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.

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

ACM Reference Format:

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., [Georges et al. 2021; Skorstengaard et al. 2019], 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 programming language, whose correctness depends on well-bracketedness of control flow:

```
let r = ref(0) in λf. r ← 0; f (); r ← 1; f (); !r
```

(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 or concurrency (see Dreyer et al. [2010] for an example where continuations break 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 of Pitts and Stark [1999], called the “awkward” example by Pitts and Stark [1999], because they could not handle it, is the following program:

\[
\text{let } r = \text{ref}(0) \text{ in } \lambda f. r \leftarrow 1; f () ; ! r
\]

(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:
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, this approach has so far only been able to match the reasoning power of 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. Indeed, these works both consider well-bracketedness as a correctness aspect of the calling convention they present, and 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.
- **Versatility** It can be 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.

---

\(^1\)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].
The Essence of the Proof of the Awkward Example in Iris. To reason about non-trivial programs in separation logic, one often uses a notion of ghost resources (aka auxiliary or logical state), which are tied to the physical program state using different mechanisms (so-called invariants in Iris). Ghost resources are strategically picked so as to ensure that they may only evolve in certain ways, e.g., 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, featuring a pair of predicates $\text{pending}(\gamma)$ and $\text{shot}(\gamma) — \gamma$ is the name of the resource instance. The theory only allows the state to start in $\text{pending}(\gamma)$ and evolve to $\text{shot}(\gamma)$ 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 $\text{pending}(\gamma)$ or it is 1 in which case we must have $\text{shot}(\gamma)$. In Iris, this requirement can be enforced using an Iris invariant. In essence, this approach essentially reflects a state transition system, similar to VAE-sts but without the private transition, in the ghost theory. (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 it perfectly adapted to the kind of reasoning supported by public transitions in state transition systems in 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, 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, and allows reasoning about programs in the concurrent higher-order imperative programming language known as HeapLang. 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 that allow the user of the program logic, whenever necessary, to allocate a new ghost stack of ghost names. (This is similar to how the standard Iris program logic allows the user to allocate new invariants when necessary.) The idea is then that, instead of directly tying physical state and ghost state together in an invariant, one ties physical state to ghost state currently tracked at the top of the ghost stack. Then the counterpart of making a private transition from state $s$ to state $s'$ is as follows. One leaves the ghost state associated to state $s$ untouched; instead, one allocates a fresh instance of ghost state associated to

---

2Our one-shot resource is a slight simplification of the one-shot resource of Jung et al. [2016].
state $s'$. One then pushes the name associated with this newly allocated ghost state on top of the stack. Finally, in order to roll back a private transition, one simply has to pop the stack.

As we will demonstrate in Section 3.2 this allows 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. Observe that the point of the well-bracketed program logic is to increase the expressiveness of the logic and allow 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 to use ghost stacks and proofs can be exactly as in the standard Iris program logic. Thus one should think of the 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 construct a unary and a binary logical relations model of the sequential subset of HeapLang using the well-bracketed Hoare logic (instead of the ordinary Hoare logic of Iris) and use both of them to reason about VAE [Versatility]. 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 using VAE, the calls to the VAE closure are indeed well-bracketed.
- 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 and uses this to introduce basic concepts of the Iris program logic. We present our well-bracketed program logic in Section 3 and show in Section 3.2 how it 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 using what we will call a theory of stack collections. We discuss in Section 4.2 how we derive the rules of our well-bracketed program logic. Then in Section 5 we discuss how the well-bracketed program logic can be used to in conjunction with the underlying program logic of Iris to reason about well-bracketedness of programs even in the presence of innocuous concurrency, and 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 discuss how our well-bracketed program logic can be used 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]. In Section 8 we discuss how ghost stacks really are a general mechanism that can 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:

\[
\{ \text{True} \} \quad \begin{align*}
\text{let } & \; r = \text{ref}(0) \text{ in } \\
\lambda f. & \; r \leftarrow 1; f();!r \\
\end{align*} \quad \{ g. \forall f. \{ \{ \text{True} \} f() \{ x. x = () \} \} g f \{ x. x = 1 \} \}
\]

(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 \( \text{pending}(\gamma) \) and \( \text{shot}(\gamma) \), where \( \text{pending}(\gamma) \) may be turned into \( \text{shot}(\gamma) \) 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 \text{MAKE-ONE-SHOT}:

\[
\text{MAKE-ONE-SHOT} \quad \vdash \exists \gamma. \text{pending}(\gamma)
\]

This rule states that we can always update ghost resources (\( \exists \) 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 \( \text{pending}(\gamma) \) resource to establish the following invariant:

\[
\text{AwkwardInv} \triangleq (\ell \mapsto 0 \ast \text{pending}(\gamma)) \lor (\ell \mapsto 1 \ast \text{shot}(\gamma))
\]

(awkward-invariant)

In Iris we write \( [P]^N \) for the persistent (and hence duplicable) proposition that asserts that \( P \) holds invariantly. The \( N \) in \( [P]^N \) is the name of the invariant and is used for bookkeeping purposes. Looking inside our invariant, the proposition \( \ell \mapsto v \) asserts exclusive ownership over memory location \( \ell \) and also states that this location stores the value \( v \). The connective \( \ast \) is called the separating conjunction. It incorporates a notion of separation or disjointness; \( P \ast Q \) 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 \text{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.\textsuperscript{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.

\[
\text{INV-OPEN} \\
\{I \ast P\} e \{x. I \ast Q\} \quad e \text{ is atomic} \\
\{\text{dyn} \ast P\} e \{x. Q\}
\]

In other words, 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. In particular, our one-shot ghost theory enjoys the following rules:

- **Shoot**
  
  \[
  \text{pending}(y) \vdash \text{shot}(y) \quad \text{pending}(y) \ast \text{shot}(y) \vdash \text{False} \quad \vdash \text{persistent}(\text{shot}(y))
  \]

  Using the **Shoot** rule, any pending resource can be shot by updating ghost resources. Note that \text{shot}(y) is persistent (duplicable) as it merely asserts the knowledge that the resource is not pending anymore, while \text{pending}(y) asserts ownership of the resource and the fact that it is not shot yet. This is to ensure that each instance of the one-shot resource can only be shot once; hence the name. In other words, we have the following for \text{shot}(y) but not the analogous for \text{pending}(y):

  \[
  \text{shot}(y) \vdash \text{shot}(y) \ast \text{shot}(y).
  \]

  Here, \(|\vdash\) 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. Note here 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.)

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

\[
\{\text{AwkwardInv} \ast \{\text{True}\} f () \{x. x = ()\}\} \ell \leftarrow 1; f (); ! \ell \{x. x = 1\}
\]

(1)

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

\[
\{(\ell \leftarrow 0 \ast \text{pending}(y)) \lor (\ell \leftarrow 1 \ast \text{shot}(y))\} \ell \leftarrow 1 \{x. x = () \ast \ell \leftarrow 1 \ast \text{shot}(y)\}
\]

(2)

Note how the postcondition above allows us to reestablish the invariant, and at the same time, since \text{shot}(y) is persistent (and thus duplicable), also retain a copy of \text{shot}(y). 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 \(\ell \leftarrow 1\) in the postcondition as the storing operation has mutated the heap of the program. To reconcile the postcondition we are then only required to produce \text{shot}(y) but in this case we have \text{pending}(y) from the precondition which allows us to use the rule **Shoot**. To finalize the proof, it then suffices to establish

\[
\{\text{AwkwardInv} \ast \{\text{True}\} f () \{x. x = ()\} \ast \text{shot}(y)\} f (); ! \ell \{x. x = 1\}
\]

(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

\textsuperscript{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.
shot(γ) 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 ℓ.

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:

$$\langle P \rangle e \langle \Psi(v) \rangle^O$$

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. 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 stating that once a program is proven correct with respect to a
while the latter is handed to the user and therefore represents the "user’s view" of the stack. The well-bracketed program logic:

\[
\text{wbHoare-wbHoare view} \text{ (read: program logic view) of the stack which is tracked by well-bracketed Hoare triples,}
\]

accessing stack user. This is why the fresh name \( \mathcal{N} \) the end, along \( \mathcal{R} \) and can use them (and any other resources and invariants) in order to establish antecedents of the rule the name \( \mathcal{N} \) fresh stack name \( \mathcal{R} \) points of interest here; note how the user of the rule has to pick a predicate Hoare triple one may create a new stack using the rule in Section 5.

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 logic with reasoning in the well-bracketed program logic, and 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, and 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 then, at any point in the proof of a well-bracketed Hoare triple one may create a new stack using the rule \( \text{wbHoare-create-stack} \). There are a few points of interest here; note how the user of the rule has to pick a predicate \( R \) parameterized by the fresh stack name \( \mathcal{N} \) that the program logic picks when creating the stack — this is why in both antecedents of the rule the name \( \mathcal{N} \) is universally quantified. Also, the user obtains \( \mathcal{N} \) and \( \mathcal{R} \) and can use them (and any other resources and invariants) in order to establish \( R \). Furthermore, in the end, along \( \mathcal{R} \), the user of the rule must return the \( \mathcal{R} \) for some initial stack chosen by the user. This is why the fresh name \( \mathcal{N} \) is not added to the stack mask unlike the following rule for accessing \( \mathcal{R} \).

\[
\begin{align*}
\text{wbHoare-create-stack} & \quad \forall \mathcal{N}. \text{stack}_\mathcal{N}(\mathcal{N}, []) \star \text{stack}_\mathcal{N}(\mathcal{N}, []) \not\in \mathcal{R}(\mathcal{N}) \star \exists \mathcal{N}. \text{stack}_\mathcal{N}(\mathcal{N}, \mathcal{N}) & \forall \mathcal{N}. \langle \mathcal{P} \star \mathcal{R}(\mathcal{N}) \rangle \in \langle \mathcal{P} \star \mathcal{O} \rangle \\
\text{wbHoare-access-stack} & \quad \mathcal{N} \not\in \mathcal{O} & \left( \forall \mathcal{N}. \langle \mathcal{P} \star \text{stack}_\mathcal{N}(\mathcal{N}, s) \rangle \in \langle \mathcal{N}. \Phi(x) \star \text{stack}_\mathcal{N}(\mathcal{N}, s) \rangle^{\mathcal{O} \cup \{\mathcal{N}\}} \right) + \langle \text{stack}_\mathcal{N}(\mathcal{N}, \mathcal{P}) \rangle \in \langle \mathcal{P} \rangle \rangle \mathcal{O} \\
\text{wbHoare-mend-stack} & \quad \langle \mathcal{P} \rangle \in \langle \mathcal{P} \rangle^{\mathcal{O} \cup \{\mathcal{N}\}} & \langle \text{stack}_\mathcal{N}(\mathcal{N}, s) \rangle \in \langle \mathcal{N}. \Phi(x) \star \text{stack}_\mathcal{N}(\mathcal{N}, s) \rangle^{\mathcal{O}} & \{\mathcal{P} \} \in \{\mathcal{P} \} \in \langle \mathcal{P} \rangle \rangle \mathcal{O} \\
\text{Hoare-wbHoare} & \quad \{\mathcal{P} \} \in \{\mathcal{P} \} \in \langle \mathcal{P} \rangle \rangle \mathcal{O} \\
\end{align*}
\]

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

Certain pre- and postcondition it remains correct in the presence of other resources, be it physical or ghost resources — the separating conjunction is key here as owning more resources disjoint from \( \mathcal{P} \) can never invalidate \( \mathcal{P} \). The rule \( \text{wbHoare-bind} \) expresses that to prove a program consisting of a program \( \mathcal{P} \) under an evaluation context \( \mathcal{K} \) (the part of the program that would run following evaluation of \( \mathcal{P} \)), it suffices to show that \( \mathcal{K} \) is correct when plugging in to \( \mathcal{K} \) any value that satisfies the postcondition of \( \mathcal{P} \).
Rule \texttt{wbHoare-access-stack} allows to access the \texttt{stack}\textsubscript{*} part of an existing stack. The crucial point here is that the contents of the stack is expected to be returned untouched in the postcondition. This is exactly 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 \texttt{VAE}.) The rule \texttt{wbHoare-access-stack} requires \texttt{stack}\textsubscript{∃} as an evidence that a stack with name \(N\) has indeed been created before. Importantly, we must show that the \texttt{stack}\textsubscript{*} part of the stack \(N\) is not already in the stack mask \(O\) of stacks that are already accessed.

The final rule for using stacks of ghost names is rule \texttt{wbHoare-mend-stack} and is the exact dual of \texttt{wbHoare-access-stack}. The rule relies on the fact that well-bracketed Hoare triples always respect all stacks. It allows us to put the stack back temporarily, so as to give access to it to the rest of the program, but at the same time guarantees that the stack will come back to us unchanged — see the discussion below regarding the proof of \texttt{VAE}. Note also that since we have \texttt{stack}\textsubscript{*} \((N, s)\) in the precondition of the Hoare triple in the consequent of the rule, we know that the name \(N\) must be in the stack mask \(O\) 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 \texttt{stack}\textsubscript{*} \((N, s)\) and \texttt{stack}\textsubscript{o} \((N, s)\) express exclusive ownership over respective halves of the ghost stack named \(N\). They consequently agree on the stack contents (\texttt{stacks-agree}) and are unique (\texttt{stack-system-uniqe} and \texttt{stack-user-uniqe}). Given \texttt{stack}\textsubscript{o} \((N, s)\) one may derive a persistent witness that the stack \(N\) exists (\texttt{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 (\texttt{stacks-push}) or popping the top of the stack (\texttt{stacks-pop}).

#### 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 with a meta-level precondition \(\varphi\) ([\(\cdot\)] embeds meta-level propositions, \(i.e.,\) Prop in Coq into Iris’s \(iProp\)) is safe, \(i.e.,\) it does not crash, and that whenever it terminates to a value \(v\) the postcondition \(\varphi(v)\) holds. Formally, this is stated as the following theorem.

\textbf{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 \( \text{Val} \rightarrow \text{Prop} \). The following holds:

\[
\vdash \triangleleft \text{SI}(\sigma) \equiv \{ \text{True} \} \ e \ (x. \ [\varphi(x)]) \implies \text{Safe}_\varphi(\sigma, e)
\]

where \( \text{SI} \) 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.

We emphasize that the statement of the theorem 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 theorem is requiring the user of the theorem to provide resources for the initial state of the program (rather trivial for the empty heap starting state). 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 — 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; proofs should always allocate their required stacks using the rule \( \text{wbHoare-create-stack} \).

Both the statement of the adequacy theorem (the implication above) and the predicate \( \text{Safe}_\varphi \) 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 and the proof of well-bracketed Hoare triples that take place in Iris — this is syntactically indicated by the Iris turnstile being delimited by the surrounding parentheses. That means, that to trust that the program is indeed safe one only needs to trust the soundness of the meta theory (Coq) and nothing about Iris or its construction. Hence, the adequacy theorem above establishes soundness of well-bracketed program logic.

Following the strategy of Iris’s ordinary program logic, 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 \( \text{VAE-spec} \) we ascribe to VAE is similar to the specification \( \text{awkward-spec} \) of the Awkward example we saw earlier, except that it is expressed in terms of well-bracketed Hoare triples.

\[
\{\text{True}\}
\]

\[
\begin{align*}
\text{let } r = \text{ref}(0) \text{ in } \lambda f. \ r \gets 0; f (); r \gets 1; f (); ! r \\
\forall g. \forall f. \{\{\text{True}\} \ f () \ (x. \ x = ())\} g f \{x. \ x = 1\}
\end{align*}
\]

(VAE-spec)

In the proof we first allocate the location \( \ell \), and obtain \( \ell \mapsto 0 \), and then we allocate an instance \( y \) of the one-shot resource using the rule \( \text{wbHoare-create-stack} \) to create a new stack and pick the invariant \( \text{VAE-invariant} \) below and \( \text{stack}_N(N) \) as the predicate \( R \). We use the resources we have at hand together with the rule \( \text{stacks-push} \) to obtain \( \text{stack}_N(N, [y]) \) and \( \text{stack}_N(N, [y]) \) in order to establish the invariant and return \( \text{stack}_N(N, [y]) \) to the program logic.

\[
\text{VAEInv} \triangleq \exists y, s. \ \text{stack}_N(N, y :: s) \ast (\ell \mapsto 0 \ast \text{pending}(y)) \lor (\ell \mapsto 1 \ast \text{shot}(y)) \]

(\( \text{VAE-invariant} \))

\[\text{4} \text{In the Coq formalization, following Iris’s terminology, the Safe predicate is called } \text{adequate}.\]
Note that the invariant $\text{VAE-invariant}$ is similar to the $\text{awkward-invariant}$ invariant, except for the fact that the name of the one-shot instance is not fixed. Instead, it is the name at the top of the stack $N$. What remains is to show the following for the body of the closure returned by $\text{VAE}$:

$$\langle \text{VAEInv} \ast \text{stack}_\exists(N) \ast \langle \text{True} \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ | \ \ell \gets 0; \ f (); \ \ell \gets 1; \ f (); \ \ell | \ f (x, x = 1) \rangle \ (4)$$

To proceed, we apply the rule $\text{wbHoare-access-stack}$ to obtain $\text{stack}_\bullet(N, s')$ for some stack $s'$, which leaves us with having to prove:

$$\langle \text{VAEInv} \ast \text{stack}_\bullet(N, s') \ast \langle \text{True} \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ | \ \ell \gets 0; \ f (); \ \ell \gets 1; \ f (); \ \ell | \ f (x, x = 1) \rangle \ (5)$$

Now, since the store operation $\ell \gets 0$ is atomic, we can open the invariant (by isolating the store operation using the $\text{wbHoare-bind}$) to obtain $\text{stack}_o(N, y_1 :: s)$ which we use along with the rule $\text{STACKS-AGREE}$ to unify $s'$ with $y_1 :: s$, which leaves us having to prove the following (note how the postcondition of (6) implies the invariant as required):

$$\langle \text{VAEInv} \ast \text{stack}_\bullet(N, y_1 :: s) \ast \text{stack}_o(N, y_1 :: s) \ast \rangle \ (N)$$

and the following for the code after the store operation:

$$\langle \text{VAEInv} \ast (\text{pending}(y_1) \lor \text{shot}(y_1)) \ast \text{stack}_\bullet(N, y_2 :: y_1 :: s) \ast \langle \text{True} \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ | \ \ell \gets 0; \ f (); \ \ell \gets 1; \ f (); \ \ell | \ f (x, x = 1) \rangle \ (7)$$

Observe that we have to show two triples because we have applied the $\text{wbHoare-bind}$ rule (from bottom to top) and that we “made a good choice” of $\Psi$ in $\text{wbHoare-bind}$, 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 $(\text{pending}(y_1) \lor \text{shot}(y_1))$ (cf. the precondition of (7)). Nonetheless, by creating a new instance $y_2$ of the one-shot resource we have managed to reestablish the invariant, this time by taking $y$ to be $y_2$ and the stack $s$ to be $y_1 :: s$. In addition, since we keep hold of $\text{stack}_\bullet$, we remember the exact contents of the stack even after reestablishing the invariant. To show that (7) holds, we use the $\text{wbHoare-bind}$ and $\text{wbHoare-frame}$ (to frame $\text{VAEInv}$ and $\text{pending}(y_1) \lor \text{shot}(y_1)$) rules, which leaves us having to prove the following:

$$\langle \text{stack}_\bullet(N, y_2 :: y_1 :: s) \ast \langle \text{True} \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ f \ \langle \ x, \ x = (\ ) \ast \text{stack}_\bullet(N, y_2 :: y_1 :: s) \rangle \ (N) \ (8)$$

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

$$\langle \text{VAEInv} \ast (\text{pending}(y_1) \lor \text{shot}(y_1)) \ast \text{stack}_\bullet(N, y_2 :: y_1 :: s) \ast \rangle \ (N)$$

$\langle \text{True} \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ f \ \langle \ x, \ x = (\ ) \rangle \ | \ \ell \gets 1; \ f (); \ \ell \gets 1; \ f (); \ \ell | \ f (x, x = 1) \rangle \ (9)$

5Note that we retain the invariant as it is persistent. Also, in the Hoare triple for the store operation we are implicitly framing the specs for $f$. 

Proc. ACM Program. Lang., Vol. POPL, No. 4, Article 111. Publication date: August 2023.
To establish (8) we simply use the rule \texttt{wbHOARE-MEND-STACK} and then it suffices to show the following trivial triple:

\[
\langle\langle \text{True} \rangle \ f \ () \ \langle x. \ x = () \rangle \ f \ () \ \langle x. \ x = () \rangle
\]

We proceed with proving (9) by again using the \texttt{wbHOARE-BIND} rule to isolate the store operation and the rule \texttt{wbHOARE-INV-OPEN} to open the invariant as before, this time unifying the stacks of \texttt{stack\_s} (which we have) and \texttt{stack\_o} (which we obtain from the invariant), from which we learn that the current instance name of the one-shot resource tracking the state of \( \ell \) is in fact \( \gamma_2 \):

\[
\begin{align*}
\langle \langle \text{VAEInv} * (\text{pending}(\gamma_1) \lor \text{shot}(\gamma_1)) * \text{stack\_s}(N, \gamma_2 :: \gamma_1 :: s) * \text{stack\_o}(N, \gamma_2 :: \gamma_1 :: s) * \rangle \rangle \\
\langle \langle \ell \mapsto 0 * \text{pending}(\gamma_2)) \lor (\ell \mapsto 1 * \text{shot}(\gamma_2)) \rangle \ 
\ell \leftarrow 1 \\
\langle x. \ x = () * \text{stack\_s}(N, \gamma_1 :: s) * \text{stack\_o}(N, \gamma_1 :: s) * \ell \mapsto 1 * \text{shot}(\gamma_1) \rangle^{[N]}
\end{align*}
\]

(10)

and the following for the code after the store operation:

\[
\begin{align*}
\langle \langle \text{VAEInv} * \text{stack\_s}(N, \gamma_1 :: s) * \text{shot}(\gamma_1) * \langle \text{True} \rangle \ f \ () \ \langle x. \ x = () \rangle \rangle \\
\ f \ () ; ! \ell \\
\langle x. \ x = 1 * \text{stack\_s}(N, \gamma_1 :: s) \rangle^{[N]}
\end{align*}
\]

(11)

To prove (10), we use the rule \texttt{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 \texttt{VAE} or not. Nonetheless, we are performing a store operation storing 1 into \( \ell \). Furthermore, whether we have obtained \texttt{pending}(\gamma_2) or \texttt{shot}(\gamma_2) from the invariant, either resource is not of any use to us anymore and we hence discard it.\footnote{Iris is an affine separation logic and hence we can always discard resources as necessary.} At this point, the only missing piece (to ensure that we have indeed applied the \texttt{wbHOARE-BIND} rule correctly) is to obtain \texttt{shot}(\gamma_1). However, we have \texttt{pending}(\gamma_1) \lor \texttt{shot}(\gamma_1), left over from the last time we opened the invariant. In either case, we can obtain \texttt{shot}(\gamma_1); in the case of the left disjunct we simply use the rule \texttt{SHOOT}. Note that since \texttt{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 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:

\[
\langle \langle \text{VAEInv} * \text{stack\_s}(N, \gamma_1 :: s) * \text{shot}(\gamma_1) \rangle \ ! \ell \ \langle x. \ x = 1 * \text{stack\_s}(N, \gamma_1 :: s) \rangle^{[N]}
\]

(12)

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

Note how, save for the use of the stack, the overarching argument of the proof of \texttt{VAE} resembles the proof of the \texttt{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 \texttt{VAE} arises from the fact that the callback \( f \) can itself invoke the closure returned by \texttt{VAE}. Here is a simple example where this happens:

\[
\text{let } g = \text{let } r = \text{ref}(0) \ \text{in} \ \lambda f. \ r \leftarrow 0; \ f () ; r \leftarrow 1; f () ; ! r \ \text{in} \ g (\lambda _. \ g (\lambda_. ()());()) \quad (\text{VAE-self-applied})
\]
It is now quite simple to prove that \( \text{VAE-self-applied} \) returns \( 1 \): Using the specification \( \text{VAE-spec} \) we only need to show the following:

\[
L \forall f. \langle\langle \text{True}\rangle f () \langle x. x = ()\rangle g f \langle x. x = 1\rangle \rangle g (\lambda_. \ g (\lambda_. ()); (); ) (\langle x. x = 1\rangle)
\]  

(13)

To prove (13) we simply use the Hoare triple in the precondition of (13), after which we only need to show:

\[
L \forall f. \langle\langle \text{True}\rangle f () \langle x. x = ()\rangle g f \langle x. x = 1\rangle \rangle g (\lambda_. (); (); ) (\langle x. x = ()\rangle)
\]  

(14)

To prove (14) we use the \texttt{wbHoare-bind} rule together with the Hoare triple in the precondition once more, which means we have to show two things: that the program, after the call to \( g \), returns () which is obvious, as well as the following, which is again trivial:

\[
\langle\langle \text{True}\rangle (\lambda_. ()); () \langle x. x = ()\rangle\rangle
\]  

(15)

Applying the Adequacy theorem above yields that running the program \( \text{VAE-self-applied} \) (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 defined it for 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). In other words, the well-bracketed program logic does not rely on a syntactic restriction of programs and instead arguably captures the notion of well-bracketedness at a semantic level: 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, and 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, which also includes the theory of stacks we have seen before (\( \text{stack}_\bullet \), \( \text{stack}_\circ \), and \( \text{stack}_\exists \)). 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:

\[
\{ P \} e \{ \Phi \} \triangleq \square (P \rightarrow wp e \{ \Phi \})
\]

Here, \( \square \) 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

\footnote{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.}
by \( P \). In other words, the precondition is not underapproximating the necessary conditions for correctness of \( e \). The connective \( \rightarrow \), pronounced magic wand, or simply wand, is the separating implication; it is to separating conjunction \( \ast \), what implication \( \Rightarrow \) is to ordinary conjunction \( \land \).

Similarly, we define well-bracketed Hoare triples in terms of well-bracketed weakest preconditions, which themselves will be defined in terms of ordinary Iris weakest preconditions.

\[
\langle P \rangle e \langle \Phi \rangle^O \triangleq \Box \left( P \rightarrow \text{wbwp} e \langle \Phi \rangle^O \right)
\]

Well-bracketed weakest preconditions in turn are defined as follows:

\[
\text{wbwp} e \langle \Phi \rangle^O \triangleq \forall S. \text{AllStacksExcept}(S, O) \rightarrow
\]

\[
\text{wp} e \{ x. \exists S'. S \subseteq S' \ast \text{AllStacksExcept}(S', O) \ast \Phi(x) \} \quad \text{(wbwp-definition)}
\]

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

All the rules governing the \( \text{AllStacksExcept} \) predicate which are necessary for validating the rules of the well-bracketed program logic are given in Figure 4. The rule \text{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 \( \text{stack}_\ast \) whenever the stack is tracked by \( \text{AllStacksExcept} \). On the other hand, the set \( O \) in \( \text{AllStacksExcept} \) are exactly the set of stacks that are not tracked by \( \text{AllStacksExcept} \),
as evidenced by the rules \texttt{STACK-SYSTEM-IS-OUT} and \texttt{CHANGE-OUT-STACK}. The latter states that the map \( S \) in \texttt{AllStacksExcept}(\( S, O \)) can be \textit{arbitrarily} updated for any stack that is not tracked by \texttt{AllStacksExcept}. The rules \texttt{STACK-TAKE-OUT} and \texttt{STACK-PUT-BACK} respectively allow us to take stack propositions out of \texttt{AllStacksExcept} and put them back (as long as they agree) with the map of stacks. These rules together with \texttt{CHANGE-OUT-STACK} allow us to take a stack out, update it \textit{arbitrarily} and then put it back. There is no restriction on how the stack may evolve as far as the theory around \texttt{AllStacksExcept} is concerned. The stack discipline is instead enforced in the definition of well-bracketed weakest preconditions \texttt{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: \( S \subseteq S' \)). Finally, the rule \texttt{create-stack} allows us to create fresh stacks whose stack part initially remains within the \texttt{AllStacksExcept} predicate.

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 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 \texttt{wbHoare-create-stack}, \texttt{wbHoare-access-stack}, and \texttt{wbHoare-mend-stack}), then we can prove soundness of that rule in the well-bracketed program logic based on the proof in the underlying program logic. The second is that we can combine reasoning steps of the two program logics, as witnessed by rule \texttt{Hoare-wbHoare}.

As for the bespoke rules of the well-bracketed program logic, \textit{i.e.}, \texttt{wbHoare-create-stack}, \texttt{wbHoare-access-stack}, and \texttt{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 can consult our rather short proofs of these rules in our Coq development. The other rules we divide into two groups, those general rules that apply to all instantiations of the logic, \textit{e.g.}, \texttt{wbHoare-consequence}, \texttt{wbHoare-inv-open}, \texttt{wbHoare-bind}, \texttt{etc.}, and those that reflect the semantics of the basic building blocks of the language, \textit{e.g.}, \texttt{wbHoare-alloc}, \texttt{wbHoare-load}, \texttt{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, and using the Iris rules for Iris’s weakest preconditions. To derive the rules for the basic building blocks of the language, we simply apply rule \texttt{Hoare-wbHoare} to lift the operation’s Hoare triple in the underlying program logic to the well-bracketed program logic.

Proving the rule \texttt{Hoare-wbHoare} is straightforward by unfolding the definition of well-bracketed triples. Indeed, 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 \texttt{Hoare-wbHoare} may seem surprising in that it seems to allow “side-stepping” the well-bracketed logic completely, by carrying all reasoning in the usual program logic then lifting the proof in 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 \texttt{VAE})! Since proving \texttt{VAE-spec} requires both \texttt{establishing} and \texttt{assuming} well-bracketed Hoare triples, one cannot simply use \texttt{Hoare-wbHoare} to get “for free” a well-bracketed specification of \texttt{VAE}.

The rule \texttt{Hoare-wbHoare} is nevertheless an interesting rule, in that it allows us to prove that innocuous uses of non-well-bracketed programs (\textit{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 \texttt{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. To do so, we use the following example:

\[
\begin{aligned}
&\text{let } g = \\
&\quad \text{let } \text{first} = \text{ref}(\text{true}) \text{ in } \text{let } \text{res} = \text{ref}(\text{None}) \text{ in } \\
&\quad \text{let } \text{rec } \text{wait}() = \text{if } ! \text{res} = \text{None} \text{ then } \text{wait}() \text{ else } () \text{ in } \\
&\quad \quad \lambda. ; \text{if } ! \text{first} \text{ then } \text{first} \leftarrow \text{false}; \text{fork } \{ \text{res} \leftarrow \text{Some}(\text{fact 1000000}) \} \text{ else } \text{wait}() \text{ in } \\
&\quad \text{let } h = \text{let } r = \text{ref}(0) \text{ in } \lambda f. r \leftarrow 0; f(); r \leftarrow 1; f(); ! r \text{ in } \\
&\quad h \ g \quad \text{(innocuous-concurrency)}
\end{aligned}
\]

This program applies VAE to a function whose execution depends on whether it is the first time it is executed or not. The first execution, i.e., when first stores true, forks a thread that computes the factorial of a large number and stores it in res. Subsequent calls will simply wait for the computation of the large factorial to terminate, i.e., for res not to be None. Clearly, in many runs the factorial does not terminate before the second call to function g. (The operational semantics of HeapLang makes no assumptions regarding scheduling.) Despite this, this program is well-bracketed as far as VAE is concerned, in the sense that we can show the following:

\[
\langle \text{True} \rangle \ h \ g \ \langle x. \ x = 1 \rangle
\]

where h and g are as in innocuous-concurrency above. To prove this, it suffices (given that we have proven VAE-spec earlier) to show:

\[
\langle \text{True} \rangle \ g \ \langle x. \ x = () \rangle
\]

and by Hoare-wbHoare it then suffices to show the Iris triple

\[
\{ \text{True} \} \ g \ \langle x. \ x = () \rangle
\]

and this is a rather easy exercise in concurrent separation logic; one simply uses the following invariant after allocating the two references first and res:

\[
\exists b \in \{ \text{true, false} \} . \text{first} \leftrightarrow b \ast (\text{res} \leftrightarrow \text{None} \lor \exists n \in \mathbb{N}. \text{res} \leftrightarrow \text{Some}(n))
\]

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, and prove 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 $\text{VAESpec}$ and parameterized by the implementation $e$ and initial resources $P_0$:

$$\text{VAESpec}(e, P_0) \triangleq \{P_0\} \{g. \forall f. \{(\text{True}) \ f \ (x. \ x = (())) \} \ g \ \{(x. \ x = 1)\}\}$$

(\text{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; it is strictly stronger.

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 higher-order argument of our program, 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\((\text{call})\) returns $\tau$, then the event $\langle \tau, \text{call} \rangle$ has been added to the trace. We then use emit to simply add an event to the trace: emit\((\tau, \text{ret})\) adds the event $\langle \tau, \text{ret} \rangle$. In other words, for each call to $f()$ we emit a pair of call and ret events with a matching unique identifier $\tau$.

$$\text{let instrument}(e) = \text{let } g = e \text{ in } \lambda f. \ g (\lambda \_ . \ \text{let } \tau = \text{fresh}(\text{call}) \text{ in } f(); \ \text{emit}(\tau, \text{ret}))$$

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

$$L_{\text{seqfull}} ::= \langle \tau, \text{call} \rangle \cdot L_{\text{seqfull}} \cdot \langle \tau, \text{ret} \rangle \mid L_{\text{seqfull}} \cdot L_{\text{seqfull}} \mid \varepsilon \quad (\tau \in \text{Tag} \triangleq \text{String})$$

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

$$\text{VAESpec}(e, \text{True}) \rightarrow \text{VAESpec}(\text{instrument}(e), \text{trace}(e) \ast \text{traceInv}(L_{\text{seq}}))$$

(instrument-trace)

Technically speaking, we use Birkedal et al. [2021]'s traceInv predicate to indicate that $L_{\text{seq}}$ 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 (a variant of Adequacy which instead of Safe$\_\phi$ 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 which enables 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 $F_{\mu,ref}$, 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 type. 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:

```
let r = ref (0) in \f. r <- 0; f (); r <- 1; f (); if ! r # 1 then () () else 1
```

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. Even though the program VAE-check is not syntactically well-typed (due to the () () part treating the unit value () as a function), it is semantically type sound of the expected type: $(1 \rightarrow 1) \rightarrow Z$. Formally, we write this as follows:

```
\emptyset | \emptyset \vdash VAE-check : (1 \rightarrow 1) \rightarrow Z
```

Here, the relation $\Xi | \Gamma \vdash e : \tau$ 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 $F_{\mu,ref}$ is given in Figure 5. At the heart of this model are the so-called expression interpretation $[\cdot]^e$ and value interpretation $[\cdot]$ which are, respectively unary relations (predicates) on closed

---

8Sequential System $F$ featuring both universal and existential types, recursive types, and higher-order references.
expressions and values. These relations have types $\text{Expr} \to \text{iProp}$ and $\text{Val} \to \text{iProp}$, respectively. Here $\text{iProp}$ is the universe of Iris propositions and $\text{iProp}_\text{pc}$ is the universe of persistent Iris propositions. The value interpretation should be persistent while expressions need not be because in our CBV semantics expressions are evaluated only once and their values can be freely duplicated and reused multiple times. The expression and value interpretations are defined in a mutually recursive fashion by induction on types. 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 $\text{Val} \to \text{iProp}_\text{pc}$. 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] thereby enabling 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, $\text{1}$, $\text{B}$, and $\text{Z}$, 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

---

9In practice (in the Coq development) this conceptual mutuality is broken by defining the expression interpretation first defined as a general predicate transformer of the type $(\text{Val} \to \text{iProp}_\text{pc}) \to (\text{Expr} \to \text{iProp})$ which is essentially just the (well-bracketed) weakest precondition. The value interpretation is then defined based on that based this general predicate transformer.
underlying terms again in the value interpretation for the corresponding type. The interpretation of the function types requires that a value of the function type must take (when applied to) a value in the interpretation of the domain type to an expression of the codomain type. Universal and existential types are interpreted using the quantifiers of the logic quantifying over arbitrary value interpretation predicates. We write $\nu(\cdot)$ for specialization of a polymorphic value to a type which we omit in the program expression; note that $\nu(\cdot)$ 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 the fixed-point combinator of the Iris logic — $\mu r. P$ is always defined and is a fixed point $(\mu r. P \equiv P[\mu r. P/r])$ whenever the variable $r$ appears only guarded in $P$, i.e., under a later modality, $\triangleright$. Intuitively, the interpretation of a recursive type $\mu a. \tau$ says that a value must be a folded value where the underlying unfolded value is of value type $\tau$ where $a$ is itself interpreted as the recursive type $\mu a. \tau$. Finally, the value interpretation of a reference type $\text{ref}(\tau)$ requires that values be memory locations always storing a value in the interpretation of $\tau$ — note that we consider locations to be of a syntactically distinct class from the rest of the values with an implicit injection to them; hence the form $\exists (\ell : \text{Loc}). (v = \ell) \triangleright \ldots$. 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 — 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 $\Xi | \Gamma \vdash e : \tau$ 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 all possible interpretations $\vec{\delta}$ for the free type variables. For this purpose we define the typing context interpretation $[\Gamma]^{\vec{v}}_{\vec{\delta}}$ which requires 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 $[(1 \to 1) \to Z]^{\vec{v}}_{\vec{\delta}}$ (VAE-check) which in turn can be expanded and simplified into a form which very closely resembles VAE-spec of Section 3.2 by unfolding expression and value interpretations, simplifying $[1]_{\vec{\delta}}$ which is a singleton, and finally folding well-bracketed weakest preconditions into equivalent well-bracketed Hoare triples:

$$\text{wbwp VAE-check } (g. \forall f. (\text{True}) \rightarrow f () (y. y = ())) \rightarrow g f (y. y \in Z)$$

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 easily 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 $\Xi | \Gamma \vdash e : \tau$ and $\Xi | \Gamma \vdash e' : \tau$, we say that $e$ contextually refines a program $e'$, written $\Xi | \Gamma \vdash e \xrightarrow{\text{ctx}} e' : \tau$, 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 $\Xi | \Gamma \vdash e \xrightarrow{\text{ctx}} e' : \tau$, we often refer to $e$ as the implementation and $e'$ is the specification. We say that $e$ is contextually equivalent to $e'$, written $\Xi | \Gamma \vdash e \approx_{\text{ctx}} e' : \tau$, if both $\Xi | \Gamma \vdash e \xrightarrow{\text{ctx}} e' : \tau$ and $\Xi | \Gamma \vdash e' \xrightarrow{\text{ctx}} e : \tau$ 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 $\Xi \vdash \Gamma \vdash e \lessdot_{\text{log}} e' : \tau$, then we can conclude $\Xi \vdash \Gamma \vdash e \lessdot_{\text{ctx}} e' : \tau$.

We use our binary logical relations model to prove that VAE is contextually equivalent to the program $\lambda f. \ f () ; \ f () ; \ 1$ at type $(1 \to 1) \to \mathbb{Z}$ which 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 \texttt{wbHoare-create-stack}, and proceed to establish the exact invariant as in \texttt{VAE-invariant} and follow a similar proof strategy to show that the VAE program returns 1 regardless which side of refinement it appears on. For the detailed proof, see the 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, and we have 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 in that 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 current state of the state transition system. These predicates are defined using the monotone resource algebra \cite{Timany and Birkedal 2021} to construct a resource that can only be updated in accordance with the public transition relation with private transitions being 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 \texttt{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 and rather 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 \texttt{AllStacksExcept}, \texttt{stack\_s}, \texttt{stack\_o}, and \texttt{stack\_\exists} we introduce the following new propositions which are themselves directly defined in terms of ghost resources: \texttt{stack\_IN}, \texttt{stack\_IN}, \texttt{EntireDom}, and \texttt{SubsetOfDom}. The predicates \texttt{AllStacksExcept}, \texttt{stack\_s}, \texttt{stack\_o}, and \texttt{stack\_\exists} are defined as follows:

\[
\begin{align*}
\text{stack\_s}(N, s) &\triangleq \text{stack\_IN}(N, s) \ast \text{SubsetOfDom}([N]) \\
\text{stack\_o}(N, s) &\triangleq \text{stack\_IN}(N, s) \ast \text{SubsetOfDom}([N]) \\
\text{stack\_\exists}(N) &\triangleq \text{SubsetOfDom}([N])
\end{align*}
\]

\[
\text{AllStacksExcept}(S, O) \triangleq O \subseteq \text{dom}(S) \ast \text{EntireDom}([\text{dom}(S)]) \ast \bigotimes_{N \in \text{dom}(S) \setminus O} \text{stack\_s}(N, S(N))
\]

The \texttt{SubsetOfDom} predicate is the only predicate among these predicates that is persistent. The predicates \texttt{SubsetOfDom} and \texttt{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 \(\text{SubsetOfDom}\) predicate is a subset of the set tracked by \(\text{EntireDom}\):

\[
\text{SubsetOfDom}(A) * \text{EntireDom}(B) \vdash A \subseteq B
\] (\text{dom-subset})

\[
\text{SubsetOfDom}(A) * \text{SubsetOfDom}(B) \vdash \text{SubsetOfDom}(A \cup B)
\] (\text{dom-distributes})

\[
\text{EntireDom}(B) * \text{finite}(A) \vdash \Rightarrow \text{EntireDom}(A \cup B) * \text{SubsetOfDom}(A)
\] (\text{dom-grow})

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

\[
\text{stack}^{\text{IN}}_\circ(N, s) * \text{stack}^{\text{IN}}_\circ(N, s') \vdash s = s'
\] (\text{stacks}^{\text{IN}}_\circ\text{-agree})

\[
\text{stack}^{\text{IN}}_\circ(N, s) * \text{stack}^{\text{IN}}_\circ(N, s') \vdash \text{False}
\] (\text{stacks}^{\text{IN}}_\circ\text{-unique})

\[
\text{stack}^{\text{IN}}_*(N, s) * \text{stack}^{\text{IN}}_*((N, s')) \vdash \text{False}
\] (\text{stacks}^{\text{IN}}_*\text{-unique})

\[
\text{stack}^{\text{IN}}_*(N, s) * \text{stack}^{\text{IN}}_*(N, s') \vdash \Rightarrow \text{stack}^{\text{IN}}_*(N, s''') * \text{stack}^{\text{IN}}_*(N, s''')
\] (\text{stacks}^{\text{IN}}_\circ\text{-update})

\[
\text{infinite}(A) \vdash \Rightarrow \exists N \in A. \text{stack}^{\text{IN}}_\circ(N, s) * \text{stack}^{\text{IN}}_\circ(N, s)
\] (\text{stacks}^{\text{IN}}_\circ\text{-create})

Note how these rules specify that the stacks always agree (\text{stacks}^{\text{IN}}_\circ\text{-agree}), that they are initially created in a way that they agree (\text{stacks}^{\text{IN}}_\circ\text{-create}), and that they may only be updated as long as the equality is maintained (\text{stacks}^{\text{IN}}_\circ\text{-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 \(\text{stack}_\circ\) or \(\text{stack}_*\), for some stack \(N\), then the stack \(N\) must indeed be in the map tracked by \text{AllStacksExcept}; the latter otherwise would only be a collection of \(\text{stack}_*\) 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 \(\text{stack}_*\) and \(\text{stack}_\circ\): \text{STACKS-AGREE, STACK-SYSTEM-UNIQUE, STACK-USER-UNIQUE, STACK-EXISTS, STACKS-PUSH, and STACKS-POP} — the stack-like behavior of the rules \text{STACKS-PUSH} and \text{STACKS-POP} is not enforced at the level of resources but only by the “interface” of the rules for updating \(\text{stack}_*\) and \(\text{stack}_\circ\).

It only remains for us to justify that the definitions of \text{AllStacksExcept}, \(\text{stack}_*\), \(\text{stack}_\circ\), and \(\text{stack}_\exists\) do indeed satisfy the rules presented in Figure 4. The rule \text{MASK-SUBSET-DOM} follows by definition. For the rule \text{STACK-EXISTSS-IN} we essentially only need to apply \text{dom-subset}. As for the rule \text{STACK-USER-NOT-OUT}, we will first apply the rule \text{STACK-EXISTS} together with \text{MASK-SUBSET-DOM} to obtain that \(N \in \text{dom}(S)\). Given that \(N \notin O\), the proposition \(\text{stack}_*(N, S(N))\) must be part of the big separating conjunction in the definition of the \text{AllStacksExcept} proposition which allows us to conclude the proof by applying the rule \text{STACKS-AGREE}. For the rule \text{STACK-SYSTEM-IS-OUT}, assume the contrary, i.e., that \(N \notin O\). Since \(\text{SubsetOfDom}(\{N\})\) is included in the definition of \(\text{stack}_*(N, s)\), we have \(N \in \text{dom}(S)\). This means that we can obtain \(\text{stack}_*(N, S(N))\) from the big separating conjunction in the definition of the \text{AllStacksExcept} proposition which is a contradiction as per \text{STACK-SYSTEM-UNIQUE}. The rules \text{STACK-TAKE-OUT} and \text{STACK-PUT-BACK} simply take the system part of the stack, \(\text{stack}_*\), out of the big separating conjunction in the definition of \text{AllStacksExcept} and put it back in there. Note that there is nothing in the definition of \text{AllStacksExcept}(S, O) that pins down the contents of a stack that is in the mask \(O\). Hence, the contents of such a stack can be changed arbitrarily in the \(S\) part of \text{AllStacksExcept}(S, O) as stated in the rule \text{CHANGE-OUT-STACK}. In order to validate the rule \text{CREATE-STACK} we need only to allocate a new stack \(N\) using the rule \text{stacks}^{\text{IN}}\text{-create}\ and subsequently use the rule \text{dom-grow} to update the \text{EntireDom} and obtain \text{SubsetOfDom}(\{N\}) so that we can construct \(\text{stack}_*\) and \(\text{stack}_\circ\) 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 \text{dom}(S). This is why in the rule \text{stacks}^{\text{IN}}\text{-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 $\text{dom}(S)$ 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 $\text{VAE}$ 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 [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 $\text{VAE}$ 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.

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 $\text{VAE}$ 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 \textit{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 \textit{kind} instead of what would normally be called a type, \textit{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 where the value of the location is asserted to be in the set $\{0, 1\}$ and to be 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 \texttt{wbHoare-mend-stack} and \texttt{wbHoare-access-stack}, the major difference being that our program logic decouples ownership (capabilities in Pottier [2009]) from the well-bracketedness mechanism, \textit{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.

\textbf{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, \textit{i.e.}, it enforces that all the uses of call/cc in their system 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, \textit{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.

\section{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 where 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

REFERENCES


Proc. ACM Program. Lang., Vol. POPL, No. 4, Article 111. Publication date: August 2023.


