the closure itself, because it was passed a function $f$ that

calls the closure itself. (See Section 3.3 for more details on the latter situation.) As we will see in the rest of the proof, during the part where the program writes 0 to $\ell$, we create a new instance of the one-shot resource and push its name on the stack. On the other hand, when the program writes 1 to $\ell$, we pop the ghost name on top of the stack. To make the intuition we develop here clearer, let us call the writing of 0 in the VAE closure "opening the bracket" and writing 1, "closing the bracket". Following this intuitive terminology, we can describe the contents of the stack $s'$ as follows. The stack $s'$, in addition to the name of the original one-shot resource allocated at the beginning of the proof (at the bottom of the stack), stores all the names of one-shot resources corresponding to the "opening the bracket" part of all the parent calls to the VAE closure (in case of nested calls), with the most recent parent call that has not yet "closed the bracket" on top of the stack. In other words, if the current call to the VAE closure is not a nested call, *i.e.*, all previous calls to the closure have already terminated, the stack $s'$ is a singleton, storing only the ghost name of the original one-shot resource that was allocated at the beginning of the proof.

To proceed with proving (5) above, since the store operation $\ell \leftarrow 0$ is atomic, we can open the invariant (by isolating the store operation using the WBHOARE-BIND) to obtain $\mathit{stack}_\circ(N, \gamma_1 :: s)$ which we use along with the rule STACKS-AGREE to unify $s'$ with $\gamma_1 :: s$, which leaves us having to prove the following (note how the postcondition of (6) implies the invariant as required):[5]

$$\begin{array}{l} \left\langle \begin{array}{l} \mathit{VAEInv} * \mathit{stack}_\bullet(N, \gamma_1 :: s) * \mathit{stack}_\circ(N, \gamma_1 :: s) * \\ (\ell \mapsto 0 * \mathsf{pending}(\gamma_1)) \vee (\ell \mapsto 1 * \mathsf{shot}(\gamma_1)) \end{array} \right\rangle \\ \quad \ell \leftarrow 0 \\ \left\langle \begin{array}{l} x.\ x = () * (\mathsf{pending}(\gamma_1) \vee \mathsf{shot}(\gamma_1)) * \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s) * \\ \quad \mathit{stack}_\circ(N, \gamma_2 :: \gamma_1 :: s) * \ell \mapsto 0 * \mathsf{pending}(\gamma_2)) \end{array} \right\rangle^{\{N\}} \end{array} \tag{6}$$

and the following for the code after the store operation:

$$\begin{array}{l} \left(\!\left| \mathit{VAEInv} * (\mathsf{pending}(\gamma_1) \vee \mathsf{shot}(\gamma_1)) * \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s) * (\!|\mathsf{True}|\!)\ f\ ()\ (\!|x.\ x = ()|\!) \right|\!\right) \\ \quad f\ (); \ell \leftarrow 1; f\ (); !\ell \\ (\!|x.\ x = 1 * \mathit{stack}_\bullet(N, \gamma_1 :: s)|\!)^{\{N\}} \end{array} \tag{7}$$

Observe that we have to show two triples because we have applied the WBHOARE-BIND rule (from bottom to top) and that we "made a good choice" of $\Psi$ in WBHOARE-BIND, which will allow us to complete the proof. Note further how at this point in the proof, we do not know whether $\ell$ stores 0 or 1 prior to our store operation; hence the only thing we can retain is $(\mathsf{pending}(\gamma_1) \vee \mathsf{shot}(\gamma_1))$ (cf. the precondition of (7)). Nonetheless, by creating a new instance $\gamma_2$ of the one-shot resource we have managed to reestablish the invariant, this time by taking $\gamma$ to be $\gamma_2$ and the stack $s$ to be $\gamma_1 :: s$. In addition, since we keep hold of $\mathit{stack}_\bullet$, we remember the exact contents of the stack even after reestablishing the invariant. To show that (7) holds, we use the WBHOARE-BIND and WBHOARE-FRAME (to frame $\mathit{VAEInv}$ and $\mathsf{pending}(\gamma_1) \vee \mathsf{shot}(\gamma_1)$) rules, which leaves us having to prove the following:

$$\left(\!\left| \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s) * (\!|\mathsf{True}|\!)\ f\ ()\ (\!|x.\ x = ()|\!) \right|\!\right) f\ ()\ (\!|x.\ x = () * \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s)|\!)^{\{N\}} \tag{8}$$

[5]Note that we retain the invariant as it is persistent. Also, in the Hoare triple (6) we implicitly frame the specs for $f$.

and the following for the code after calling $f$:

$$\left\langle \begin{array}{l} \mathit{VAEInv} * (\mathsf{pending}(\gamma_1) \lor \mathsf{shot}(\gamma_1)) * \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s) * \\ \langle\!|\mathsf{True}|\!\rangle\ f\ ()\ \langle\!|x.\ x = ()|\!\rangle \end{array} \right\rangle \quad \ell \leftarrow 1; f\ (); !\,\ell \quad \langle\!|x.\ x = 1 * \mathit{stack}_\bullet(N, \gamma_1 :: s)|\!\rangle^{\{N\}} \tag{9}$$

To establish (8) we simply use the rule wbHoare-mend-stack and then it suffices to show the following trivial triple:

$$\left\langle\!\left|\langle\!|\mathsf{True}|\!\rangle\ f\ ()\ \langle\!|x.\ x = ()|\!\rangle\right|\!\right\rangle\ f\ ()\ \langle\!|x.\ x = ()|\!\rangle$$

We proceed with proving (9) by again using the wbHoare-bind rule to isolate the store operation. We then use the rule wbHoare-inv-open to open the invariant as before, this time unifying the stacks of $\mathit{stack}_\bullet$ (which we have) and $\mathit{stack}_\circ$ (which we obtain from the invariant). From this we learn that the current instance name of the one-shot resource tracking the state of $\ell$ is in fact $\gamma_2$. Thus, we have to show:

$$\left\langle \begin{array}{l} \mathit{VAEInv} * (\mathsf{pending}(\gamma_1) \lor \mathsf{shot}(\gamma_1)) * \mathit{stack}_\bullet(N, \gamma_2 :: \gamma_1 :: s) * \mathit{stack}_\circ(N, \gamma_2 :: \gamma_1 :: s) * \\ (\ell \mapsto 0 * \mathsf{pending}(\gamma_2)) \lor (\ell \mapsto 1 * \mathsf{shot}(\gamma_2)) \end{array} \right\rangle \quad \ell \leftarrow 1 \quad \langle\!|x.\ x = () * \mathit{stack}_\bullet(N, \gamma_1 :: s) * \mathit{stack}_\circ(N, \gamma_1 :: s) * \ell \mapsto 1 * \mathsf{shot}(\gamma_1)|\!\rangle^{\{N\}} \tag{10}$$

and the following for the code after the store operation:

$$\left\langle\!\left|\mathit{VAEInv} * \mathit{stack}_\bullet(N, \gamma_1 :: s) * \mathsf{shot}(\gamma_1) * \langle\!|\mathsf{True}|\!\rangle\ f\ ()\ \langle\!|x.\ x = ()|\!\rangle\right|\!\right\rangle \quad f\ (); !\,\ell \quad \langle\!|x.\ x = 1 * \mathit{stack}_\bullet(N, \gamma_1 :: s)|\!\rangle^{\{N\}} \tag{11}$$

To prove (10), we use the rule stacks-pop to pop $\gamma_2$ from the stack. Again, at this point, we do not know what the value of $\ell$ is. As we discussed earlier, it depends on whether the function $f$ has called the closure of VAE or not. Nonetheless, we are performing a store operation storing 1 into $\ell$. Furthermore, whether we have obtained $\mathsf{pending}(\gamma_2)$ or $\mathsf{shot}(\gamma_2)$ from the invariant, either resource is not of any use to us anymore. Hence, we discard it.[6] At this point, the only missing piece (to ensure that we have indeed applied the wbHoare-bind rule correctly) is to obtain $\mathsf{shot}(\gamma_1)$. However, we have $\mathsf{pending}(\gamma_1) \lor \mathsf{shot}(\gamma_1)$, left over from the last time we opened the invariant. In either case, we can obtain $\mathsf{shot}(\gamma_1)$ — in the case of the left disjunct we simply use the rule shoot. Note that since $\mathsf{shot}(\gamma_1)$ is persistent we can both use it to reestablish the invariant, and at the same time pass it to the rest of the program, *i.e.*, the postcondition of (10), and the precondition of (11). To prove the correctness of the call to $f$ in (11) we simply repeat the argument as in the previous call to $f$. This leaves us to show the following in order to finish the proof:

$$\langle\!|\mathit{VAEInv} * \mathit{stack}_\bullet(N, \gamma_1 :: s) * \mathsf{shot}(\gamma_1)|\!\rangle\ !\,\ell\ \langle\!|x.\ x = 1 * \mathit{stack}_\bullet(N, \gamma_1 :: s)|\!\rangle^{\{N\}} \tag{12}$$

Here we can access the invariant again, and since we know that the stack must be $\gamma_1 :: s$ and, moreover, we have $\mathsf{shot}(\gamma_1)$ we can exclude the case where $\ell$ stores 0, and thereby finish the proof!

[6] Iris is an affine separation logic. Hence, we can always discard resources as necessary.

Note how, save for the use of the stack, the overarching argument of the proof of VAE resembles the proof of the Awkward example. In a sense, our methodology of using the stack has decoupled the part of the reasoning that relates to well-bracketedness from the rest of the proof.

### 3.3 VAE Applied to Itself

As we discussed earlier, the intricacy of reasoning about VAE arises from the fact that the callback $f$ can itself invoke the closure returned by VAE. Here is a simple example where this happens:

$$\mathsf{let}\, g = \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\ r \leftarrow 0; f\ (); r \leftarrow 1; f\ (); !\, r\, \mathsf{in}\, g\ (\lambda\_.\ g\ (\lambda\_.\ ()); ()) \quad \text{(VAE-self-applied)}$$

It is now quite simple to prove that VAE-self-applied returns 1. We use the specification VAE-spec above, which only requires us to show the following:

$$\left(\!\!\left|\forall f.\ \left(\!\!\left|\mathsf{True}\right|\!\!\right) f\ ()\ \left(\!\!\left|x.\ x = ()\right|\!\!\right)\right|\!\!\right) g\ f\ \left(\!\!\left|x.\ x = 1\right|\!\!\right)\Big) g\ (\lambda\_.\ g\ (\lambda\_.\ ()); ())\ \left(\!\!\left|x.\ x = 1\right|\!\!\right) \tag{13}$$

To prove (13) we simply use the Hoare triple in the precondition of (13). After this step, we only need to show:[7]

$$\left(\!\!\left|\forall f.\ \left(\!\!\left|\mathsf{True}\right|\!\!\right) f\ ()\ \left(\!\!\left|x.\ x = ()\right|\!\!\right)\right|\!\!\right) g\ f\ \left(\!\!\left|x.\ x = 1\right|\!\!\right)\Big) g\ (\lambda\_.\ ()); ()\ \left(\!\!\left|x.\ x = ()\right|\!\!\right) \tag{14}$$

To prove (14) we use the WBHOARE-BIND rule together with the Hoare triple in the precondition once more. Thus, there remains only two things to show: (1) that the program, after the call to $g$, returns $()$, which is obvious, and (2) the following, which is again trivial:

$$\left(\!\!\left|\mathsf{True}\right|\!\!\right)\ (\lambda\_.\ ())\ ()\ \left(\!\!\left|x.\ x = ()\right|\!\!\right) \tag{15}$$

Applying the Adequacy theorem yields that running the program VAE-self-applied (in terms of the operational semantics) starting in the empty heap will always result in 1.

### 3.4 The Semantic Essence of Well-Bracketedness

An interesting aspect of our well-bracketed program logic is the fact that we have instantiated it with HeapLang, a *concurrent* programming language, with the caveat that the program logic does not have a rule for reasoning about forking of threads. In Section 5 we will show how this allows us to reason about well-bracketedness even for programs that use concurrency (in an innocuous way). This means that, the well-bracketed program logic does not rely on a syntactic restriction of programs to capture well-bracketedness. Instead, it captures the notion of well-bracketedness at a semantic level. That is, we can use the well-bracketed Hoare logic to establish well-bracketedness, in addition to the usual partial correctness (safety) property of Iris Hoare triples. For further justification of this claim, see Section 6, where we show that well-bracketed Hoare triples enforce well-bracketedness of calls and returns, stated as an intensional trace property.

## 4 THE INTERNALS OF THE WELL-BRACKETED PROGRAM LOGIC

In this section we begin the process of explaining how the well-bracketed program logic is constructed internally. We explain the internal construction of the well-bracketed program logic by peeling off layers of abstraction, one after the other. We first explain how well-bracketed Hoare triples are defined in terms of well-bracketed weakest preconditions. We then explain how well-bracketed weakest preconditions themselves are defined in terms of ordinary Iris weakest preconditions. To define well-bracketed weakest preconditions we use a *ghost theory of stack collections*. Later on, in Section 9 we present the details of the ghost theory of stack collections.

[7] Since (well-bracketed) Hoare triples are persistent they can be freely duplicated and also moved in and out of preconditions just like ordinary Hoare triples of Iris; see Birkedal and Bizjak [2017] for more details.

This also includes the theory of stacks we have seen before governing predicates $\mathit{stack}_\bullet$, $\mathit{stack}_\circ$, and $\mathit{stack}_\exists$. We conclude this section with a discussion of how well-bracketed weakest precondition rules can be derived from corresponding Iris weakest precondition rules.

### 4.1 The Definition of the Well-Bracketed Program Logic

In Iris, Hoare triples are defined in terms of weakest preconditions as follows:

$$\{P\}\, e\, \{\Phi\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\}\right)$$

Here, $\Box$ is the persistently modality which enforces that Hoare triples are persistent. In practice this means that all the non-persistent resources necessary for correctness of $e$ must be captured by $P$. In other words, the precondition is not underapproximating the necessary conditions for correctness of $e$. The connective $-\!\!*$, pronounced magic wand, or simply wand, is the separating implication; it is to separating conjunction $*$, what implication $\Longrightarrow$ is to ordinary conjunction $\wedge$. We define well-bracketed Hoare triples in terms of well-bracketed weakest preconditions, in the same way ordinary Hoare triples are defined in terms of ordinary weakest preconditions.

$$(\!|P|\!)\, e\, (\!|\Phi|\!)^{\mathcal{O}} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wbwp}\, e\, (\!|\Phi|\!)^{\mathcal{O}}\right)$$

Well-bracketed weakest preconditions themselves are defined in terms of ordinary Iris weakest preconditions.

$$\mathsf{wbwp}\, e\, (\!|\Phi|\!)^{\mathcal{O}} \triangleq \forall \mathcal{S}.\ \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{x.\ \exists \mathcal{S}'.\ \mathcal{S} \subseteq \mathcal{S}' * \mathsf{AllStacksExcept}(\mathcal{S}', \mathcal{O}) * \Phi(x)\} \quad \text{(wbwp-definition)}$$

Here $\mathcal{S}$ is a mapping from stack names to stacks. The proposition $\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O})$ asserts that the map $\mathcal{S}$ describes *all* stacks in existence, and, moreover, that it owns all propositions $\mathit{stack}_\bullet(N, \mathcal{S}(N))$ for all stacks except for those whose names are in $\mathcal{O}$. Note that we allow the map of stacks to grow from $\mathcal{S}$ to $\mathcal{S}'$. This is to allow the program logic to allocate more stacks as necessary. However, we require that in the end $\mathcal{S} \subseteq \mathcal{S}'$ (as sets of pairs), meaning that all the stacks that existed in $\mathcal{S}$ must remain exactly the same in $\mathcal{S}'$!

All the rules governing the AllStacksExcept predicate which are necessary for validating the rules of the well-bracketed program logic are given in Figure 4. The rule STACK-EXISTS-IN states that an existing stack must always be in the map of all stacks. Note how the stack map must always agree with $\mathit{stack}_\circ$ whenever the stack is tracked by AllStacksExcept. On the other hand, the set $\mathcal{O}$ in AllStacksExcept are exactly the set of stacks that are *not* tracked by AllStacksExcept, as evidenced by the rules STACK-SYSTEM-IS-OUT and CHANGE-OUT-STACK. The latter states that the map $\mathcal{S}$ in $\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O})$ can be *arbitrarily* updated for any stack that is not tracked by AllStacksExcept. The rules STACK-TAKE-OUT and STACK-PUT-BACK respectively allow us to take $\mathit{stack}_\bullet$ propositions out of AllStacksExcept and put them back (as long as they agree) with the map of stacks. These rules together with CHANGE-OUT-STACK allow us to take a stack out, update it *arbitrarily* and then put it back. There is no restriction on how the stack may evolve as far as the theory around AllStacksExcept predicate is concerned. The stack discipline is instead enforced in the definition of well-bracketed weakest preconditions wbwp-definition as it requires in the postcondition that the map of stacks has only grown but not changed in any other way (captured by the condition: $\mathcal{S} \subseteq \mathcal{S}'$). Finally, the rule create-stack allows us to create fresh stacks whose $\mathit{stack}_\bullet$ part initially remains within the AllStacksExcept predicate.

MASK-SUBSET-DOM
$$\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \mathcal{O} \subseteq \mathrm{dom}(\mathcal{S})$$

STACK-EXISTS-IN
$$\mathit{stack}_\exists(N) * \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash N \in \mathrm{dom}(\mathcal{S})$$

STACK-USER-NOT-OUT
$$\frac{N \notin \mathcal{O}}{\mathit{stack}_\circ(N, s) * \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \mathcal{S}(N) = s}$$

STACK-SYSTEM-IS-OUT
$$\mathit{stack}_\bullet(N, s) * \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash N \in \mathcal{O}$$

STACK-TAKE-OUT
$$\frac{N \in \mathrm{dom}(\mathcal{S}) \setminus \mathcal{O}}{\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \exists s.\ \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O} \cup \{N\}) * \mathit{stack}_\bullet(N, s)}$$

STACK-PUT-BACK
$$\frac{\mathcal{S}(N) = s}{\mathit{stack}_\bullet(N, s) * \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O} \setminus \{N\})}$$

CHANGE-OUT-STACK
$$\frac{N \in \mathcal{O}}{\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \mathsf{AllStacksExcept}(\mathcal{S}[N \mapsto s'], \mathcal{O})}$$

CREATE-STACK
$$\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O}) \vdash \Rrightarrow \exists N.\ N \notin \mathrm{dom}(\mathcal{S}) * \mathsf{AllStacksExcept}(\mathcal{S}[N \mapsto \emptyset], \mathcal{O}) * \mathit{stack}_\circ(N, \emptyset)$$

Fig. 4. Rules governing AllStacksExcept.

### 4.2 Deriving Well-Bracketed Weakest Precondition Rules

The well-bracketed program logic is constructed on top of the existing Iris program logic. This has two major implications. The first implication is that whenever a rule in the well-bracketed program logic has a counterpart in the Iris program logic (this is the case for all rules except for HOARE-WBHOARE, WBHOARE-CREATE-STACK, WBHOARE-ACCESS-STACK, and WBHOARE-MEND-STACK), then we can prove soundness of that rule in the well-bracketed program logic by just applying the proof of the corresponding rule in Iris's program logic. The second implication is that we can combine reasoning steps of the two program logics, as witnessed by rule HOARE-WBHOARE. Below, we will discuss how the well-racketed program logic's rules are derived.

As for the bespoke rules of the well-bracketed program logic, *i.e.*, WBHOARE-CREATE-STACK, WBHOARE-ACCESS-STACK, and WBHOARE-MEND-STACK, it is not difficult to see that the rules presented in Figure 4 allow us to validate all of them. The interested reader is referred to the rather short proofs of these rules in our Coq development. We divide the rest of the rules into two groups, those general rules that apply to all instantiations of the logic, *e.g.*, WBHOARE-CONSEQUENCE, WBHOARE-INV-OPEN, WBHOARE-BIND, *etc.*, and those that reflect the semantics of the basic building blocks of the language, *e.g.*, WBHOARE-ALLOC, WBHOARE-LOAD, *etc.*

The general rules are proven directly by unfolding the definition of Hoare triples (both well-bracketed Hoare triples and the underlying ones), and the definition of well-bracketed weakest preconditions. The proof then follows using the corresponding Iris rule for Iris's weakest preconditions. To derive the rules for the basic building blocks of the language, we simply apply the rule HOARE-WBHOARE to lift the operation's Hoare triple in the underlying program logic to the well-bracketed program logic.

Proving the rule Hoare-wbHoare is straightforward. We just need to unfold the definition of well-bracketed triples. Intuitively, if an expression does not interact with other parts of the program that rely on well-bracketedness (as its correctness does not require taking ghost stacks into account), then it cannot break well-bracketedness either, because it does not have access to ghost stacks.

The rule Hoare-wbHoare may seem surprising in that it seems to allow "side-stepping" the well-bracketed logic completely. That is, by carrying all reasoning in the ordinary program logic, and then lifting it to the well-bracketed logic only at the very end. However, this is not the case, in particular in the case of higher-order programs (such as VAE)! Since proving VAE-spec requires both *establishing* and *assuming* well-bracketed Hoare triples, one cannot simply use Hoare-wbHoare to get a well-bracketed specification of VAE "for free". The rule Hoare-wbHoare is nevertheless an interesting rule, in that it allows us to prove that innocuous uses of non-well-bracketed programs (*e.g.*, programs featuring concurrency) can be safely combined with other well-bracketed programs, as we discuss next in Section 5.

## 5 WELL-BRACKETED REASONING AND CONCURRENCY

One difference between the current work and other works that have verified correctness of VAE [Dreyer et al. 2010; Georges et al. 2021; Skorstengaard et al. 2019] is that others have done so using a logical relations model. Logical relations models, put succinctly, are models of the entire programming language stating particular properties that *all* (well-typed) programs in the language satisfy. This makes them useful for showing language-level properties, *e.g.*, that all programs terminate (normalization), type-safety, well-bracketedness, *etc.* By contrast, in this paper, we present a *program logic* which expresses well-bracketedness of programs without saying anything about the other programs in the programming language. Indeed, in the HeapLang programming language we use here, one can easily use concurrency to write non-well-bracketed programs. As we will discuss in Section 7, our program logic can also be used to construct logical relations models capable of showing that all programs are well-bracketed (if this is guaranteed by the programming language in question). In this section we demonstrate how the well-bracketed program logic can be used to show well-bracketedness of certain function calls even if part of the program is not well-bracketed. Before that, however, we will first demonstrate, via a simple example, how concurrency *can* break well-bracketedness.

***Concurrency Breaking Well-Bracketedness.*** As an example of concurrency breaking well-bracketedness consider the following program:

$$\begin{array}{l}\mathsf{let}\, g = \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\, r \leftarrow 0; f\ (); r \leftarrow 1; f\ (); !\, r\, \mathsf{in} \\ \mathsf{fork}\, \{g\ (\lambda\_.\ ())\}\, ; g\ (\lambda\_.\ ()) \end{array} \qquad \text{(bad-concurrency)}$$

It takes the VAE closure, $g$ here, and calls it twice *concurrently* (with a trivial argument). Here, we assume that programs' execution terminates when all its threads terminate. When the program terminates, it returns the result of its *main* thread. That is, the final result of the program bad-concurrency above is the result of the call to $g$ in the main thread. Under certain schedulings, the program bad-concurrency above can produce 0 as its final result. Here is an example:

I. Schedule all of the calls to $g$ in the main thread until before the very last expression which reads and returns the value of the reference.
II. Switch to the forked-off thread and execute $g$ until it writes 0 into the reference.
III. Switch back to the main thread to read 0 and terminate the main thread.
IV. Switch back again to the forked-off thread and run it to completion.

This shows that concurrency can break well-bracketedness in general. Indeed, the reason for the program bad-concurrency above returning 0 under the above scheduling is that this scheduling

breaks well-bracketedness. That is, under this scheduling, the call to $g$ in the forked-off thread terminates *after* its caller, the main thread program.

***Innocuous Concurrency.*** Despite concurrency breaking well-bracketedness, as discussed above, we present here an example of innocuous concurrency and show that we can use our well-bracketed program logic to reason about such cases. The example innocuous-concurrency below is a simple case of innocuous concurrency that uses VAE. Many schedulings of the program innocuous-concurrency below are not well-bracketed as per our definition. That is, a function execution outlives it caller, but nonetheless this does not impact well-bracketedness of the calls and returns to the VAE closure. The program innocuous-concurrency is as follows:

$$
\begin{array}{l}
\mathsf{let}\ g = \\
\quad \mathsf{let}\ \mathit{first} = \mathsf{ref}(\mathsf{true})\ \mathsf{in}\ \mathsf{let}\ \mathit{res} = \mathsf{ref}(\mathit{None})\ \mathsf{in} \\
\quad \mathsf{let}\ \mathsf{rec}\ \mathit{wait}() = \mathsf{if}\ !\,\mathit{res} = \mathit{None}\ \mathsf{then}\ \mathit{wait}\ ()\ \mathsf{else}\ ()\mathsf{in} \\
\quad \lambda\_.\ \mathsf{if}\ !\,\mathit{first}\ \mathsf{then}\ \mathit{first} \leftarrow \mathsf{false}; \mathsf{fork}\,\{\mathit{res} \leftarrow \mathit{Some}(\mathit{fact}\ 1000000)\}\ \mathsf{else}\ \mathit{wait}\ () \\
\mathsf{in} \\
\mathsf{let}\ h = \mathsf{let}\ r = \mathsf{ref}(0)\ \mathsf{in}\ \lambda f.\ r \leftarrow 0; f\ (); r \leftarrow 1; f\ (); !\,r\ \mathsf{in} \\
h\ g
\end{array}
\qquad \text{(innocuous-concurrency)}
$$

This program applies the VAE closure to a function whose execution depends on whether it is the first time it is executed or not. The first execution, *i.e.*, when the reference *first* stores true, forks a thread that computes the factorial of a large number and stores the result in he reference *res*. The subsequent calls will simply wait for the computation of the large factorial to terminate, *i.e.*, for *res* to not be *None*. Clearly, in many runs the factorial function in the forked thread does not terminate before the second call to function $g$. (The operational semantics of HeapLang makes no assumptions regarding scheduling.) Hence, this program *does* exhibit non-well-bracketed behavior as per our definition. Despite this, innocuous-concurrency still behaves well-bracketed, at least as far as VAE is concerned. We can prove this by showing the following:

$$(\!|\mathsf{True}|\!)\ h\ g\ (\!|x.\ x = 1|\!) \tag{16}$$

where $h$ and $g$ are as in innocuous-concurrency above. To prove (16) above, by using the VAE-spec from earlier, it suffices to show the following:

$$(\!|\mathsf{True}|\!)\ g\ ()\ (\!|x.\ x = ()|\!) \tag{17}$$

We can prove (17) above using the rule HOARE-WBHOARE, which leave us to prove the following Iris triple:

$$\{\mathsf{True}\}\ g\ ()\ \{x.\ x = ()\} \tag{18}$$

The Hoare triple (18) above, however, is just a rather easy exercise in concurrent separation logic. It can be proven by simply using the following invariant after allocating the two references *first* and *res*:

$$\boxed{\exists b \in \{\mathsf{true}, \mathsf{false}\}\ .\ \mathit{first} \mapsto b * (\mathit{res} \mapsto \mathit{None} \vee \exists n \in \mathbb{N}.\ \mathit{res} \mapsto \mathit{Some}(n))}^{\mathcal{N}_{\mathsf{fact}}}$$

## 6 WELL-BRACKETEDNESS AS A TRACE PROPERTY

Our well-bracketed program logic allows proving that VAE returns 1. This is then understood as a witness that our program logic is strong enough to reason about programs that make essential use of well-bracketedness of calls and returns. We now make this argument very explicit by showing that

well-bracketed Hoare triples *imply well-bracketedness of calls and returns, stated as an intensional trace property*.

To this end, we apply the technique introduced by Birkedal et al. [2021], in which one can derive intensional trace properties from separation logic specifications as so-called "free theorems". (We do not have space to recall all the details of Birkedal et al. [2021], so this section is necessarily a bit high-level.) Using this technique, we show that given an arbitrary program specified using a well-bracketed Hoare triple, the trace of its interactions with a client (as "call" and "return" events) necessarily belongs to a language of well-bracketed traces. (Well-bracketed traces being defined inductively using a grammar matching corresponding call and return events.) In other words, our program logic not only allows proving that VAE returns 1, but is also powerful enough to explicitly capture the well-bracketedness of every function call and return across a higher-order, reentrant specification such as that of VAE.

We first port the general framework of Birkedal et al. [2021] to work with our program logic. We then consider *an arbitrary expression* $e$ satisfying the specification of VAE, dubbed *VAESpec* and parameterized by the implementation $e$ and initial resources $P_0$:

$$\mathit{VAESpec}(e, P_0) \triangleq (\!|P_0|\!)\ e\ \left(\!\!\left| g.\ \forall f.\ \left(\!\!\left|\{\mathsf{True}\}\ f\ ()\ \{x.\ x = ()\}\right|\!\!\right)\ g\ f\ (\!|x.\ x = 1|\!)\right|\!\!\right) \qquad (\mathit{VAESpec})$$

Note that, since this specification uses well-bracketed triples in a higher-order fashion, the earlier remark of Section 4.2 applies. Such a specification cannot be derived from a standard specification that would be stated using standard Hoare triples.

Next, the central idea is to consider an instrumented version of $e$ with additional calls to primitives that *record events* on a global linear trace collected during the execution. Since we want to inspect calls and returns to the VAE closure, the instrumentation is defined as follows. We use the fresh primitive to generate a fresh identifier (a string) and record an event on the trace with this identifier: if fresh(call) returns $\tau$, then the event $\langle \tau, \mathsf{call}\rangle$ has been added to the trace. We then use emit to simply add an event to the trace: $\mathsf{emit}(\tau, \mathsf{ret})$ adds the event $\langle \tau, \mathsf{ret}\rangle$. In other words, for each call to the VAE closure we emit a pair of call (before) and ret events with a matching unique identifier $\tau$.

$$\mathit{instrument}(e) \triangleq \mathsf{let}\ g = e\ \mathsf{in}\ \lambda f.\ \mathsf{let}\ \tau = \mathsf{fresh}(\mathsf{call})\ \mathsf{in}\ \mathsf{let}\ r = g\ f\ \mathsf{in}\ \mathsf{emit}(\tau, \mathsf{ret}); r$$

For any $e$ that satisfies *VAESpec*, the goal is now to prove that $\mathit{instrument}(e)$ produces a trace of call and ret events that is well-bracketed. The language $\mathcal{L}_{seq}$ of well-bracketed traces (as sequences of program values) is easily defined. We let $\mathcal{L}_{seqfull} \subseteq \mathit{Val}^*$ be the language of well-bracketed complete traces defined below, and then take $\mathcal{L}_{seq}$ to be its prefix closure.

$$\mathcal{L}_{seqfull} ::= \langle \tau, \mathsf{call}\rangle \cdot \mathcal{L}_{seqfull} \cdot \langle \tau, \mathsf{ret}\rangle \mid \mathcal{L}_{seqfull} \cdot \mathcal{L}_{seqfull} \mid \varepsilon \qquad (\tau \in \mathit{Tag} \triangleq \mathit{String})$$

Following the methodology from Birkedal et al. [2021] we can then prove that $\mathit{instrument}(e)$ satisfies *VAESpec* while emitting traces in $\mathcal{L}_{seq}$, as captured by:

$$\mathit{VAESpec}(e, \mathsf{True}) \twoheadrightarrow \mathit{VAESpec}(\mathit{instrument}(e), \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{seq})) \qquad \text{(instrument-trace)}$$

Technically speaking, we use Birkedal et al. [2021]'s traceInv predicate to indicate that $\mathcal{L}_{seq}$ is the *trace invariant* which has to be upheld during all executions. Establishing instrument-trace is the main proof step, where we make use of the key reasoning principles provided by our well-bracketed program logic.

As a last step, we combine the instrument-trace lemma with the Adequacy Theorem of Birkedal et al. [2021], adapted to our program logic. (This adequacy theorem, in our case, is derived from the stronger variant of Adequacy we mentioned in Section 3.1, which instead of $\mathit{Safe}_\varphi$ concludes trace properties of the program). We obtain a theorem that does not depend on auxiliary predicates such

as traceInv, and which expresses that emitted events belong to the expected language of traces directly according to the operational semantics. (We point to the Coq development for the exact statement of this theorem.)

It is worth noting that the present result holds for any implementation satisfying *VAESpec*, and for any client of this implementation satisfying the adequate well-bracketed Hoare triple. As such, we can instantiate the above theorem by implementations or clients that, *e.g.*, internally use concurrency, as long as they satisfy the required specifications: we still get that their interaction trace is a well-bracketed sequence of calls and returns.

## 7 WELL-BRACKETED LOGICAL RELATIONS MODELS

So far we have shown that our well-bracketed program logic, which has only been instantiated with HeapLang, a concurrent language, can be used to reason about well-bracketed programs in a language that does not guarantee well-bracketedness. We furthermore showed, in Section 5, that our program logic can be employed to reason about well-bracketedness even in the presence of innocuous concurrency. In this section we will show that, when instantiated to a programming language where all programs are well-bracketed, our well-bracketed program logic can be used to construct a logical relations model in Iris. Such logical relations models enable reasoning about well-bracketedness, similarly to the more explicit Kripke logical relations of Georges et al. [2021], Dreyer et al. [2010], and Skorstengaard et al. [2019] mentioned earlier.

To demonstrate this point, we took both the unary and binary logical relations models presented in Krebbers et al. [2017] for $\mathsf{F}_{\mu,ref}$,[8] almost verbatim (we removed concurrency), and *only* replaced the weakest preconditions in these logical relations models with well-bracketed weakest preconditions *mutatis mutandis*. For an introductory treatise on logical relations models in Iris see Timany et al. [2022].

### 7.1 Unary Logical Relations

The unary logical relations model of Krebbers et al. [2017] allows one to prove *semantic* type soundness: by showing that a program $e$ is in the logical relation for its type, it holds that $e$ does not crash when run. Type safety is achieved by showing that *all* well-typed programs are indeed in the logical relations that corresponds to their types. As discussed in detail by Timany et al. [2022] this approach allows one to safely combine syntactically well-typed programs with semantically type sound programs, even if the latter are not necessarily syntactically well-typed. Here we consider the following example program, which we explain below:

$$\mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \lambda f.\, r \leftarrow 0; f\,(); r \leftarrow 1; f\,(); \mathsf{if}\, !r \neq 1\, \mathsf{then}\,()\,()\, \mathsf{else}\, 1 \qquad \text{(VAE-check)}$$

This program is essentially the same as VAE except it performs a check before returning the final value. If the return value is *not* 1 then it crashes (applies the unit value to itself), otherwise it simply returns 1. The program VAE-check is not syntactically well-typed as it is treating the unit value $()$ as a function. Nonetheless, it is semantically type sound of the expected type: $(\mathbf{1} \to \mathbf{1}) \to \mathbf{Z}$. Formally, we write this as follows:

$$\emptyset \mid \emptyset \vDash \text{VAE-check} : (\mathbf{1} \to \mathbf{1}) \to \mathbf{Z} \qquad \text{(semantic typing of VAE-check)}$$

Here, the relation $\Xi \mid \Gamma \vDash e : \tau$ is the semantic typing relation, distinguished from the syntactic typing relation by the use of $\vDash$ instead of $\vdash$. The $\Xi$ is the context of type variables (ranged over by $\alpha, \beta, \ldots$), while $\Gamma$ is the context of term variables (ranged over by $x, y, \ldots$); both are empty in this case as VAE-check is a closed program. The complete definition of our unary logical relations model for $\mathsf{F}_{\mu,ref}$ is given in Figure 5. At the heart of this model are the so-called expression interpretation

[8] Sequential System F featuring both universal and existential types, recursive types, and higher-order references.

$$
\begin{aligned}
[\![\tau]\!]^{\mathrm{e}}_{\delta} &\triangleq \lambda e.\, \mathsf{wbwp}\ e\ \left(\!\left[\!\left[\tau\right]\!\right]_{\delta}\right)\\
[\![\alpha]\!]_{\delta} &\triangleq \delta(\alpha)\\
[\![\mathbf{1}]\!]_{\delta} &\triangleq \lambda v.\, v = ()\\
[\![\mathbf{B}]\!]_{\delta} &\triangleq \lambda v.\, v \in \{\mathsf{true}, \mathsf{false}\}\\
[\![\mathbf{Z}]\!]_{\delta} &\triangleq \lambda v.\, v \in \mathbb{Z}\\
[\![\tau_1 \times \tau_2]\!]_{\delta} &\triangleq \lambda v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * [\![\tau_1]\!]_{\delta}(v_1) * [\![\tau_2]\!]_{\delta}(v_2)\\
[\![\tau_1 + \tau_2]\!]_{\delta} &\triangleq \lambda v.\, \bigvee_{i\in\{1,2\}} \exists w.\, (v = \mathsf{inj}_i\ w) * [\![\tau_i]\!]_{\delta}(w)\\
[\![\tau \to \rho]\!]_{\delta} &\triangleq \lambda v.\, \Box\left(\forall w.\, [\![\tau]\!]_{\delta}(w) \mathrel{-\!\!*} [\![\rho]\!]^{\mathrm{e}}_{\delta}(v\ w)\right)\\
[\![\forall \alpha.\, \tau]\!]_{\delta} &\triangleq \lambda v.\, \Box\left(\forall (\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(v\langle\rangle)\right)\\
[\![\exists \alpha.\, \tau]\!]_{\delta} &\triangleq \lambda v.\, \exists (\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, \exists w.\, (v = \mathsf{pack}\langle w\rangle) * [\![\tau]\!]_{\delta,\alpha\mapsto\Psi}(w)\\
[\![\mu \alpha.\, \tau]\!]_{\delta} &\triangleq \mu(\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, \lambda v.\, \exists w.\, (v = \mathsf{fold}\ w) * \triangleright [\![\tau]\!]_{\delta,\alpha\mapsto\Psi}(w)\\
[\![\mathsf{ref}(\tau)]\!]_{\delta} &\triangleq \lambda v.\, \exists (\ell : \mathit{Loc}).\, (v = \ell) * \boxed{\exists w.\, \ell \mapsto w * [\![\tau]\!]_{\delta}(w)}^{\mathcal{N}_{\mathsf{ty}}.\ell}\\
[\![\emptyset]\!]^{\mathrm{c}}_{\delta}(\epsilon) &\triangleq \mathsf{True}\\
[\![\Gamma, x : \tau]\!]^{\mathrm{c}}_{\delta}(\vec{v}\ w) &\triangleq [\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\vec{v}) * [\![\tau]\!]_{\delta}(w)\\
\Xi \mid \Gamma \vDash e : \tau &\triangleq \Box\left(\forall \delta, \vec{v}.\, \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\delta) \mathrel{-\!\!*} [\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} [\![\tau]\!]^{\mathrm{e}}_{\delta}(e[\vec{v}/\vec{x}])\right)
\end{aligned}
$$

Fig. 5. The expression interpretation $[\![\_]\!]^{\mathrm{e}}$, value interpretation $[\![\_]\!]$, typing context interpretation $[\![\_]\!]^{\mathrm{c}}$, and semantic typing judgment for $\mathsf{F}_{\mu,ref}$.[9]

$[\![\_]\!]^{\mathrm{e}}$ and value interpretation $[\![\_]\!]$.[10] These are, unary relations (predicates) on *closed* expressions and values, respectively. The former has type $\mathit{Expr} \to \mathit{iProp}$, while the latter has type $\mathit{Val} \to \mathit{iProp}_{\Box}$. Here $\mathit{iProp}$ is the universe of Iris propositions and $\mathit{iProp}_{\Box}$ is the universe of *persistent* Iris propositions. The value interpretation should be persistent while expressions need not be. This is because in our CBV semantics expressions are evaluated only once. The resulting values, however, can be freely duplicated and reused multiple times. The expression and value interpretations are defined in a mutually recursive fashion by induction on types.[11] As types feature type-level variables (used in universal, existential, and recursive types) all interpretation relations are parameterized by a mapping $\delta$ mapping free type variables to their value interpretations, *i.e.*, predicates of type $\mathit{Val} \to \mathit{iProp}_{\Box}$. The expression interpretation is simply defined in terms of well-bracketed weakest preconditions, capturing that a closed expression is in the interpretation for a type if it is safe and it produces a value in the value interpretation of that type. The definition of the expression relation is the only place where our logical relations model deviates from that in Timany et al. [2022]. We use well-bracketed weakest preconditions as opposed to the ordinary Iris weakest preconditions used by Timany et al. [2022], which enables reasoning about well-bracketedness. The value interpretation describes when a value is, or rather behaves as, a value of a type. For the base types, **1**, **B**, and

[10] Note that the superscript $e$ in the $[\![\_]\!]^{\mathrm{e}}$ relation is part of the notation, not an argument.

[11] In practice (in the Coq development) this conceptual mutual dependency is broken by defining the expression interpretation first as a general predicate transformer of the type $(\mathit{Val} \to \mathit{iProp}_{\Box}) \to (\mathit{Expr} \to \mathit{iProp})$, which is essentially just the (well-bracketed) weakest precondition in our case. The value interpretation is then defined based on this general predicate transformer.

$\mathbf{Z}$, it simply requires the value to be an appropriate constant. Type variables are interpreted by looking up in the map $\delta$. Products and sums are interpreted to require their values to have the appropriate form with underlying terms again in the value interpretation for the corresponding type. A value in the interpretation of a function type, when applied to a value in the interpretation of the domain type, must produce an expression in the codomain type. Universal and existential types are interpreted using the quantifiers of the logic, *i.e.*, by quantifying over arbitrary value interpretation predicates. We write $v\langle\rangle$ for specialization of a polymorphic value to a type, which we omit in the program expression. Note that $v\langle\rangle$ is an expression and not a value as polymorphic terms are suspended, and are only run after specialization. The value interpretation of recursive types is defined using Iris' guarded fixed points. The guarded fixed point $\mu r.\, P$ is always defined, and is a fixed point, *i.e.*, $\mu r.\ P \dashv\vdash P[\mu r.\ P/r]$, whenever the variable $r$ appears only guarded in $P$, *i.e.*, under a later modality, $\triangleright$. Intuitively, the interpretation of a recursive type $\mu\alpha.\,\tau$ says that a value must be a *folded* value where the underlying *unfolded* value is a value of type $\tau$, where $\alpha$ is itself interpreted as the recursive type $\mu\alpha.\,\tau$. Finally, the value interpretation of a reference type $\mathsf{ref}(\tau)$ requires that values be memory locations always storing a value in the interpretation of $\tau$. Here, we consider locations to be of a syntactically distinct class from the rest of the values with an implicit injection to them. This is why we write $\exists(\ell : Loc).\ (v = \ell) * \ldots$ to express that the value is some location $\ell$. The "always storing a value in the interpretation of $\tau$" part is captured by an invariant asserting ownership over the location, and enforcing the desired property. Note that the invariant name is derived from the location itself assigning distinct names to distinct locations.

The expression and value interpretations, which are predicates on closed terms, are lifted to arbitrary open programs in the semantic typing judgment $\Xi \mid \Gamma \vDash e : \tau$ in the standard way for logical relations models, *i.e.*, by closing open terms with semantically well-typed values $\vec{v}$ of the appropriate type, and by universally quantifying over all possible semantic type-level environments $\delta$ for the free type variables. For this purpose we define the typing context interpretation $[\![\Gamma]\!]^{\mathsf{c}}_{\delta}(\vec{v})$ so as to require that each value is in the value interpretation for its corresponding type in $\Gamma$.

Now, given the definition of the unary logical relations we return to establishing semantic typing of VAE-check. Since VAE-check is a closed program, semantic typing of VAE-check above is equivalent to $[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{Z}]\!]^{\mathsf{e}}_{\emptyset}(\text{VAE-check})$. This can in turn, after unfolding the expression and value interpretations, be simplified into a form which very closely resembles VAE-spec of Section 3.2 — note that $[\![\mathbf{1}]\!]_{\emptyset}$ only holds for a singleton consisting of the unit value. That is, we get the following, after the aforementioned simplifications:

$$\mathsf{wbwp}\ \text{VAE-check}\ \Big(\!\Big|g.\ \Box\,(\forall f.\ \Box\,(\forall w.\ w = ()\ -\!\!*\ \mathsf{wbwp}\ f\ w\ (\!|y.\ y = ()|\!))\ -\!\!*\ \mathsf{wbwp}\ g\ f\ (\!|y.\ y \in \mathbb{Z}|\!))\Big|\!\Big)$$

Finally, we can get rid of the quantified $w$ inside, and substitute it with $()$, and fold the well-bracketed weakest preconditions into equivalent well-bracketed Hoare triples:

$$\mathsf{wbwp}\ \text{VAE-check}\ \Big(\!\Big|g.\ \big(\!|\forall f.\ (\!|\mathsf{True}|\!)\ f\ ()\ (\!|y.\ y = ()|\!)|\!|\big)\ g\ f\ (\!|y.\ y \in \mathbb{Z}|\!)\Big|\!\Big)$$

As a result, we can employ the same line of reasoning as in Section 3.2 to prove the semantic typing of VAE-check! We emphasize that the earlier logical relations model of Krebbers et al. [2017] does not support reasoning about well-bracketedness, and thus cannot be used to show semantic typing of VAE-check.

### 7.2 Binary Logical Relations

In addition to the unary logical relations model above, our work also includes a binary logical relations model which can be used to establish contextual refinements and equivalences. Again, this logical relations model is very similar to that in Timany et al. [2022] with the minor difference

of replacing weakest preconditions with well-bracketed weakest preconditions, and removing concurrency. Recall that for a pair of programs $e$ and $e'$ such that $\Xi \mid \Gamma \vdash e : \tau$ and $\Xi \mid \Gamma \vdash e' : \tau$, we say that $e$ contextually refines a program $e'$, written $\Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau$, if in any context (any program of ground type with a hole), we can replace $e'$ with $e$ without changing the overall behavior of the program. In a refinement $\Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau$, we often refer to $e$ as the *implementation* and $e'$ is the *specification*. We say that $e$ is contextually equivalent to $e'$, written $\Xi \mid \Gamma \vdash e \approx_{\text{ctx}} e' : \tau$, if both $\Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau$ and $\Xi \mid \Gamma \vdash e' \leq_{\text{ctx}} e : \tau$ hold. The binary logical relations model is set up so that whenever $e$ *logically refines* $e'$, *i.e.*, when the pair $(e, e')$ is in the logical relation, written $\Xi \mid \Gamma \vdash e \leq_{\log} e' : \tau$, then we can conclude $\Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau$.

We use our binary logical relations model to prove that VAE is contextually equivalent to the program $\lambda f.\ f\ ();f\ ();1$ at type $(\mathbf{1} \to \mathbf{1}) \to \mathbf{Z}$. The latter simply calls $f$ twice before returning 1. We prove the equivalence by showing the two refinements. For both directions, the essence of the reasoning is the same as that of the proof of VAE in Section 3.2. We start by allocating the reference for VAE, regardless of which side VAE is on, obtain a stack using wbHoare-create-stack, and proceed to establish the exact invariant as in VAE-inv. We then follow a similar proof strategy to show that the VAE program returns 1 regardless of the side of refinement it appears on. For the detailed proof, see our accompanying Coq formalization.

## 8 DISCUSSION: ENCODING STATE TRANSITION SYSTEMS

In this paper we have proposed ghost stacks as a logical mechanism for reasoning about well-bracketedness. We demonstrated that ghost stacks can be used to reason logically about examples that prior work reasoned about using relational models indexed over state transition systems with private and public transitions. One may naturally ask whether ghost stacks are indeed stronger than state transition systems. That is, can we reason about all programs that one could using state transition systems? The answer is: yes, we can. We prove this by showing that we can use ghost stacks to encode a given state transition system with private and public transitions into program logic predicates that essentially track the state of the state transition system. These predicates are defined using the monotone resource algebra [Timany and Birkedal 2021] to construct a resource that can only be updated in accordance with the public transition relation. The private transitions are modeled using ghost stacks. For brevity, we refer the reader to our accompanying technical appendix and Coq formalization for further details of this encoding, and a proof of VAE which uses the encoding of the state transition system VAE-sts instead of the more direct proof using ghost stacks presented earlier.

One interesting observation regarding the proof of VAE is that we only ever use the top element of the ghost stack. Indeed, the proof could also have been carried out if instead of pushing on the stack we had kept the stack a singleton. That is, instead of pushing, we could have swapped the top (only) element of the stack with the new ghost name when making a "private transition". However, in the general encoding of state transition systems (see the Coq proof for details) we need to refer to the stack elements below the top elements in order to assert that "private transitions" do take place according to the private transition relation of the state transition system we work with.

## 9 THE GHOST THEORY OF STACK COLLECTIONS

To define the predicates AllStacksExcept, $stack_\bullet$, $stack_\circ$, and $stack_\exists$ we introduce the following new propositions which are themselves directly defined in terms of ghost resources: $stack_\bullet^{\mathsf{IN}}$, $stack_\circ^{\mathsf{IN}}$, *EntireDom*, and *SubsetOfDom*. The predicates AllStacksExcept, $stack_\bullet$, $stack_\circ$, and $stack_\exists$ are defined

as follows:

$$\begin{aligned}
\mathit{stack}_\bullet(N,s) &\triangleq \mathit{stack}_\bullet^{\mathsf{IN}}(N,s) * \mathit{SubsetOfDom}(\{N\}) \\
\mathit{stack}_\circ(N,s) &\triangleq \mathit{stack}_\circ^{\mathsf{IN}}(N,s) * \mathit{SubsetOfDom}(\{N\}) \\
\mathit{stack}_\exists(N) &\triangleq \mathit{SubsetOfDom}(\{N\}) \\
\mathsf{AllStacksExcept}(\mathcal{S},O) &\triangleq O \subseteq \mathrm{dom}(\mathcal{S}) * \mathit{EntireDom}(\mathrm{dom}(\mathcal{S})) * \mathop{\Large *}_{N\in\mathrm{dom}(\mathcal{S})\setminus O} \mathit{stack}_\bullet(N,\mathcal{S}(N))
\end{aligned}$$

The *SubsetOfDom* predicate is the only predicate among these predicates that is persistent. The predicates *SubsetOfDom* and *EntireDom* together track, in terms of resources, the *finite* domain of the map of all stacks. These predicates, as their names suggest, are defined so that the set tracked by *a SubsetOfDom* predicate is a subset of *the* set tracked by *EntireDom*:

$$\mathit{SubsetOfDom}(A) * \mathit{EntireDom}(B) \vdash A \subseteq B \qquad \text{(dom-subset)}$$

$$\mathit{SubsetOfDom}(A) * \mathit{SubsetOfDom}(B) \dashv\vdash \mathit{SubsetOfDom}(A \cup B) \qquad \text{(dom-distributes)}$$

$$\mathit{EntireDom}(B) * \mathsf{finite}(A) \vdash \Rrightarrow \mathit{EntireDom}(A \cup B) * \mathit{SubsetOfDom}(A) \qquad \text{(dom-grow)}$$

The predicates $\mathit{stack}_\bullet$ and $\mathit{stack}_\circ$ both feature $\mathit{SubsetOfDom}(\{N\})$ to express that the name $N$ is indeed in the domain of all tracked stacks. The other parts of these predicates, $\mathit{stack}_\bullet^{\mathsf{IN}}$ and $\mathit{stack}_\circ^{\mathsf{IN}}$ are defined to satisfy the following basic rules:

$$\mathit{stack}_\bullet^{\mathsf{IN}}(N,s) * \mathit{stack}_\circ^{\mathsf{IN}}(N,s') \vdash s = s' \qquad (\mathsf{stacks}^{\mathsf{IN}}\text{-agree})$$

$$\mathit{stack}_\bullet^{\mathsf{IN}}(N,s) * \mathit{stack}_\bullet^{\mathsf{IN}}(N,s') \vdash \mathsf{False} \qquad (\mathsf{stacks}_\bullet{}^{\mathsf{IN}}\text{-unique})$$

$$\mathit{stack}_\circ^{\mathsf{IN}}(N,s) * \mathit{stack}_\circ^{\mathsf{IN}}(N,s') \vdash \mathsf{False} \qquad (\mathsf{stacks}_\circ{}^{\mathsf{IN}}\text{-unique})$$

$$\mathit{stack}_\bullet^{\mathsf{IN}}(N,s) * \mathit{stack}_\circ^{\mathsf{IN}}(N,s') \vdash \Rrightarrow \mathit{stack}_\bullet^{\mathsf{IN}}(N,s'') * \mathit{stack}_\circ^{\mathsf{IN}}(N,s'') \qquad (\mathsf{stacks}^{\mathsf{IN}}\text{-update})$$

$$\mathsf{infinite}(A) \vdash \Rrightarrow \exists N \in A.\ \mathit{stack}_\bullet^{\mathsf{IN}}(N,s) * \mathit{stack}_\circ^{\mathsf{IN}}(N,s) \qquad (\mathsf{stacks}^{\mathsf{IN}}\text{-create})$$

Note how these rules specify that the stacks always agree ($\mathsf{stacks}^{\mathsf{IN}}$-agree), that they are initially created in a way that they agree ($\mathsf{stacks}^{\mathsf{IN}}$-create), and that they may only be updated as long as the agreement is maintained ($\mathsf{stacks}^{\mathsf{IN}}$-update). The reason why it is necessary to track the domain of the map of stacks in a separate resource is because we need to prove that whenever we have $\mathit{stack}_\bullet$ or $\mathit{stack}_\circ$, for some stack $N$, then the stack $N$ must indeed be in the map tracked by AllStacksExcept. Otherwise, the latter would only be a collection of $\mathit{stack}_\bullet$ predicates for which we only know that they are not in the stack mask.

It is easy to see how the rules above can derive the rules governing $\mathit{stack}_\bullet$ and $\mathit{stack}_\circ$: STACKS-AGREE, STACK-SYSTEM-UNIQUE, STACK-USER-UNIQUE, STACK-EXISTS, STACKS-PUSH, and STACKS-POP — the stack-like behavior of the rules STACKS-PUSH and STACKS-POP is not enforced at the level of resources but only by the "interface" of the rules for updating $\mathit{stack}_\bullet$ and $\mathit{stack}_\circ$.

It only remains for us to justify that the definitions of AllStacksExcept, $\mathit{stack}_\bullet$, $\mathit{stack}_\circ$, and $\mathit{stack}_\exists$ do indeed satisfy the rules presented in Figure 4. The rule MASK-SUBSET-DOM follows by definition. For the rule STACK-EXISTS-IN we essentially only need to apply dom-subset. As for the rule STACK-USER-NOT-OUT, we will first apply the rule STACK-EXISTS together with MASK-SUBSET-DOM to obtain that $N \in \mathrm{dom}(\mathcal{S})$. Given that $N \notin O$, the proposition $\mathit{stack}_\bullet(N,\mathcal{S}(N))$ must be part of the big separating conjunction in the definition of the AllStacksExcept proposition. This allows us to conclude the proof by applying the rule STACKS-AGREE. For the rule STACK-SYSTEM-IS-OUT, assume the contrary, *i.e.*, that $N \notin O$. Since $\mathit{SubsetOfDom}(\{N\})$ is included in the definition of $\mathit{stack}_\bullet(N,s)$, we have $N \in \mathrm{dom}(\mathcal{S})$. This means that we can obtain $\mathit{stack}_\bullet(N,\mathcal{S}(N))$ from the big separating conjunction

in the definition of the AllStacksExcept proposition. This is a contradiction as per STACK-SYSTEM-UNIQUE. The rules STACK-TAKE-OUT and STACK-PUT-BACK, respectively, simply take the system part of the stack, $stack_\bullet$, out of the big separating conjunction in the definition of AllStacksExcept, and put it back in there. Note that there is nothing in the definition of $\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O})$ that pins down the contents of a stack that is in the mask $\mathcal{O}$. Hence, the contents of such a stack can be changed arbitrarily in the $\mathcal{S}$ part of $\mathsf{AllStacksExcept}(\mathcal{S}, \mathcal{O})$. This is exactly what the rule CHANGE-OUT-STACK states. In order to validate the rule CREATE-STACK we allocate a new stack $N$ using the rule $\mathsf{stacks}^{\mathsf{IN}}$-create. We then use the rule dom-grow to update the *EntireDom* and obtain $\mathit{SubsetOfDom}(\{N\})$. This allows us to construct $stack_\bullet$ and $stack_\circ$ propositions as necessary. However, we need to guarantee that the name $N$ is indeed fresh, in the sense that it is not already an element of $\mathrm{dom}(\mathcal{S})$. This is why in the rule $\mathsf{stacks}^{\mathsf{IN}}$-create we have the possibility to pick a set $A$ of stack names to draw the stack name from. The only side condition is that the set $A$ must be an infinite set. The idea is that the infinitude of $A$ guarantees existence of a fresh stack name that appears in $A$. Here we take $A$ to be the set of all stack names except for those in $\mathrm{dom}(\mathcal{S})$ which is by definition a finite set.

## 10 RELATED AND FUTURE WORK

We have already discussed the most closely related work in the Introduction; in this section we discuss other related work.

***Reasoning About Well-Bracketedness via Contextual Equivalences.*** To the best of our knowledge, prior work has only reasoned about correctness of programs that depend on well-bracketedness by using models that can prove contextual program equivalences. For instance, Dreyer et al. [2010] used a binary logical relations model to prove that VAE is (contextually) equivalent to a program that simply calls its higher-order argument twice and returns 1. In contrast, we present a program logic for reasoning about well-bracketedness. One advantage of our program logic approach is that it can *both* be used to reason about partial correctness of programs and to build logical relation models for reasoning about semantic type soundness and contextual equivalence, as shown in Section 7.

Apart from logical relations models, which we have already discussed, other semantic approaches have also been used to study well-bracketedness. In particular, well-bracketedness has been captured semantically in game semantics models, which have been developed for a variety of programming languages, *e.g.*, [Abramsky et al. 1998; Laird 1997; Murawski 2005; Murawski and Tzevelekos 2011]. Game semantics has been used to obtain fully-abstract denotational semantics which can thus in principle be used to prove program equivalences by showing that the game-based denotations of programs (called strategies) are equivalent. However, it is not trivial to do so for concrete challenging examples, such as the Awkward and Very Awkward examples.

Inspired by game semantics models, so-called operational game semantics techniques [Lassen and Levy 2007, 2008; Støvring and Lassen 2009] develop an approach called "normal form bisimulation" which are coinductive relations which, in essence, can be used to show equivalence of programs' Böhm trees.

Environmental bisimulation techniques [Biernacki et al. 2019; Hur et al. 2012; Sumii 2009; Sumii and Pierce 2004] define relational models on expressions using coinduction. These techniques can prove the contextual equivalence for the VAE example (similar to our logical relation in Section 7.2). Notable among these is the work of Hur et al. [2012] where the authors, inspired by Dreyer et al. [2010], incorporate state transition systems with public/private transitions to improve support for reasoning about evolution of local state. Biernacki et al. [2019] present a bisimulation relation over untyped terms. In order to ensure well-bracketedness Biernacki et al. [2019], following Jagadeesan et al. [2009]; Laird [2007], uses a stack of evaluation contexts. Similarly to Dreyer et al. [2010] and

what we did in Section 7.2, Biernacki et al. [2019] use their bisimulation relation to show that VAE is equivalent to a program that calls its argument twice and returns 1.

Recently, Jaber and Murawski [2021] proposed so-called Kripke normal-form bisimulations, which combine ideas from normal-form bisimulation and Kripke logical relations with game semantics, and which also accounts for well-bracketing and can be used to prove contextual equivalence for the VAE example.

The closest work to ours is a short unpublished note by Pottier [2009]. Pottier [2009] generalizes the higher-order frame rule and the anti-frame rule of his earlier work [Pottier 2008] to enable reasoning about well-bracketedness. The system that Pottier [2009] works in is not a program logic but rather a *type-and-capability* system. A capability $I$ in this system, according to Pottier [2009]: "at the same time asserts ownership and describes the type structure of a piece of state. (One could also think of $I$ as a separation logic assertion.)" Intuitively, capabilities indicate the (possible) side-effects of a computation. The higher-order frame rule of Pottier [2009] allows conjoining capabilities to the type of computations that do not require those capabilities — essentially weakening a computation without a certain side-effect to one where (the capability for) the side-effect is included in the type. The anti-frame rule, being the dual of the higher-order frame rule, allows hiding a *local* side-effect — a capability can be removed from the type of a computation if that capability regards resources that are *allocated and initialized* by the code in question. The generalization presented by Pottier [2009] parameterizes capabilities with an arbitrary kind and a relation on the elements of that kind. (Pottier [2009] uses the term kind instead of what would normally be called a type, *e.g.* the kind $\mathbb{Z}$ of integers.) The generalizations of the higher-order frame rule and the anti-frame rule enforce that programs only change state in a way that obeys the given relation. Intuitively, in effect, the parameter of the capability and the relation on it are, respectively, analogous to the states of the state-transition-system and its public transition relation. Pottier [2009] present a proof, in the form of a typing derivation, of VAE with an assertion before the end of the closure that asserts that the return value is always 1. The capability picked here asserts ownership over the location allocated by VAE. It furthermore requires that the value of the location is in the set $\{0, 1\}$, and that this value is exactly the same as the integer parameter of the capability. The relation on the parameter is simply taken to be the $\leq$ relation on integers. The higher-order frame rule and the anti-frame rule of Pottier [2009], at a very high intuitive level, respectively play the roles of our rules wbHoare-mend-stack and wbHoare-access-stack, the major difference being that our program logic decouples ownership (capabilities in Pottier [2009]) from the well-bracketedness mechanism, *i.e.*, stacks. In contrast to our formalized proofs, Pottier [2009] does not prove but only conjectures the soundness of his proposed generalizations of the higher-order frame and anti-frame rules.

***Innocuous Control Effects.*** In Section 5 we discussed how our program logic can be used to reason about well-bracketedness of programs that make use of concurrency in an innocuous way. This raises the question of whether our program logic can reason about well-bracketed programs in the presence of innocuous control effects such as call/cc or algebraic effects and handlers. We conjecture that it is the case if there are reasoning principles in place that can be used as the basis for our logic. For instance, for call/cc, the context-local weakest preconditions of Timany and Birkedal [2019] ensure context-locality, *i.e.*, that all the uses of call/cc are not observable from outside. Building our well-bracketed program logic on top of context-local weakest preconditions should result in a system which can reason about innocuous uses of call/cc. As for algebraic effects and handlers with delimited control effects, Hoare-logic-based reasoning principles for programs in such systems usually come equipped with clear markers which demarcate the limits of the control effects within the program, *e.g.*, the protocols in de Vilhena and Pottier [2021]. We conjecture

that such logics can serve as a basis for a well-bracketed program logic that can reason about well-bracketedness in the presence of innocuous uses of control effects.

## 11 CONCLUSION

We have presented a versatile well-bracketed program logic which captures the essence of well-bracketedness. We showed how our program logic can be used directly to reason about tricky examples, whose correctness relies on well-bracketedness. In particular, we used the well-bracketed program logic to prove correctness of implementation of the very awkward example (VAE) in HeapLang, a concurrent language which does not enforce well-bracketedness. We used this fact to demonstrate that even in the presence of innocuous concurrency, we can still employ our program logic to establish well-bracketedness of the calls to the VAE closure, even if the entire execution trace is not necessarily well-bracketed. Furthermore, we employed the technique of Birkedal et al. [2021] to show that well-bracketed Hoare triples imply that the traces of calls and returns produced by the program are in fact well-bracketed. We also used our well-bracketed program logic to construct (unary and binary) logical relations models for a sequential programming language were well-bracketedness is enforced by the operational semantics. The remarkable aspect of these logical relations models is that they are almost entirely standard [Krebbers et al. 2017] except for the use of well-bracketed weakest preconditions in place of ordinary weakest preconditions of Iris. We used both of these logical relations models to show examples that rely on well-bracketedness.

## ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

## REFERENCES

S. Abramsky, K. Honda, and G. McCusker. 1998. A fully abstract game semantics for general references. In *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*. 334–344. https://doi.org/10.1109/LICS.1998.705669

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1480881.1480925

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019. A Complete Normal-Form Bisimilarity for State. In *Foundations of Software Science and Computation Structures*, Mikołaj Bojańczyk and Alex Simpson (Eds.). Springer International Publishing, Cham, 98–114.

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. (2017). http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (aug 2021), 29 pages. https://doi.org/10.1145/3473586

Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proc. ACM Program. Lang.* 5, POPL, Article 33 (jan 2021), 28 pages. https://doi.org/10.1145/3434314

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/1863543.1863566

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities. *Proc. ACM Program. Lang.* 5, POPL, Article 6 (jan 2021), 30 pages. https://doi.org/10.1145/3434287

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/2103656.2103666

J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

Guilhem Jaber and Andrzej S. Murawski. 2021. Compositional relational reasoning via operational game semantics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470524

Radha Jagadeesan, Corin Pitcher, and James Riely. 2009. *Open Bisimulation for Aspects.* Springer Berlin Heidelberg, Berlin, Heidelberg, 72–132. https://doi.org/10.1007/978-3-642-02059-9_3

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 205–217. http://dl.acm.org/citation.cfm?id=3009855

J. Laird. 1997. Full abstraction for functional languages with control. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. 58–67. https://doi.org/10.1109/LICS.1997.614931

James Laird. 2007. A Fully Abstract Trace Semantics for General References. In *Automata, Languages and Programming*, Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 667–679.

Soren B. Lassen and Paul Blain Levy. 2007. Typed Normal Form Bisimulation. In *Computer Science Logic*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–297.

Soren B. Lassen and Paul Blain Levy. 2008. Typed Normal Form Bisimulation for Parametric Polymorphism. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 341–352. https://doi.org/10.1109/LICS.2008.26

Andrzej S. Murawski. 2005. Functions with local state: Regularity and undecidability. *Theoretical Computer Science* 338, 1 (2005), 315–349. https://doi.org/10.1016/j.tcs.2004.12.036

Andrzej S. Murawski and Nikos Tzevelekos. 2011. Game Semantics for Good General References. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 75–84. https://doi.org/10.1109/LICS.2011.31

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310.

A. M. Pitts and I. D. B. Stark. 1999. *Operational Reasoning for Functions with Local State.* Cambridge University Press, USA, 227–274.

Francois Pottier. 2008. Hiding Local State in Direct Style: A Higher-Order Anti-Frame Rule. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 331–340. https://doi.org/10.1109/LICS.2008.16

François Pottier. 2009. Generalizing the higher-order frame and anti-frame rules. (2009). http://cambium.inria.fr/~fpottier/publis/fpottier-gaf-2009.pdf [Unpublished notes available on Pottier's institutional homepage (accessed on Jun 7$^{th}$ 2023)].

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (jan 2019), 28 pages. https://doi.org/10.1145/3290332

Kristian Støvring and Soren B. Lassen. 2009. *A Complete, Co-inductive Syntactic Theory of Sequential Control and State.* Springer Berlin Heidelberg, Berlin, Heidelberg, 329–375. https://doi.org/10.1007/978-3-642-04164-8_17

Eijiro Sumii. 2009. A Complete Characterization of Observational Equivalence in Polymorphic $\lambda$-Calculus with General References. In *Computer Science Logic*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 455–469.

Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/964001.964015

Amin Timany and Lars Birkedal. 2019. Mechanized Relational Verification of Concurrent Programs with Continuations. *Proc. ACM Program. Lang.* 3, ICFP, Article 105 (July 2019), 28 pages. https://doi.org/10.1145/3341709

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 91–104. https://doi.org/10.1145/3437992.3439931

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

Received 2023-07-11; accepted 2023-11-07