Program logics à la carte

MAX VISTRUP, ETH Zurich, Switzerland
MICHAEL SAMMLER, ETH Zurich, Switzerland
RALF JUNG, ETH Zurich, Switzerland

Program logics have proven a successful strategy for verification of complex programs. By providing local reasoning and means of abstraction and composition, they allow reasoning principles for individual components of a program to be combined to prove guarantees about a whole program. Crucially, these components and their proofs can be *reused*. However, this reuse is only available once the program logic has been defined. It is a frustrating fact of the status quo that whoever defines a new program logic must establish every part, both semantics and proof rules, from scratch. In spite of programming languages and program logics typically sharing many core features, reuse is generally not available across languages. Even inside one language, if the same underlying operation appears in multiple language primitives, reuse is typically not possible when establishing proof rules for the program logic.

To enable reuse across and inside languages when defining complex program logics (and proving them sound), we serve program logics *à la carte* by combining program logic fragments for the various effects of the language. Among other language features, the menu includes shared state, concurrency, and non-determinism as reusable, composable blocks that can be combined to define a program logic modularly. Our theory builds on ITrees as a framework to express language semantics and Iris as the underlying separation logic; the work has been mechanized in the Coq proof assistant.

1 Introduction

Program logics are a widely successful approach to program verification [12, 40, 36, 8, 2, 24, 26, 16]. However, they require a non-trivial amount of preparation, especially for programs written in complicated languages. First, a *formal definition of the language semantics* needs to be developed, Then, one must find and state the *rules of the program logic*. Finally, the two need to be connected by a *soundness proof*.

Let us consider what is required to build a program logic for a new language. Probably our language has some form of global mutable state; maybe it has concurrency; maybe it has other forms of non-determinism. The dominant approach to defining language semantics is to use an operational semantics, which has standard ways to model all these language features. To obtain a program logic, again there are common ways of reasoning about such features (the combination of mutable state and concurrency makes concurrent separation logic a common choice), so we know the rough shape the logic takes, and we just have to make some adjustments to account for the particularities of our concrete language. Maybe the language has some particularly complicated operation that does many things at once; this will require a complicated transition in the operational semantics and an associated complicated proof rule in the program logic. To justify the correctness of the logic, we again have to do proofs that are largely standard.

All of this is a lot of work! If one aims to fully formalize the entire logic and soundness proof (whether on paper or in a mechanized proof), one ends up proving very similar theorems for each new language. The *pattern* is always the same, but there is no reusable theorem that would let us, say, "plug in" a heap with associated points-to assertions $\ell \mapsto v$ to obtain the standard separation logic reasoning principles.

Fundamentally, this is caused by the fact that typical language definitions are monolithic: an operational semantics captures *all* the state that is relevant for the behavior of the program, and a *single* relation describes how all language constructs act on the entire state. Operational semantics

Authors' Contact Information: Max Vistrup, contact@maxv.dk, ETH Zurich, Switzerland; Michael Sammler, michael.sammler@inf.ethz.ch, ETH Zurich, Switzerland; Ralf Jung, research@ralfj.de, ETH Zurich, Switzerland.

provide no clear way to define the operational behavior of a heap once and for all, and turn it into a reusable component with associated reasoning principles. They also provide no good way to compose a single, complicated operation from smaller pieces: every program step is a single state transition. This becomes particularly onerous for concurrent languages where the individual steps of a small-step operational semantics typically mark the granularity of atomicity, meaning that one cannot simply define a complicated atomic operation as syntactic sugar composed of many smaller operations.

In this paper, we present *program logics à la carte*: a new approach for defining program semantics and associated program logics from reusable building blocks.

To achieve this, we leverage *ITrees* [41] to represent program semantics in a style that is closer to denotational semantics rather than operational semantics. ITrees provide a general monadic encoding of effectful computations in a pure meta-language. They are parametric in the set of effects that the program may invoke. Examples of such effects are non-determinism, state, or concurrency. In this work, we consider these effects to be the *building blocks* that make up a program semantics.

The core of our work is a general program logic for arbitrary ITrees. Just like ITrees are parameterized over effects, our program logic is parameterized over *logical effect handlers* that define the verification condition for invoking an effect (think: preconditions and postconditions). We have implemented logical effect handlers for a number of common effects, including mutable state, non-deterministic choice (both demonic and angelic), concurrency, and abnormal program failure. These are the building blocks that a language designer has at their disposal, and they all come with an associated program logic fragment that is established once-and-for-all, and an adequacy theorem showing soundness of this program logic fragment.

A language designer can pick the effects of their choice from this menu and describe the language semantics by denoting their language into ITrees, composing the primitive operations provided by these effects to build up the core language operations as needed. They can then use the general ITree program logic to define a language-specific program logic. The program logic fragments of each effect are automatically available, and their proof rules can be used to build up the reasoning principles for the core language operations. Crucially, we are able to use the compositional power of program-logic-based reasoning to establish the program logic itself. ITrees serve as a common foundation for effectful computation—a shared domain with a wide range of applicability.

Our program logic is based on Iris [18], which is a natural choice since Iris already has a strong focus on modularity and reuse: Iris is a "separation logic *framework*" designed to serve as the foundation for domain-specific separation logics [17, 5, 30, 29, 13, 32, 25]. Iris already provides reusable building blocks for "ghost state" constructions to capture common reasoning patterns (such as finite maps with fractional per-key permissions, or append-only lists). However, so far only very limited reuse was possible for the part of the program logic that directly interacts with the language semantics, leading to a lot of duplicated effort across Iris-based projects. With our new approach, this is no longer necessary: language components and their associated program logics rules can be shared and reused with the same ease that Iris users already commonly share and reuse purely logical components.

Contributions. The key contributions of our work are a novel, general-purpose program logic for ITrees and a library of effects with associated reasoning principles expressed in that program logic. We have built effect libraries for concurrency, global state, demonic and angelic non-determinism, safe and unsafe program termination, and partial correctness. The program logic is proven sound (or "adequate") w.r.t. two notions of execution for ITrees: the typical ITree approach involving gradual interpretation of events, and a novel translation of ITrees into state machines. To demonstrate the potential for reuse and the applicability of this framework, we have ported two existing Iris-based

$$
\begin{aligned}
[\![v]\!] &:= \mathsf{Ret}(v) \\
[\![e_1 \mathbin{\widehat{+}} e_2]\!] &:= v_1 \leftarrow [\![e_1]\!]; v_2 \leftarrow [\![e_2]\!]; z_2 \leftarrow \mathsf{to\_int}(v_2); z_1 \leftarrow \mathsf{to\_int}(v_1); \mathsf{Ret}(z_1 + z_2) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!]; v_2 \leftarrow [\![e_2]\!]; (x, e) \leftarrow \mathsf{to\_lam}(v_1); [\![e[v_2/x]]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &:= v_1 \leftarrow [\![e_1]\!]; z_1 \leftarrow \mathsf{to\_int}(v_1); \text{if } z_1 \neq 0 \text{ then } [\![e_2]\!] \text{ else } [\![e_3]\!] \\
\mathsf{to\_int}(v) &:= \text{match } v \text{ with } z \implies \mathsf{Ret}(z) \mid \_ \implies \text{fail end} \\
\mathsf{to\_lam}(v) &:= \text{match } v \text{ with } \lambda x.\, e \implies \mathsf{Ret}((x, e)) \mid \_ \implies \text{fail end}
\end{aligned}
$$

Fig. 1. Denotational semantics of $\lambda_{\mathbb{Z}}$.

program logics: the default Iris example language, HeapLang, as well as the language used by Islaris [29]. The HeapLang program logic supports both total and partial correctness reasoning and comes with a provably-sound interpreter (including a termination proof). All our work (except for the running example language in §2) is mechanized in the Coq proof assistant [38].

Structure of the paper. The rest of the paper is structured as follows: First, §2 gives an overview of our approach by gradually equipping a language with more and more effects and building up an associated program logic alongside. Then, §3 explains how our program logic and effect libraries are defined in technical detail. Finally, §4 and §5 describe the HeapLang resp. Islaris case studies. We conclude by discussing related work (§6).

2 Key ideas

In this section, we showcase the key ideas of this paper by building a simple example language and an associated program logic step-by-step. Each step adds one more building block, demonstrating how the language and program logic are built up from reusable components.

2.1 Starting point: A pure lambda calculus ($\lambda_{\mathbb{Z}}$)

We start with $\lambda_{\mathbb{Z}}$, a basic untyped λ-calculus with integers, addition, and if-expressions:

$$v \in \mathsf{val} ::= z \mid \lambda x.\, e \quad (z \in \mathbb{Z}) \qquad e \in \mathsf{expr} ::= v \mid e_1 \mathbin{\widehat{+}} e_2 \mid e_1(e_2) \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

To reason about this language, we first need to give it a semantics. For this, we use a denotational semantics with *interaction trees* [41] as our domain. ITrees, short for interaction trees, are represented by the type **itree** $E\ R$ that forms a monad in the result type R. The event type E specifies a set of user-defined events that is used to represent the effects of the language.

The first step in defining the semantics of our language is to pick a suitable set of events E. So far, our language has one effect (not counting non-termination as non-termination is natively supported by ITrees): programs can fail, such as when trying to add a number and a function. To represent failure, we use an event type **FailE** which gives us access to an operation fail : **itree FailE** $\emptyset$. A failure does not return but "crashes" the program and thus the result type of fail is the empty set. To summarize, the set of events for $\lambda_{\mathbb{Z}}$ is:

$$\mathbf{LangE}_{\mathbb{Z}} := \mathbf{FailE}$$

With $\mathbf{LangE}_{\mathbb{Z}}$ at hand, we can define a function $[\![e]\!]$ that maps an expression e into the domain **itree** $\mathbf{LangE}_{\mathbb{Z}}$ val as shown in Figure 1. This is a shallow embedding: pure computations in the language are mapped to computations in the meta-logic, as can be seen for the addition or if-expressions. Operations not supported by the meta-logic are treated monadically. We can also easily represent non-structural recursion (as in the case of function application) thanks to the general

$$\textsc{WpConsequence}\ \frac{\forall r.\, \Phi(r) \vdash \Psi(r) \qquad \mathrm{wp}\, e\, \{\Phi\}}{\mathrm{wp}\, e\, \{\Psi\}} \qquad \textsc{WpFrame}\ \frac{P * \mathrm{wp}\, e\, \{\Phi\}}{\mathrm{wp}\, e\, \{v.\, P * \Phi(v)\}} \qquad \textsc{WpVal}\ \frac{\Phi(v)}{\mathrm{wp}\, v\, \{\Phi\}}$$

$$\textsc{WpBindPlusL}\ \frac{\mathrm{wp}\, e_1\, \{v_1.\, \mathrm{wp}\, v_1 \mathbin{\widehat{+}} e_2\, \{\Phi\}\}}{\mathrm{wp}\, e_1 \mathbin{\widehat{+}} e_2\, \{\Phi\}} \qquad \textsc{WpBindPlusR}\ \frac{\mathrm{wp}\, e_2\, \{v_2.\, \mathrm{wp}\, v_1 \mathbin{\widehat{+}} v_2\, \{\Phi\}\}}{\mathrm{wp}\, v_1 \mathbin{\widehat{+}} e_2\, \{\Phi\}} \qquad \textsc{WpPlus}\ \frac{\Phi(z_1 + z_2)}{\mathrm{wp}\, z_1 \mathbin{\widehat{+}} z_2\, \{\Phi\}}$$

$$\textsc{WpApp}\ \frac{\mathrm{wp}\, e[v/x]\, \{\Phi\}}{\mathrm{wp}\, (\lambda x.\, e)(v)\, \{\Phi\}} \qquad \textsc{WpIfTrue}\ \frac{\mathrm{wp}\, e_1\, \{\Phi\} \qquad z \neq 0}{\mathrm{wp}\, \texttt{if}\ z\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi\}} \qquad \textsc{WpIfFalse}\ \frac{\mathrm{wp}\, e_2\, \{\Phi\}}{\mathrm{wp}\, \texttt{if}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi\}}$$

Fig. 2. Excerpt of the program logic for the $\lambda_{\mathbb{Z}}$.

fixpoint combinator provided by ITrees. $[\![e]\!]$ can be viewed as a form of denotational semantics, but note that events are still uninterpreted at this stage and hence treated purely syntactically.[1]

Program logic. To reason about programs in $\lambda_{\mathbb{Z}}$, we will define a program logic. As the framework for our program logic we use Iris [18], a versatile separation logic that comes with a good foundation of reusable reasoning principles and has already been used as the basis for numerous program logics [17, 5, 30, 29, 13, 25]. Following the usual approach in Iris, we use a *weakest precondition* connective as the core of our program logic. A more traditional Hoare-style program logic can be easily defined on top of this by setting $\{P\}\ e\ \{Q\} \coloneqq P \mathrel{-\!\!*} \mathrm{wp}\, e\, \{Q\}$; the typical rules can be derived.

Concretely, $\mathrm{wp}\, e\, \{\Phi\}$ says that in the current state (which can be constrained by assumptions in the logical context), every execution of e is well-behaved and the returned value v satisfies the postcondition $\Phi(v)$. We obtain the expected rules for wp, as shown in Figure 2: we have the rule of consequence and the specific rules for each language construct. For instance, WpPlus says that $z_1 \mathbin{\widehat{+}} z_2$ satisfies postcondition Φ whenever the corresponding mathematical term $z_1 + z_2$ satisfies Φ. WpBindPlusL is a "bind" rule that recurses into the program structure; it says that to reason about $e_1 \mathbin{\widehat{+}} e_2$, we can first reason about e_1. Every value v_1 that e_1 can evaluate to must then satisfy the property that evaluating $v_1 \mathbin{\widehat{+}} e_2$ satisfies the desired postcondition. WpBindPlusR does the same for the right-hand operand (but this rule only applies if the left-hand operand is a value, indicating a left-to-right evaluation order). We omit similar rules for application and if-expressions. (The frame rule WpFrame will only become relevant when we get to reasoning about state.)

These rules are entirely standard. The key idea of our approach lies in *how* wp is defined: instead of defining a new wp for each language, we want to enable reuse across languages. Therefore, we introduce a new general-purpose weakest precondition connective for *arbitrary ITrees*: given $t : \mathbf{itree}\ E\ R$, we define $\mathrm{wpi}_H\, t\, \{\Phi\}$ as the weakest precondition that ensures t terminates with a return value that satisfies postcondition Φ. The definition takes as input a *logical effect handler* (or just *handler*) H which provides specifications for the events in E. On top of this, we can define the weakest precondition for $\lambda_{\mathbb{Z}}$ by choosing a suitable handler $\mathbf{LangH}_{\mathbb{Z}}$ and then setting:

$$\mathrm{wp}\, e\, \{\Phi\} \coloneqq \mathrm{wpi}_{\mathbf{LangH}_{\mathbb{Z}}}\, [\![e]\!]\, \{\Phi\}$$

The reason this is useful is that wpi satisfies the rules in Figure 3—they come "for free", without us having to do any language-specific work. Aside from the rule of consequence, we have rules for the bind and return operators of the ITree monad, as well as WpiEutt which states that wpi is

[1]Specifically, as we will see in §3.1, ITrees use the free monad to represent events.

$$\textsc{WpiConsequence}\ \frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\, t\, \{\Phi\}}{\mathsf{wpi}_H\, t\, \{\Psi\}} \qquad \textsc{WpiFrame}\ \frac{P * \mathsf{wpi}_H\, t\, \{\Phi\}}{\mathsf{wpi}_H\, t\, \{v.\ P * \Phi(v)\}} \qquad \textsc{WpiEutt}\ \frac{t_1 \approx t_2}{\mathsf{wpi}_H\, t_1\, \{\Phi\} \dashv\vdash \mathsf{wpi}_H\, t_2\, \{\Phi\}}$$

$$\textsc{WpiBind}\ \frac{\mathsf{wpi}_H\, t\, \{x.\ \mathsf{wpi}_H\, k(x)\, \{\Phi\}\}}{\mathsf{wpi}_H\, x \leftarrow t; k(x)\, \{\Phi\}} \qquad \textsc{WpiRet}\ \frac{\Phi(r)}{\mathsf{wpi}_H\, \mathsf{Ret}(r)\, \{\Phi\}}$$

Fig. 3. Basic, generic proof rules for weakest preconditions.

compatible with $\approx$ ("equivalence up to τ"), the canonical extensional notion of equality on ITrees. This is needed to unfold general recursive ITree definitions, such as our $[\![e]\!]$, which can only be unfolded up to $\approx$, not up to full definitional equality.

To complete this definition, we need to define the handler $\mathbf{LangH}_{\mathbb{Z}}$. The only event we have to worry about for now is fail, which comes with a handler **FailH** that assigns fail the precondition False. Accordingly, fail can never be called in a verified program. (We will see in §3.3 how exactly handlers capture event specifications.) We can thus pick $\mathbf{LangH}_{\mathbb{Z}} \coloneqq \mathbf{FailH}$.

The rules in Figure 2 for our weakest precondition wp are now easily derived from the rules in Figure 3 for the underlying ITree weakest precondition wpi. We consider two examples.

Proof of WpIfFalse. By definition of $[\![\_]\!]$ and monadic laws, we have $[\![\texttt{if 0 then } e_1 \texttt{ else } e_2]\!] \approx [\![e_2]\!]$. Using WpiEutt, this reduces our goal to $\mathsf{wp}\ e_2\ \{\Phi\}$, and we are done immediately. □

Proof of WpBindPlusL. We have $[\![e_1 + e_2]\!] \approx v_1 \leftarrow [\![e_1]\!]; k(v_1)$ for k a notational shorthand for the continuation. Using WpiEutt and WpiBind, our goal thus turns into $\mathsf{wp}\ e_1\ \{v_1.\ \mathsf{wp}\ k(v_1)\ \{\Phi\}\}$. By monadic laws, $[\![v_1 + e_2]\!] \approx v_1' \leftarrow \mathsf{Ret}(v_1); k(v_1') \approx k(v_1)$, and we are done by WpiEutt. □

These proofs demonstrate how the language-agnostic rules for wpi greatly simplify the typically tedious task of establishing proof rules for every single language construct.

2.2 2nd effect: mutable state ($\lambda_{\mathbb{Z},!}$)

To demonstrate that we can obtain a language and program logic by composing reusable pieces, we extend our language with a higher-order heap:

$$v \in \mathsf{val} ::= \cdots \mid \ell \quad (\ell \in \mathbb{N}) \qquad\qquad e \in \mathsf{expr} ::= \cdots \mid \texttt{ref}(e) \mid\ !\,e \mid e_1 \leftarrow e_2$$

Here, ℓ is a heap location. The term $\texttt{ref}(e)$ allocates a heap cell with content e, $!(e)$ loads the contents at heap location e, and $e_1 \leftarrow e_2$ stores e_2 at heap location e_1.

In the setting of operational semantics, making such an extension to the language would require invasive surgery to the semantics: even the type of the stepping relation changes because it has to thread through the global state. However, as we shall see, with ITree-based semantics and program logics, we do not have to labor hard for an extension like this.

In our setting, this extension is provided via the $\mathbf{HeapE}_V$ event type for a value type V, which admits operations

(1) $\mathsf{alloc} : V \to \mathbf{itree}\ \mathbf{HeapE}_V\ \mathbb{N}$ which allocates a new heap cell,
(2) $\mathsf{load} : \mathbb{N} \to \mathbf{itree}\ \mathbf{HeapE}_V\ (\mathsf{option}\ V)$ which loads the value in a heap cell if any, and
(3) $\mathsf{store} : \mathbb{N} \to V \to \mathbf{itree}\ \mathbf{HeapE}_V\ (\mathsf{option}\ V)$ which stores a value in a heap cell and returns the old value if any.

WpRef
$$\mathsf{wp}\ \mathtt{ref}(v)\ \{v'.\ \exists \ell.\ v' = \ell * \ell \mapsto v\}$$

WpLoad
$$\frac{\ell \mapsto v}{\mathsf{wp}\ !\ell\ \{v'.\ v = v' * \ell \mapsto v\}}$$

WpStore
$$\frac{\ell \mapsto v}{\mathsf{wp}\ \ell \leftarrow v'\ \{w.\ w = v * \ell \mapsto v'\}}$$

Fig. 4. Program logic for $\lambda_{\mathbb{Z},!}$.

WpiAlloc
$$\mathsf{wpi}_{\mathbf{HeapH}_V}\ \mathsf{alloc}(v)\ \{\ell.\ \ell \mapsto v\}$$

WpiLoad
$$\frac{\ell \mapsto v}{\mathsf{wpi}_{\mathbf{HeapH}_V}\ \mathsf{load}(\ell)\ \{v'.\ v = v' * \ell \mapsto v\}}$$

WpiStore
$$\frac{\ell \mapsto v}{\mathsf{wpi}_{\mathbf{HeapH}_V}\ \mathsf{store}(\ell, v')\ \{w.\ w = v * \ell \mapsto v'\}}$$

Fig. 5. Proof rules for $\mathsf{wpi}_{\mathbf{HeapH}_V}$.

The event type for $\lambda_{\mathbb{Z},!}$ is defined as $\mathbf{LangE}_{\mathbb{Z},!} := \mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{val}}$, using the sum operator ($\oplus$) on event types. (We will use blue color to indicate changes to previous definitions.)

We extend the denotation $[\![e]\!] : \mathbf{itree}\ \mathbf{LangE}_{\mathbb{Z},!}\ \mathsf{val}$ to account for the new operations:

$$\begin{aligned}
[\![\mathtt{ref}(e)]\!] &:= v \leftarrow [\![e]\!]; \ell \leftarrow \mathsf{alloc}(v); \mathsf{Ret}(\ell)\\
[\![!\,e]\!] &:= v \leftarrow [\![e]\!]; \ell \leftarrow \mathsf{to_loc}(v); w \leftarrow \mathsf{load}(\ell); \mathsf{unwrap}(w)\\
[\![e_1 \leftarrow e_2]\!] &:= v_1 \leftarrow [\![e_1]\!]; v_2 \leftarrow [\![e_2]\!]; \ell \leftarrow \mathsf{to_loc}(v_1); w \leftarrow \mathsf{store}(\ell, v_2); \mathsf{unwrap}(w)\\
\mathsf{to_loc}(v) &:= \mathsf{match}\ v\ \mathsf{with}\ \ell \implies \mathsf{Ret}(\ell) \mid _ \implies \mathsf{fail}\ \mathsf{end}\\
\mathsf{unwrap}(w) &:= \mathsf{match}\ w\ \mathsf{with}\ \mathsf{some}(v) \implies \mathsf{Ret}(v) \mid \mathsf{none} \implies \mathsf{fail}\ \mathsf{end}
\end{aligned}$$

Program logic. Our library also provides a handler $\mathbf{HeapH}_V$ for $\mathbf{HeapE}_V$. To obtain a program logic for the extended language, we can combine the handler for **FailE** and for $\mathbf{HeapE}_{\mathsf{val}}$ into a handler $\mathbf{LangH}_{\mathbb{Z},!} := \mathbf{FailH} \oplus \mathbf{HeapH}_{\mathsf{val}}$, and update wp to use this new handler.

The extended logic continues to satisfy all the rules in Figure 2—all proofs continue to proceed as before. In addition, we obtain the rules for the heap primitives as shown in Figure 4. These rules are direct consequences of the rules for the **HeapE** operations displayed in Figure 5 (which lift to $\mathsf{wpi}_{\mathbf{LangH}_{\mathbb{Z},!}}$ as we will see in §3.3). Some technicalities are omitted from the latter rules (consult §3.4).

2.3 3rd effect: non-determinism ($\lambda_{\mathbb{Z},!,\mathbf{pick}}$)

The next extension we consider is non-determinism. Namely, we extend the language with an operation to pick an arbitrary integer:

$$e \in \mathsf{expr} ::= \cdots \mid \mathtt{pick\_int}()$$

This introduces one new effect into the language: **DemonicE**, modeling demonic non-determinism. Accordingly, we define $\mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick}} := \mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE}$.

We now have access to an operation $\mathsf{choice}_A : \mathbf{itree}\ \mathbf{DemonicE}\ A$ which non-deterministically picks out an element from an inhabited type A.[2] We extend the denotation $[\![e]\!]$ to account for choice_A as follows:

$$[\![\mathtt{pick\_int}()]\!] := z \leftarrow \mathsf{choice}_{\mathbb{Z}}; \mathsf{Ret}(z)$$

[2]We will explain the restriction to inhabited types in §2.4.

$$\textsc{WpiDemonic}\ \frac{\forall r \in A.\, \Phi(r)}{\mathsf{wpi}_{\mathbf{DemonicH}}\, \mathsf{choice}_A\, \{\Phi\}} \qquad \textsc{WpPickInt}\ \frac{\forall z.\, \Phi(z)}{\mathsf{wp}\, \mathtt{pick\_int}()\, \{\Phi\}}$$

Fig. 6. Proof rules for $\mathsf{wpi}_{\mathbf{DemonicH}}$ and $\lambda_{\mathbb{Z},!,\mathsf{pick}}$.

Program logic. Our library provides a handler **DemonicH** for **DemonicE**, with a specification for choice_A as shown in Figure 6. With this handler in our quiver, we can take $\mathbf{LangH}_{\mathbb{Z},!,\mathsf{pick}} := \mathbf{FailH} \oplus \mathbf{HeapH}_{\mathsf{val}} \oplus \mathbf{DemonicH}$. Simple as that, we get an extended program logic $\mathsf{wp}\, e\, \{\Phi\}$. Again, the wp rules in Figure 2 and Figure 4 can be carried over. There is also an additional proof rule for choice_A, WpPickInt, which is a direct consequence of WpiDemonic and the general laws for wpi that we have already seen in action above.

2.4 Adequacy

We saw how to build up $\mathsf{wp}\, e\, \{\Phi\}$ in a modular fashion. But what guarantees do we actually get from proving $\mathsf{wp}\, e\, \{\Phi\}$? Intuitively, the answer should be "every execution of e is well-behaved and the returned value satisfies Φ". This is formalized by an *adequacy* (or *soundness*) theorem for the program logic. To makes this precise, we have to define what an *execution* of a $\lambda_{\mathbb{Z},!,\mathsf{pick}}$ program is and when it is *well-behaved*. For our example language, "well-behaved" will mean "terminates and does not reach fail". In §4.1, we will also explain how our approach can in fact deal with partial correctness where non-terminating programs are also considered "well-behaved".

Execution via relational interpretation. A large part of what makes up an "execution" is already defined by $[\![e]\!]$, expressed as a shallow embedding in the meta-logic. The only part that still concerns us is what to do with events, for which our denotation ascribes no computational meaning.

Previous work on ITrees used the notion of an *interpretation* to give meaning to events. These interpretations are based on the idea of an effect handler turning events into monadic operations, and then "lifting" that transformation to an entire ITree.

Using this approach, one can obtain an *interpretation function* for **FailE** events:

$$f_{\mathbf{FailE}} : \mathbf{itree}\ (\mathbf{FailE} \oplus E)\ R \to \mathbf{itree}\ E\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\})$$

$f_{\mathbf{FailE}}$ transforms an ITree by removing the **FailE** events and returning $\bot_{\mathsf{fail}}$ whenever fail is encountered. By applying this function to $[\![e]\!]$, we obtain $t_1 : \mathbf{itree}\ (\mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE})\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\})$.

Next, we can interpret $\mathbf{HeapE}_V$ events by threading the state through the computation, starting at the empty heap. This defines for any V, E, R an interpretation function with the following signature:

$$f_{\mathbf{HeapE}} : \mathbf{itree}\ (\mathbf{HeapE}_V \oplus E)\ R \to \mathbf{itree}\ E\ R$$

(To simplify the signature, this interpretation function discards the final state.) Applying this to t_1, we thus obtain $t_2 : \mathbf{itree}\ \mathbf{DemonicE}\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\})$.

This leaves us with executing the **DemonicE** events. Interpreting non-determinism with an interpretation *function* fails to capture that there is not just a single way to execute choice_A: the entire point is that there is a possible execution for each $a \in A$! This is where we use the concept of a propositional interpretation [42]. Instead of a function, we give an *interpretation relation* to characterize possible executions:

$$\downarrow_{\mathbf{DemonicE}} : \mathbf{itree}\ (\mathbf{DemonicE} \oplus E)\ R \to \mathbf{itree}\ E\ R \to \mathit{Prop}$$

Thus, for t_2, we get a set of possible interpretations $t' : \mathbf{itree}\ \emptyset\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\})$.

At this point, there are no more events left: t' is either an infinite loop, or it terminates with some value in $\mathsf{val} \cup \{\bot_{\mathsf{fail}}\}$. We can hence say that t' is a possible execution of the original ITree $[\![e]\!]$ and therefore of e.

For notational consistency, we view the interpretation functions $f_{\mathbf{FailE}}$ and $f_{\mathbf{HeapE}}$ as relations as well (*i.e.*, , $t \downarrow_E t' \iff t' = f_E(t)$). This lets us define the executions of t : **itree** $\mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick}}\ R$ as the set of t' : **itree** $\emptyset$ $(R \cup \{\bot_{\mathsf{fail}}\})$ that satisfy the following composite interpretation relation:

$$t \downarrow_{\mathbb{Z},!,\mathsf{pick}} t' := \exists t_1, t_2.\ t \downarrow_{\mathbf{FailE}} t_1 \downarrow_{\mathbf{HeapE}} t_2 \downarrow_{\mathbf{DemonicE}} t'$$

Proving adequacy effect-by-effect. Having defined our notion of execution, we turn towards what wp has to say about them. As with everything else, we follow a modular approach. We show adequacy of the underlying wpi for one effect type at a time, and then compose those reusable results to obtain adequacy for wp.

The adequacy theorems for the effects we have seen so far are stated in Figure 7. They all take basically the same shape: if $t \downarrow t'$, then wpi t $\{\Phi\}$ implies wpi t' $\{\Phi'\}$, showing that every property provable via wpi is preserved under all possible interpretations. The postcondition Φ' remains entirely unchanged for HeapAdequate and DemonicAdequate, but for FailAdequate we have to adjust the postcondition to say that the program will never fail. Finally, EmptyAdequate says that proving wpi for an ITree with no effects establishes total correctness: the ITree is equivalent to one that immediately returns a return value r that satisfies the postcondition. Note that the conclusion of these rules is still an Iris proposition, but in the case that Φ is a pure, meta-level predicate, we can use the soundness theorem of Iris to obtain a theorem that lives entirely in the meta-logic without having to trust Iris.

We can compose these adequacy theorems to show that each of the 3 stages (t_1, t_2, t') of an execution $[\![e]\!] \downarrow_{\mathbb{Z},!,\mathsf{pick}} t'$ preserves the weakest precondition:

$$\begin{array}{ll}
[\![e]\!] : \mathbf{itree}\ (\mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE})\ \mathsf{val} & \mathsf{wpi}\ [\![e]\!]\ \{\Phi\} \\
\big\downarrow_{\mathbf{FailE}} & \big| \\
t_1 : \mathbf{itree}\ (\mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE})\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\}) & \mathsf{wpi}\ t_1\ \{v.\ v \neq \bot_{\mathsf{fail}} * \Phi(v)\} \\
\big\downarrow_{\mathbf{HeapE}} & \big| \\
t_2 : \mathbf{itree}\ \mathbf{DemonicE}\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\}) & \mathsf{wpi}\ t_2\ \{v.\ v \neq \bot_{\mathsf{fail}} * \Phi(v)\} \\
\big\downarrow_{\mathbf{DemonicE}} & \big| \\
t' \approx \mathsf{Ret}(v) : \mathbf{itree}\ \emptyset\ (\mathsf{val} \cup \{\bot_{\mathsf{fail}}\}) & v \neq \bot_{\mathsf{fail}} * \Phi(v)
\end{array}$$

This chain is summarized by the adequacy theorem LangAdequate for our program logic, formalizing the intuitive reading of wp e $\{\Phi\}$ from the beginning of this subsection.

Note that the proof is entirely compositional and factors into intermediate stages, each focusing on one effect at a time. Just as how we build up the program logic from smaller building blocks, this confers an advantage of reusability: if one wants to derive a program logic for a language with more kinds of effects, one can define these new effects and prove adequacy theorems for them, and then use these together with the reusable components our library provides without having to monolithically reprove the entire logic to be adequate.

Interpreter soundness proof. As already mentioned, ITrees come with a concept of interpretation functions that make events executable. For non-deterministic choice, there is more than one possible execution, and thus we considered not an interpretation function but an interpretation relation. However, it is still possible to define an interpretation function $f_{\mathbf{DemonicE}}$ that computes

$$\textsc{FailAdequate}\ \frac{t \downarrow_{\mathbf{FailE}} t' \qquad \mathsf{wpi}_{\mathbf{FailH} \oplus H}\, t\, \{\Phi\}}{\mathsf{wpi}_H\, t'\, \{v.\, v \neq \bot_{\mathsf{fail}} * \Phi(v)\}} \qquad \textsc{DemonicAdequate}\ \frac{t \downarrow_{\mathbf{DemonicE}} t' \qquad \mathsf{wpi}_{\mathbf{DemonicH} \oplus H}\, t\, \{\Phi\}}{\mathsf{wpi}_H\, t'\, \{\Phi\}}$$

$$\textsc{HeapAdequate}\ \frac{t \downarrow_{\mathbf{HeapE}} t' \qquad \mathsf{wpi}_{\mathbf{HeapH}_V \oplus H}\, t\, \{\Phi\}}{\mathsf{wpi}_H\, t'\, \{\Phi\}} \qquad \textsc{EmptyAdequate}\ \frac{\mathsf{wpi}_{\emptyset}\, t\, \{\Phi\}}{\exists r.\, t \approx \mathsf{Ret}(r) * \Phi(r)} \qquad \textsc{LangAdequate}\ \frac{[\![e]\!] \downarrow_{\mathbb{Z},!,\mathsf{pick}} t' \qquad \mathsf{wp}\, e\, \{\Phi\}}{\exists r \neq \bot_{\mathsf{fail}}.\, t' \approx \mathsf{Ret}(r) * \Phi(r)}$$

Fig. 7. A selection of adequacy theorems.

a legal instantiation of demonic choice: $t \downarrow_{\mathbf{DemonicE}} f_{\mathbf{DemonicE}}(t)$. This crucially relies on the constraint that choice_A can only be used for *inhabited* types A. Following standard ITree patterns, we can compose these functions for all our effects into a single end-to-end interpreter $f_{\mathbb{Z},!,\mathsf{pick}}$ that can execute $\lambda_{\mathbb{Z},!,\mathsf{pick}}$ programs. Thanks to the ITree-based formulation of LangAdequate, one can easily show the following soundness property: if $\vdash \mathsf{wp}\, e\, \{\Phi\}$, the interpreter applied to e will terminate in a value $v \neq \bot_{\mathsf{fail}}$. This guarantees not only safety but also termination for our interpreter.

2.5 4th effect: Concurrency ($\lambda_{\mathbb{Z},!,\mathbf{pick},\mathbf{fork}}$)

Having tied up the story of $\lambda_{\mathbb{Z},!,\mathsf{pick}}$ with an adequacy theorem, we consider another language extension. This one cuts a lot deeper than the ones we presented so far: we will add concurrency to our language. The syntax becomes:

$$e \in \mathsf{expr} ::= \cdots \mid \mathsf{spawn}\ \{e\}$$

$\mathsf{spawn}\ \{e\}$ represents spawning a new thread executing e.

Since we are adding a new effect (concurrency), we will need to extend the set of events **LangE**:

$$\mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}} := \mathbf{ConcE} \oplus \mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE}$$

While our language has preemtive concurrency, we model it using *cooporative concurrency*. Specifically, the event type **ConcE** unlocks the following new operations:

(1) $\mathsf{spawn} : \mathbf{itree}\ \mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}\ () \to \mathbf{itree}\ \mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}\ ()$ spawns a new thread executing some ITree.
(2) $\mathsf{yield} : \mathbf{itree}\ \mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}\ ()$ yields control to an arbitrary thread in the thread pool (including possibly the current thread).

With this, we can extend our semantic interpretation with a denotation for spawn (using 0 as a throwaway value since our language does not have a "unit" value):

$$[\![\mathsf{spawn}\ \{e\}]\!] := \mathsf{spawn}([\![e]\!]; \mathsf{Ret}(())); \mathsf{Ret}(0)$$

But this is not enough. We must also augment the denotation of other program terms to insert a yield at any point when control may be passed to another thread. It is necessary to exercise some care in doing so to ensure that we model the preemptive semantics in the intended way:

(1) We want to yield "between" any two computation steps. For instance, when evaluating $!\,\ell + !\,\ell$, it is crucial that we yield in between the two loads.
(2) However, some expressions such as $\ell \leftarrow v$ are *atomic* which means they should execute "in a single step" without interleaving with other threads. We do not want yields in the denotations of such expressions.

$$
\begin{aligned}
\mathsf{yield\_if\_not\_val}(e) &:= \mathsf{match}\ e\ \mathsf{with}\ v \implies \mathsf{Ret}(()) \mid \_ \implies \mathsf{yield}\ \mathsf{end} \\
[\![e]\!]_{\mathsf{yield}} &:= v \leftarrow [\![e]\!]; \mathsf{yield\_if\_not\_val}(e); \mathsf{Ret}(v) \\
[\![e_1 \leftarrow e_2]\!] &:= v_1 \leftarrow [\![e_1]\!]_{\mathsf{yield}}; v_2 \leftarrow [\![e_2]\!]_{\mathsf{yield}}; \ell \leftarrow \mathsf{to\_loc}(v_2); w \leftarrow \mathsf{store}(\ell, v_1); \mathsf{unwrap}(w) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!]_{\mathsf{yield}}; v_2 \leftarrow [\![e_2]\!]_{\mathsf{yield}}; (x, e) \leftarrow \mathsf{to\_lam}(v_1); \\
&\quad \mathsf{yield\_if\_not\_val}(e[v_2/x]); [\![e[v_2/x]]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &:= v_1 \leftarrow [\![e_1]\!]_{\mathsf{yield}}; z_1 \leftarrow \mathsf{to\_int}(v_1); \\
&\quad \text{if } z_1 \neq 0 \text{ then } (\mathsf{yield\_if\_not\_val}(e_2); [\![e_2]\!]) \text{ else } (\mathsf{yield\_if\_not\_val}(e_3); [\![e_3]\!])
\end{aligned}
$$

Fig. 8. Excerpt of the placement of yields.

(3) Expressions such as $e_1(e_2)$ and $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ that evaluate by first transforming to another expression e' should yield before continuing with computing e'. For example, β-reduction may not terminate and we do not want one thread to block the thread pool.

An excerpt of the placement of yields in $[\![\_]\!]$ according to these considerations is displayed in Figure 8. To honor (1), we define a notational shorthand $[\![e]\!]_{\mathsf{yield}}$ that places a yield after the evaluation of e; we use this to evaluate the subexpressions of an expression (except for the branches of an if-expression and the argument to `spawn {_}`, which are not eagerly evaluated). However, to honor (2), we make use of a helper function $\mathsf{yield\_if\_not\_val}(e)$ which exhibits a yield only if the expression e is not a value. Finally, to honor (3), we ensure that denotations $[\![e]\!]$ that end on a recursive instance $[\![e']\!]$ have a $\mathsf{yield\_if\_not\_val}(e')$ before this instance to mark the computational step from e to e'.

Intermezzo: Concurrency in Iris. Before we can introduce the program logic for $\lambda_{\mathbb{Z},!,\mathrm{pick},\mathrm{fork}}$, we have to briefly explain how Iris-based program logics deal with concurrency. As usual for concurrent separation logics [27], the case of disjoint concurrency is handled via a separating conjunction: if the forked-off thread has precondition P, then the parent thread needs to prove $P * Q$ where P is handed to the forked-off thread and only Q remains in the parent thread.

But what if the two threads are sharing state? The answer to that is to use an *invariant*: the logical assertion $\boxed{P}^{\mathcal{N}}$ expresses that P is permanently maintained as an invariant on the shared state. Threads can get access to P *atomically*, for an instant in time, with a proof rule like this:[3]

$$
\frac{P \twoheadrightarrow \mathrm{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{r.\, P * \Phi(r)\} \qquad \boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ is atomic}}{\mathrm{wp}_{\mathcal{E}}\, e\, \{\Phi\}}
$$

Ignoring all the $\mathcal{E}$ and $\mathcal{N}$ for now, this rule expresses that to prove correctness of e, we may instead prove $P \twoheadrightarrow \cdots$, *i.e.,* P is made available as an extra assumption. This proof can temporarily break the invariant, but when e finishes execution, we have to re-establish P, thus ensuring that the invariant is maintained again. Crucially, since e is an atomic expression, no other thread can notice that the invariants was temporarily broken—it always holds between any two atomic steps of the program.

However, atomicity alone is not sufficient to ensure soundness of this rule. The other potential problem is *reentrancy*: if the rule could be used twice on the same invariant, the program would gain access to $P * P$, and that would be unsound. This is where the *mask* $\mathcal{E}$ comes in: every invariant lives in a *namespace* $\mathcal{N}$, and the weakest precondition connective keeps track of which invariants are still available in its mask. To open an invariant, the entire namespace must be in the mask ($\mathcal{N} \subseteq \mathcal{E}$), and it is subsequently removed from the mask (e is verified with the reduced mask $\mathcal{E} \setminus \mathcal{N}$).

[3]We ignore slight technicalities that have to do with either later modalities or timeless propositions; see [18].

$$\textsc{WpiSpawn}\ \frac{\mathsf{wpi}_{\mathbf{ConcH}\oplus H;\top}\, t\, \{\mathsf{True}\} * \Phi(())}{\mathsf{wpi}_{\mathbf{ConcH}\oplus H;\mathcal{E}}\, \mathsf{spawn}(t)\, \{\Phi\}} \qquad \textsc{WpiYield}\ \frac{\Phi(())}{\mathsf{wpi}_{\mathbf{ConcH};\top}\, \mathsf{yield}\, \{\Phi\}}$$

Fig. 9. Weakest precondition proof rules for ITrees with concurrency.

Program logic. With that out of the way, we get back to $\lambda_{\mathbb{Z},!,\mathrm{pick},\mathrm{fork}}$. The program logic for our concurrent language is defined largely as before, with a new handler **ConcH** for the **ConcE** events—and with a mask, to handle opening and closing of invariants:

$$\mathbf{LangH}_{\mathbb{Z},!,\mathrm{pick},\mathrm{fork}} := \mathbf{ConcH} \oplus \mathbf{FailH} \oplus \mathbf{HeapH}_{\mathrm{val}} \oplus \mathbf{DemonicH}$$
$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_{\mathbb{Z},!,\mathrm{pick},\mathrm{fork}};\mathcal{E}}\, [\![e]\!]\, \{\Phi\}$$

Our ITree weakest precondition connective does in fact also support masks, we just omitted them until now to make the presentation easier to follow. All the wpi proof rules shown so far implicitly use an arbitrary mask (but fix the same mask for all wpi within a rule).[4]

Our library for **ConcH** provides the rules in Figure 9 for spawn and yield. Both of these rules are more subtle than meets the eye. Unlike other rules stated so far, WpiSpawn is stated over an extended handler $\mathbf{ConcH} \oplus H$ to allow the spawned thread t to also exhibit events that are not related to concurrency. The separating conjunction in the premise expresses that the parent thread needs to split its resources in two *disjoint* parts: one is passed to the new thread t, the other is used to prove the postcondition Φ. Remember that after using "bind"-like rules, Φ will typically itself be a weakest precondition, capturing the verification condition for the continuation in the parent thread. That way, this separating conjunction exactly captures that parent thread and the child thread may only access disjoint state. But of course, this happens in the context of Iris, so one can use invariants to get around that. This is where the other rule comes in: WpiYield only holds for the full mask $\top$. This ensures that all invariants are closed, thus ensuring that whenever execution switches from one thread to another, the new thread can rely on all invariants being satisfied.

Furthermore, wpi satisfies the following rule for opening invariants:

$$\textsc{WpiInvOpen}\ \frac{P \mathrel{-\!\!*} \mathsf{wpi}_{H;\mathcal{E}\setminus\mathcal{N}}\, t\, \{r.\, P * \Phi(r)\} \qquad \boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\mathsf{wpi}_{H;\mathcal{E}}\, t\, \{\Phi\}}$$

The attentive reader may notice the lack of an atomicity side-condition, which could seem like it would render our rule unsound. This is, however, not the case. Yes, we can in fact open invariants around arbitrary code t, even when t exhibits yields. But in that case, we will not be able to show the wpi $t\, \{\cdots\}$ in the assumption because, as we saw, we can only step over yield using WpiYield if we are at full mask $\top$.

After replacing each wp by $\mathsf{wp}_{\mathcal{E}}$, the extended logic continues to satisfy the rules seen so far: the rules in Figure 2, the rules in Figure 4, and the rule WpPickInt. Additionally, it satisfies a number of new proof rules, some of which are shown in Figure 10. Most rules, such as WpPlus and WpLoad, support an arbitrary mask. However, the "bind" rules (*e.g.*, WpBindStoreL and WpBindStoreR) as well as the rule for β-reduction (WpApp) and reduction of if-expressions (WpIfTrue and WpIfFalse) require all invariants to be closed. The reason for this is that these operations contain a yield, so WpiYield forces the mask to be $\top$.

[4]For EmptyAdequate and LangAdequate, we add an Iris *update modality* in the conclusion that carries the mask.

$$\textsc{WpSpawn}\ \frac{\mathsf{wp}_\top\, e\, \{\mathsf{True}\}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{spawn}\ \{e\}\ \{v.\, v = \mathsf{unit}\}} \qquad \textsc{WpInvOpen}\ \frac{P \twoheadrightarrow \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{r.\, P * \Phi(r)\} \qquad \boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

$$\textsc{WpBindStoreL}\ \frac{\mathsf{wp}_\top\, e_1\, \{v_1.\, \mathsf{wp}_\top\, v_1 \leftarrow e_2\, \{\Phi\}\}}{\mathsf{wp}_\top\, e_1 \leftarrow e_2\, \{\Phi\}} \qquad \textsc{WpBindStoreR}\ \frac{\mathsf{wp}_\top\, e_2\, \{v_2.\, \mathsf{wp}_\top\, v_1 \leftarrow v_2\, \{\Phi\}\}}{\mathsf{wp}_\top\, v_1 \leftarrow e_2\, \{\Phi\}}$$

Fig. 10. Excerpt of the program logic for the $\lambda_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}$.

In other words, while we can apply WpInvOpen around expressions like $\ell_1 \leftarrow v_1; \ell_2 \leftarrow v_2$, apparently treating the clearly non-atomic sequence of two stores as "atomic", we cannot complete that proof because further down, we would have to apply WpApp to reduce away the semicolon (which desugars to a λ-abstraction in the usual way). Instead of the typical Iris approach of ensuring atomicity up-front via a dedicated side-condition in WpInvOpen, we ensure atomicity "semantically" by making it impossible to complete the proof when an invariant has been opened around an operation that yields.

Adequacy. Concurrency fits into the adequacy story from §2.4 as yet another reusable component. Given an ITree t : **itree** (**ConcE** $\oplus$ E) R, we specify what are the valid executions (or *interleavings*) t' : **itree** E R by means of an interpretation relation:[5]

$$\downarrow_{\mathbf{ConcE}} : \mathbf{itree}\ (\mathbf{ConcE} \oplus E)\ R \to \mathbf{itree}\ E\ R \to \mathit{Prop}$$

This satisfies an adequacy theorem in the same style as before:

$$\textsc{ConcAdequate}\ \frac{t \downarrow_{\mathbf{ConcE}} t' \qquad \mathsf{wpi}_{\mathbf{ConcH}\oplus H;\top}\, t\, \{\Phi\}}{\mathsf{wpi}_{H;\top}\, t'\, \{\Phi\}}$$

The new interpretation relation for our language uses a composite relation:

$$\downarrow_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}} : \mathbf{itree}\ \mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}\ R \to \mathbf{itree}\ \emptyset\ (R \cup \{\bot_{\mathsf{fail}}\}) \to \mathit{Prop}$$
$$t \downarrow_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}} t' := \exists t_1, t_2, t_3.\ t \downarrow_{\mathbf{ConcE}} t_1 \downarrow_{\mathbf{FailE}} t_2 \downarrow_{\mathbf{HeapE}} t_3 \downarrow_{\mathbf{DemonicE}} t'$$

Composing ConcAdequate with the adequacy theorems in Figure 7, we obtain adequacy for our language with concurrency:

$$\textsc{ExampleAdequate}\ \frac{[\![e]\!] \downarrow_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}} t' \qquad \mathsf{wp}_\top\, e\, \{\Phi\}}{\exists r \neq \bot_{\mathsf{fail}}.\ t' \approx \mathsf{Ret}(r) * \Phi(r)}$$

Although the order of the other events did not matter, it is important that we put **ConcE** in the beginning of $\mathbf{LangE}_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}$. Indeed, **ConcE** is a very special kind of event whose semantics does not commute with, for example, the semantics of **HeapE**: if we interpreted **HeapE** before **ConcE** (that is, $t \downarrow_{\mathbf{HeapE}} t_1 \downarrow_{\mathbf{ConcE}} t_2 \cdots$), each thread would have its own independent copy of the heap. To have the right interaction with other events, **ConcE** must always be interpreted first.

[5]This omits a technical complication related to all threads terminating without any of them returning a value.

Interpreter. Using a round-robin scheduler, we can define an interpretation function

$$f_{\mathbf{ConcE}} : \mathbf{itree}\ (\mathbf{ConcE} \oplus E)\ R \to \mathbf{itree}\ E\ R$$

that instantiates the relation $\downarrow_{\mathbf{ConcE}}$ in the sense that $\forall t.\ t \downarrow_{\mathbf{ConcE}} f_{\mathbf{ConcE}}(t)$. As before, this can be composed to an end-to-end interpreter for $\lambda_{\mathbb{Z},!,\mathsf{pick},\mathsf{fork}}$ and a soundness result showing that if one proves $\mathsf{wp}_\top\ e\ \{\Phi\}$, the interpreter applied to e will terminate in a value $v \neq \bot_{\mathsf{fail}}$.

2.6 5th effect: Angelic choice ($\lambda_{\mathbb{Z},!,\mathbf{pick},\mathbf{fork},\mathbf{ang}}$)

Let us consider one final extension to the language, angelic choice:

$$e \in \mathsf{expr} ::= \cdots \mid \texttt{angelic\_pick\_int()}$$

`angelic_pick_int()` behaves dually to the `pick_int()` expression introduced in §2.3: when verifying a program with `angelic_pick_int()`, we (the verifier) get to pick the value of the choice (unlike `pick_int()` where we have to handle all possible choices). The program is correct if there exists any way to make a choice that leads to the desired outcome. An operation like `angelic_pick_int()` is not commonly found in programming languages and cannot, in general, be compiled to executable machine code. However, angelic choice does have a wide variety of use-cases such as modelling partial programs [3] and concurrency [14, 9], reasoning about interaction with external code [31, 13], and encoding concise specifications [10, 34].

Supporting angelic choice in our weakest precondition for ITrees is straightforward: we introduce an event type **AngelicE** that provides an operation $\mathsf{angelic\_choice}_A : \mathbf{itree}\ \mathbf{AngelicE}\ A$ with corresponding handler **AngelicH** such that we obtain the following rules:

$$\textsc{WpiAngelic}\ \frac{\exists r.\ \Phi(r)}{\mathsf{wpi}_{\mathbf{AngelicH}}\ \mathsf{angelic\_choice}_A\ \{\Phi\}} \qquad \textsc{WpAngelicPickInt}\ \frac{\exists z.\ \Phi(z)}{\mathsf{wp}\ \texttt{angelic\_pick\_int()}\ \{\Phi\}}$$

However, a problem arises when we try to apply the adequacy approach from §2.4 to **AngelicE**. So far all effects were *computational* in the sense that the executions of programs could be described by interpretation relations with corresponding interpretation functions. But angelic choice does not fit this pattern: we cannot remove angelic choice by interpretation since the witness of the angelic choice is not chosen by the interpretation, but during verification. (Another example where the computational approach to adequacy from §2.4 turned out to limit expressivity is the fact that we restricted the demonic choice_A to *inhabited* types, which excludes the option of encoding "non-behavior" via empty demonic non-determinism.)

To equip our framework with the ability to handle such non-computational effects, we introduce a novel notion of "execution" for ITrees that turns an ITree into a state machine in a modular way, side-stepping event interpretation altogether. We then prove adequacy of wpi w.r.t. that state machine. While supporting more kinds of effects, this approach loses the connection to the typical concept of event interpretation in ITrees, and in particular it does not give rise of a soundness proof relating wpi to an interpreter.

Turning ITrees into state machines. Concretely, for an ITree with events E we pick a type of states Σ and construct a state machine given by the following multi-step multi-relation:

$$(\Downarrow) : \mathbf{itree}\ E\ R \to \Sigma \to (\mathbf{itree}\ E\ R \to \Sigma \to \mathsf{Prop}) \to \mathsf{Prop}$$

The relation $(t, \sigma) \Downarrow T$ behaves like a normal multi-step execution relation, except that instead of stepping to a single final ITree and state, it steps to a set T of final ITrees and states (represented by a predicate). Note that this is not quite the typical "big-step" relation in the sense that the final ITree does not have to be a value—the relation holds as long as we reach a state satisfying T at some point

during the execution. The meaning of $(t, \sigma) \Downarrow T$ is that t with initial state σ can reach a state in T, given the right angelic choices. Conversely, when both $(t, \sigma) \Downarrow T_1$ and $(t, \sigma) \Downarrow T_2$ can be derived, this indicates the possibility of demonic choice between those (sets of) outcomes. This representation of state machines with both kinds of non-determinism follows prior work [28, 31, 13, 7] (but note that some prior work swaps the choice of how angelic and demonic choice are represented).

We define $\Downarrow$ modularly by defining a step relation $\rightsquigarrow$ for every operation provided by the event type E. This relation shows how the operation takes a "small" step to its result. For example, the $\rightsquigarrow$ relations for the **AngelicE**, **DemonicE**, and **HeapE**$_V$ are given as follows:

$$\frac{\exists x.\, T(x, \sigma)}{(\mathsf{choice}_A, \sigma) \rightsquigarrow T} \qquad \frac{\forall x.\, T(x, \sigma)}{(\mathsf{angelic\_choice}_A, \sigma) \rightsquigarrow T} \qquad \frac{T(\sigma[\ell], \sigma)}{(\mathsf{load}(\ell), \sigma) \rightsquigarrow T} \qquad \frac{T(\sigma[\ell], \sigma[\ell \mapsto v])}{(\mathsf{store}(\ell, v), \sigma) \rightsquigarrow T}$$

Note how these rules match small-step rules for a state machine: to construct an execution with demonic choice, we need to provide an instantiation of the choice. The step to T is derivable if there *exists* a choice that satisfies T. Angelic choice behaves dually; *all* choices must be in T. Operations like load and store (from §2.2) use the state σ to read resp. write the value for the given location.

We can define $\rightsquigarrow$ for $E_1 \oplus E_2$ from $\rightsquigarrow$ for E_1 and E_2 by building the product of states and using the step relation corresponding to the event. Overall, this allows us to automatically obtain $\rightsquigarrow$ for a combined event type like **LangE**.

With $\rightsquigarrow$ at hand, we define $\Downarrow$ by coinductively lifting $\rightsquigarrow$ to ITrees, which satisfies these rules:

$$\frac{T(t, \sigma)}{(t, \sigma) \Downarrow T} \qquad \frac{(\epsilon, \sigma) \rightsquigarrow (\lambda x, \sigma'.\, (k(x), \sigma') \Downarrow T)}{(x \leftarrow \epsilon; k(x), \sigma) \Downarrow T}$$

Note how the first rule corresponds to terminating the current execution, but applies any time, not just when t is a Ret.

After defining $\Downarrow$, we need to prove that its handling of events agrees with the handler H used by wpi_H. We encode this as a condition $\mathsf{sound}(H, I)$ where I is a invariant on the state Σ. (The definition of $\mathsf{sound}(H, I)$ can be found in the accompanying Coq development.) Note that this condition can be proven once and forall for each handler since it is independent of the verified ITree. We prove $\mathsf{sound}(H, I)$ for all handlers presented in this paper and show that it can be lifted along $E_1 \oplus E_2$. (We even support concurrency via **ConcE** using an extended version of $\Downarrow$.) We prove the following adequacy theorem for wpi_H:[6]

$$\textsc{StateMachineAdequate}\ \frac{\mathsf{sound}(H, I) \qquad (t, \sigma) \Downarrow T \qquad I(\sigma) * \mathsf{wpi}_H\, t\, \{\Phi\}}{\exists t', \sigma'.\, T(t', \sigma') * I(\sigma') * \mathsf{wpi}_H\, t'\, \{\Phi\}}$$

This theorem states that, for a sound handler H, we can "step in" wpi_H along a $\Downarrow$ execution after proving I for its initial state. We obtain a result (t', σ') in the set of final states T of the exection. The invariant I holds for the final state σ' and the "remaining" ITree t' satisfies the weakest precondition wpi_H. In particular, if T ensures that t' is equal to $\mathsf{Ret}(x)$, we obtain $\Phi(x)$. Note that this holds for every possible T (demonically), but only for one (t', σ') in T (angelically).

The fact that $\Downarrow$ is coinductively defined makes this theorem stronger than if it were inductively defined. In particular, the premise can be established without proving that t terminates. As a consequence, this adequacy can even be used to prove termination by choosing a suitable T indicating that the original t terminates.

Overall, we obtain another approach to defining the concept of "executing" an ITree and a corresponding adequacy theorem, both of which are built compositionally by combining reusable pieces for individual effects.

[6] We omit masks from this theorem to avoid clutter.

3 Weakest preconditions for ITrees and Logical Effect Handlers

We have seen the high-level idea of how a language-specific program logic can be defined in terms of a general weakest precondition connective for ITrees, wpi, alongside a menu of reusable effect libraries. We have also seen some examples of such effect libraries. In this section, we will show how wpi and these effect libraries are defined. We start with some background on ITrees (§3.1) and a first version of the definition of wpi (§3.2). We introduce show how to define logical effect handlers with the simple examples of **FailH** and **DemonicH** (§3.3), and then discuss the more complicated handler **HeapH** (§3.4). The final subsection discusses **ConcH** and the associated interpretation relation, alongside the final version of wpi (§3.5).

3.1 Background: ITrees

ITrees [41] are a general domain for representing effectful computations. The ITree type **itree** $E\ R$ is parameterized by a return type R : **Type** and an event type E : **Type** $\to$ **Type**. $E\ A$ should be understood as the type of events with answer type A. ITrees are coinductively defined as

$$t \in \mathbf{itree}\ E\ R ::=_{\text{coind}} \ \mathsf{Ret}(r : R) \mid \mathsf{Tau}(t : \mathbf{itree}\ E\ R) \mid \mathsf{Vis}_A(\epsilon : E\ A, k : A \to \mathbf{itree}\ E\ R)$$

where

(1) $\mathsf{Ret}(r)$ represents just returning value r,
(2) $\mathsf{Tau}(t)$ represents taking a silent step and continuing the computation t, and
(3) $\mathsf{Vis}_A(\epsilon, k)$ represents emitting an event ϵ, receiving answer $a : A$, and then continuing with $k(a)$. (We shall often elide A and write just $\mathsf{Vis}(\epsilon, a)$.)

Tau steps do not represent a visible action in the program, and as such, it is often desirable to ignore them. However, infinite sequences of Tau steps are still relevant since they represent diverging computations. To this end, ITrees come with the bisimulation $\approx$, known as "equivalence up to (finitely many) Taus", which allows removing/introducing finitely many Taus on either side.

itree $E\ R$ is a monad in R. More specifically, the monadic laws hold up to $\approx$. The monadic bind enables us to write ITrees in the style of sequential code: $x_1 \leftarrow \cdots ; x_2 \leftarrow \cdots ; \cdots$.

We implicitly coerce events ϵ to ITrees that emit this event, *i.e.,* if we write ϵ for an event in a context where an ITree is expected, we implicity desugar it to $\mathsf{Vis}(\epsilon, (\lambda a.\ \mathsf{Ret}(a)))$. (This construction is called trigger in the ITree library.)

3.2 Defining wpi_H (v0.1)

We can now define the weakest precondition $\mathsf{wpi}_{H;\mathcal{E}}\ t\ \{\Phi\}$ for ITrees at the core of our theory. It is defined in terms of a preliminary connective that does not carry a mask:[7]

$$\mathsf{wpi}_H\ t\ \{\Phi\} := \Rrightarrow \begin{cases} \Phi(r) & \text{if } t = \mathsf{Ret}(r) \\ \mathsf{wpi}_H\ t'\ \{\Phi\} & \text{if } t = \mathsf{Tau}(t') \\ H_A(\epsilon, (\lambda a.\ \mathsf{wpi}_H\ k(a)\ \{\Phi\})) & \text{if } t = \mathsf{Vis}_A(\epsilon, k) \end{cases}$$

The first two cases are straightforward: $\mathsf{Ret}(r)$ asserts the postcondition Φ and $\mathsf{Tau}(t')$ continues with t'. The more interesting case is the one for events, $\mathsf{Vis}_A(\epsilon, k)$, which is deferred to a *logical effect handler* H. The handler H determines the verification condition for each event ϵ in E in predicate transformer style: if the event ϵ with answer type A has precondition P and postcondition $Q(a)$ for answer a, we would set $H(\epsilon, \Psi) := P * (\forall a : A.\ Q(a) \mathrel{-\!\!*} \Psi(a))$. (We will discuss handlers in more detail in the next subsection. We will also slightly revise wpi in §3.5.)

[7] The Iris update modality $\Rrightarrow$ allows performing updates to the ghost state; it can largely be ignored for now. The modality can also carry a mask to give access to invariants; we follow the convention that if the mask is empty, we leave it away.

All the rules for wpi_H in Figure 3 (on page 5) hold for free, or rather, for cheap: we only need a single property of H. Namely, we require that handlers satisfy *monotonicity*. For all events ϵ with answer type A, and all $\Phi, \Psi : A \to \mathit{iProp}$, the following must hold:

$$(\forall a.\, \Phi(a) \mathrel{-\!\!*} \Psi(a)) \mathrel{-\!\!*} H_A(\epsilon, \Phi) \mathrel{-\!\!*} H_A(\epsilon, \Psi) \qquad \text{(HandlerMono)}$$

This property trivially holds for all handlers presented in this paper.

Thanks to this monotonicity property, the recursive definition of wpi_H is well-formed by taking the least fixpoint. Choosing the least fixpoint as opposed the greatest fixpoint means that (by default) our weakest precondition is *termination sensitive* or *total*, that is, it implies termination of programs. However, we shall see *later* in §4.1 how this definition also subsumes termination insensitive reasoning, thus uniting both total and partial verification in one, common framework.

Masks and invariants. As explained in §2.5, reasoning about concurrent programs in Iris (and by extension in our framework) rests on the concept of invariants to share state across threads, and masks to track which invariants can still be opened in the current thread. To account for this, we equip wpi with a mask $\mathcal{E}$:

$$\mathsf{wpi}_{H;\mathcal{E}}\, t\, \{\Phi\} := {}_{\mathcal{E}}\!\Rrightarrow_{\emptyset} \mathsf{wpi}_H\, t\, \{r.\ {}_{\emptyset}\!\Rrightarrow_{\mathcal{E}} \Phi(r)\}$$

This definition permits all invariants in $\mathcal{E}$ to be opened for the entire computation represented by t: ${}_{\mathcal{E}}\!\Rrightarrow_{\emptyset}$ is a *mask-changing update modality*, which says that starting with mask $\mathcal{E}$, arbitrary invariants can be opened. (Iris masks support framing, so the empty mask does not *force* all invariants to be opened.) As discussed previously in §2.5, this does *not* mean that t is considered a single atomic action; the specification for yield ensures that all invariants are closed at each yield point, thus ensuring sound concurrent reasoning.

We will omit the mask when it is not relevant for the current discussion; in that case the mask is arbitrary but fixed (*i.e.,* it must be the same for all wpi in a rule or theorem statement).

3.3 Logical Effect Handlers: failure, non-deterministic choice

We now have a more in-depth look at handlers H. Handlers codify what one needs to prove when verifying an event, *i.e.,* we have:

$$H(\epsilon, \Phi) \vdash \mathsf{wpi}_H\, \epsilon\, \{\Phi\}$$

(On the right hand side, the event ϵ is implicitly coerced to an ITree as described in §3.1.)

We define a corresponding handler for each event type. For example, for the **FailE** and **DemonicE** events (introduced in §2.1 and §2.3), the handlers take the following shape:

$$\mathbf{FailH}_{\emptyset}(\mathsf{fail}, \Phi) := \mathsf{False} \qquad \mathbf{DemonicH}_A(\mathsf{choice}_A, \Phi) := \forall a.\, \Phi(a)$$

FailE has a single event fail that we want to prove never happens. We can encode this by defining **FailH**(fail, Φ) as False and thus ensuring that we can never prove $\mathsf{wpi}_H\, \mathsf{fail}\, \{\Phi\}$. For demonic choice over a type A (given by the choice_A event), we want to verify that the program is correct *for all* possible choices. We encode this by using a universal quantifier in the handler **DemonicH**. From this, we obtain the rule WpiDemonic in §2.3.

Composing handlers. To define a handler for a programming language with lots of different effects, we compose handlers for the individual effects. If H_1 is a handler for E_1 and H_2 is a handler for E_2, we can define a handler $H_1 \oplus H_2$ for the sum $E_1 \oplus E_2$ in the obvious way.

Subsumption. A program using only a subset of the available effects can be verified in the corresponding fragment of the program logic. Suppose H is a handler for E and H' is a handler for E' such that E' is contained in E. We write $H' \subseteq H$ if $H(\epsilon, \Phi) \dashv\vdash H'(\epsilon, \Phi)$ for every event ϵ in E' and every Φ. This gives rise to the rule:

$$\textsc{WpiSubsume}\ \frac{E' \subseteq E \qquad H' \subseteq H \qquad t : \mathbf{itree}\ E'\ R}{\mathsf{wpi}_H\, t\, \{\Phi\} \dashv\vdash \mathsf{wpi}_{H'}\, t\, \{\Phi\}}$$

The most salient examples are $H_1 \subseteq H_1 \oplus H_2$ and $H_2 \subseteq H_1 \oplus H_2$ for any handlers H_1, H_2. For instance, proof rules from $\mathsf{wpi}_{\mathbf{DemonicH}}$ thus lift to proof rules for $\mathsf{wpi}_{\mathbf{DemonicH} \oplus \cdots}$.

3.4 Handling mutable state

In §2.2, we introduced the $\mathbf{HeapE}_V$ event type for a mutable heap. This event type is in fact itself a derived construction, based on the $\mathbf{StateE}_{\mathcal{S}}$ event type that adds global state of type $\mathcal{S}$. $\mathbf{HeapE}_V$ is then defined via $\mathbf{StateE}_{\mathbb{N} \xrightarrow{\mathsf{fin}} V}$.

The $\mathbf{StateE}_{\mathcal{S}}$ event type is a standard ITree construction. It consists of two events: get, with answer type $\mathcal{S}$, returns the current state; $\mathsf{put}(s)$, with answer type $()$, overwrites the current state.

The handler for $\mathbf{StateE}_{\mathcal{S}}$ follows the usual Iris recipe for dealing with global state: it is parameterized by a *state interpretation* $S : \mathcal{S} \to \mathit{iProp}$ which is used to relate the physical state to Iris's logical state. Intuitively, $S(s)$ says "we own the state and it is currently s".

Based on the state interpretation, we define the state handler as follows:

$$\mathbf{StateH}^S_{\mathcal{S}}(\mathsf{get}, \Phi) := \forall s.\ S(s) \mathrel{-\!\!*} \Rrightarrow (S(s) * \Phi(s))$$
$$\mathbf{StateH}^S_{\mathcal{S}}(\mathsf{put}(s'), \Phi) := \forall s.\ S(s) \mathrel{-\!\!*} \Rrightarrow (S(s') * \Phi(s))$$

The handler for put says that to prove $\mathsf{wpi}_{\mathbf{StateH}^S_{\mathcal{S};\mathcal{E}}}\, \mathsf{put}(s')\, \{\Phi\}$, we can briefly take ownership $S(s)$ of the old state, and then we have to give back ownership $S(s')$ of the new state and establish $\Phi(s)$. (We will get back to the update modality $\Rrightarrow$ shortly.) The handler for get is similar, except that the state interpretation has to be given back unchanged.

Building HeapE on top of StateE. On top of get and put, we can define heap operations:

$$\mathsf{load}(\ell) := \sigma \leftarrow \mathsf{get}; \mathsf{Ret}(\sigma(\ell)) \qquad : \mathbf{itree}\ \mathbf{HeapE}_V\ (\mathsf{option}\ V)$$
$$\mathsf{store}(\ell, v) := \sigma \leftarrow \mathsf{get}(); \mathsf{put}(\sigma[\ell := \mathsf{some}(v)]); \mathsf{Ret}(\sigma(\ell)) \qquad : \mathbf{itree}\ \mathbf{HeapE}_V\ (\mathsf{option}\ V)$$
$$\mathsf{alloc}(v) := \sigma \leftarrow \mathsf{get}; \ell := \mathsf{find\_free}(\sigma); \mathsf{put}(\sigma[\ell := \mathsf{some}(v)]); \mathsf{Ret}(\ell) : \mathbf{itree}\ \mathbf{HeapE}_V\ \mathbb{N}$$

For space reasons, we have to elide derivation of the laws in Figure 5 (on page 6) from the basic laws for state. The high-level summary is that we follow basically the same recipe as the standard Iris program logic [21]: we set up ghost state that tracks the current contents of physical state in S (this is why we need an update modality in the state handler) and use the same ghost state to give meaning to $\ell \mapsto v$. The one technical wrinkle is that we have to put a third view of this ghost state into a shared invariant to permit the proof to "remember" facts about the global state in between invocations of state operations.[8] The use of an invariant gives rise to a technical side-condition, requiring the namespace of this invariant to be in the mask for each heap operation. Up to such venial side-conditions, this construction lets us then derive the desired rules in Figure 5.

The adequacy theorem HeapAdequate in Figure 7 (on page 9) is a special case of a more general adequacy theorem for $\mathbf{StateH}^S_{\mathcal{S}}$, which we omit for lack of space.

[8] For the Iris experts: we use the "authoritative" construction. Its authoritative part supports fractional permissions, so we can put one half in S and one half in a global invariant. The points-to connective is defined, as usual, as a fragment of the same ghost state. The invariant itself is then also made part of S so that it does not have to be explicitly threaded through.

3.5 Handling concurrency, and wpi_H (v1)

Next, we consider the concurrency effect introduced in §2.5. Modeling and reasoning about concurrency is yet another reusable component in our theory. However, as it turns out, concurrency is a sufficiently "different" effect that we have to extend our definition of wpi to support it.

Similar to Choice Trees [6], we encode *cooporative concurrency* in ITrees using an event type **ConcE** with three kinds of events:

(1) fork with answer type {cur, new}. This rather unusual event causes the continuation to be run twice: the current thread continues its execution with the answer cur, and a new thread is created that continues with answer new.
(2) yield with answer type (). This yields control to some running thread (which may be the current thread again).
(3) endthread with answer type $\emptyset$. This safely ends the current thread and yields control to another thread.

From these ingredients, one can define the more familiar operation that spawns a thread running some code t as follows:

$$\mathsf{spawn}(t) \coloneqq x \leftarrow \mathsf{fork}; \mathsf{if}\ x = \mathsf{cur}\ \mathsf{then}\ (\mathsf{yield}; \mathsf{Ret}(()))\ \mathsf{else}\ (t; \mathsf{endthread})$$

The newly created thread runs t and then invokes endthread, thus ensuring that spawn itself only returns once (in the parent thread).

To obtain a program logic, we have to define a handler for these three events. Let us start by considering fork. One candidate definition would be:

$$\mathbf{ConcH}(\mathsf{fork}, \Phi) \coloneqq \Phi(\mathsf{cur}) * \Phi(\mathsf{new})$$

This models multithreading through the separating conjuction, as is usual in concurrent separation logic. However, this definition breaks monotonicity. Note that HandlerMono is stated as a *separation logic* version of monotonicity, *i.e.,* using magic wands. This means the implication from Φ to Ψ can only be used once, but our **ConcH** above would have to used it twice. As a consequence, it is incompatible with fundamental rules such as the frame rule. Intuitively, the problem is that fork returns twice, so its continuation gets duplicated, which is not compatible with the basic premise of separation logic where resources can only be used once.

Making handlers and wpi fit for concurrency. To overcome this, we extend the notion of handlers to account for concurrency. We add one extra parameter Φ_s to handlers: $H_A(e, \Phi, \Phi_s)$. All the existing, sequential handlers will simply ignore this argument. $\Phi_s(a)$ represents the weakest precondition for spawning a new thread executing the continuation for $a : A$. We shall impose the following extended monotonicity condition on handlers:

$$(\forall a.\, \Phi(a) \mathrel{-\!\!*} \Psi(a)) \mathrel{-\!\!*} \Box(\forall a.\, \Phi_s(a) \mathrel{-\!\!*} \Psi_s(a)) \mathrel{-\!\!*} H_A(\epsilon, \Phi, \Phi_s) \mathrel{-\!\!*} H_A(\epsilon, \Psi, \Psi_s)$$

In particular, the implication from Φ_s to Ψ_s is given under Iris' *persistence modality* $\Box$, which means that it can be used multiple times.

We also amend the weakest precondition for ITrees with support for concurrency, arriving at its final, actual definition:

$$\mathsf{wpi}_H\, t\, \{\Phi\} \coloneqq \Rrightarrow \begin{cases} \Phi(r) & \text{if } t = \mathsf{Ret}(r) \\ \mathsf{wpi}_H\, t'\, \{\Phi\} & \text{if } t = \mathsf{Tau}(t') \\ H_A(\epsilon, (\lambda a.\, \mathsf{wpi}_H\, k(a)\, \{\Phi\}), (\lambda a.\, {}_{\top}\Rrightarrow_{\emptyset} \mathsf{wpi}_H\, k(a)\, \{\mathsf{False}\})) & \text{if } t = \mathsf{Vis}_A(\epsilon, k) \end{cases}$$

We use postcondition False for the thread spawning continuation to enforce that new threads never end in a $\mathsf{Ret}(r)$, *i.e.,* only the main thread can return. This is critical to ward off the issues

$$\textsc{ConcIrel}\ \frac{[t] \downarrow^{0}_{\mathbf{ConcE}} t'}{t \downarrow_{\mathbf{ConcE}} t'} \qquad \textsc{ConcIrelRet}\ \mathsf{tp}[i := \mathsf{Ret}(r)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Ret}(r) \qquad \textsc{ConcIrelTau}\ \frac{\mathsf{tp}[i := t] \downarrow^{i}_{\mathbf{ConcE}} t'}{\mathsf{tp}[i := \mathsf{Tau}(t)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Tau}(t')}$$

$$\textsc{ConcIrelVis}\ \frac{\epsilon \notin \mathbf{ConcE} \qquad \forall a.\, \mathsf{tp}[i := k(a)] \downarrow^{i}_{\mathbf{ConcE}} k'(a)}{\mathsf{tp}[i := \mathsf{Vis}(\epsilon, k)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Vis}(\epsilon, k')} \qquad \textsc{ConcIrelYield}\ \frac{\mathsf{tp}[i := k(())] \downarrow^{j}_{\mathbf{ConcE}} t'}{\mathsf{tp}[i := \mathsf{Vis}(\mathsf{yield}, k)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Tau}(t')}$$

$$\textsc{ConcIrelEndthread}\ \frac{\mathsf{delete}(i, \mathsf{tp}) \downarrow^{j}_{\mathbf{ConcE}} t'}{\mathsf{tp}[i := \mathsf{Vis}(\mathsf{endthread}, k)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Tau}(t')} \qquad \textsc{ConcIrelFork}\ \frac{\mathsf{tp}[i := k(\mathsf{cur})] \twoheadrightarrow [k(\mathsf{new})] \downarrow^{i}_{\mathbf{ConcE}} t'}{\mathsf{tp}[i := \mathsf{Vis}(\mathsf{fork}, k)] \downarrow^{i}_{\mathbf{ConcE}} \mathsf{Tau}(t')}$$

Fig. 11. Definition of $\downarrow_{\mathbf{ConcE}}$.

related to fork returning twice. Essentially, this imposes a proof obligation to show that only the original thread can return. spawn's use of endthread ensures that this is the case.

A handler for ConcE. With this out of the way, we can finally define **ConcH**:

$$\begin{aligned}
\mathbf{ConcH}(\mathsf{fork}, \Phi, \Phi_s) &:= \Phi(\mathsf{cur}) * \Phi_s(\mathsf{new}) \\
\mathbf{ConcH}(\mathsf{yield}, \Phi, \Phi_s) &:= {}_{\emptyset}\Rightarrow_{\top}\, {}_{\top}\Rightarrow_{\emptyset} \Phi(()) \\
\mathbf{ConcH}(\mathsf{endthread}, \Phi, \Phi_s) &:= {}_{\emptyset}\Rightarrow_{\top} \mathsf{True}
\end{aligned}$$

Crucially, the handler for fork uses Φ only once and thus satisfies monotonicity. The handler for yield uses a mask-changing update ${}_{\emptyset}\Rightarrow_{\top}$ to force *all* invariants to be closed, and then immediately switches back to the empty mask which lets the invariants be opened again. However, this is enough to ensure that for one instant, all invariants are satisfied, and thus we can soundly switch from one thread to another. The handler for endthread is similar, except that it never returns to the program, so after closing all invariants there is nothing left to be proven. From these handlers, standard Iris reasoning can derive the rules in Figure 9 (on page 11).

An interpretation relation for ConcE. This concludes the setup of the program logic for **ConcE**. The last missing step is to define what it means to *execute* an ITree with **ConcE** events: we need to define the interpretation relation $\downarrow_{\mathbf{ConcE}}$ for **ConcE**.

To this end, we first define a *thread pool evaluation relation* $\mathsf{tp} \downarrow^{i}_{\mathbf{ConcE}} t'$. Here, tp is a list of ITrees representing the currently existing threads, and i is the index of the thread that is currently running. The relation characterizes all the ITrees t' that can arise by interleaving thread executions in an arbitrary way.

Thread pool evaluation is defined coinductively as shown in Figure 11 (we implicitly assume that every index i, j is in-bounds). The key rules are ConcIrelYield, ConcIrelEndthread, and ConcIrelFork, which define what happens when the active thread $\mathsf{tp}(i)$ runs one of the **ConcE** events. On a yield, we pick an arbitrary new thread j with which to continue the execution. The current thread i is updated to reflect that the yield has been executed and returned $()$. We also add a Tau event to t'; this ensures that the coinduction is well-formed. On an endthread, we delete the current thread from the thread pool and continue the execution at some new thread j.[9] And finally, fork updates the current thread i to continue with the $k(\mathsf{cur})$ continuation and adds a new thread

[9]We omit some technical details related to what happens if the *last* thread ends, which is something that can never happen in the languages we consider since the denotations into ITrees never put an endthread into the main thread.

to the thread pool that executes the $k(\mathsf{new})$ continuation. This is the key rule and the source of all the complications we had to deal with above since it duplicates k.

The remaining rules say that the interleaved ITree t' mirrors the behavior of the active thread: ConcIrelTau forwards silent steps, ConcIrelRet terminates execution when the active thread reaches a Ret,[10] and ConcIrelVis forwards visible events. Note how the latter requires the premise to be shown for *all* possible answers a; the "interleaving" of an ITree is not just a single execution but resolves all scheduling questions for all possible answers to uninterpreted events.

With this definition in place, we can tie it all together and prove ConcAdequate. This proof is highly non-trivial due to our use of a least fixpoint in the definition of wpi: as part of the proof, we are showing that the per-thread termination proof that is implicit in the premise carries over to all possible interleavings.

The proof also relies on a technical side-condition omitted in the paper: the handler H for the remaining events needs to be *sequential*, which means that it must be constant in the argument Φ_s. HeapAdequate also carries this side-condition. Aside from **ConcH**, all handlers discussed in this paper are sequential, so in practice, this means that **ConcE** must be interpreted as the first event.

4 Case study: HeapLang

In this section, we demonstrate the applicability of our approach by using it to build a program logic for HeapLang, the default language for program verification in Iris. HeapLang [18, §6.1] is an untyped lambda calculus with an ML-like higher order heap and concurrency. Our goal is to recover the original HeapLang program logic with all its basic rules [18, Figure 13] in a compositional style using the theory developed in earlier sections.

We start by stating the syntax of the language:

$$v \in \mathsf{val} ::= () \mid z \mid \mathtt{true} \mid \mathtt{false} \mid \ell \mid \mathtt{rec}\, f(x) := e \mid \cdots \qquad (z \in \mathbb{Z})$$
$$e \in \mathsf{expr} ::= v \mid x \mid e_1(e_2) \mid \mathtt{spawn}\ \{e\} \mid \mathtt{ref}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{CAS}(e, e_1, e_2) \mid \cdots$$

(Arithmetic operations and the usual operations on pairs and sums are ommited for brevity.)

Following the same pattern as §2, we specify a semantics for HeapLang by denoting expressions into **itree HeapLangE** val where

$$\mathbf{HeapLangE} := \mathbf{ConcE} \oplus \mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{val}} \oplus \mathbf{DemonicE}$$

For most of the expressions, the semantic interpretation $[\![e]\!]$ is defined as for the example language in §2, except that HeapLang uses right-to-left evaluation order and its closures have a binder f for recursive calls. Also, we tweak alloc (see §3.4) to instead pick the free location non-deterministically (thus necessitating **DemonicE** among our events). The main new operation worth talking about is CAS, the atomic compare-and-swap operation, which is denoted as follows:

$$\begin{aligned}[\![\mathsf{CAS}(e, e_1, e_2)]\!] := {} & v_2 \leftarrow [\![e_2]\!]_{\mathsf{yield}}; v_1 \leftarrow [\![e_1]\!]_{\mathsf{yield}}; v \leftarrow [\![e]\!]_{\mathsf{yield}}; \ell \leftarrow \mathsf{to\_loc}(v);\\ & v_?' \leftarrow \mathsf{load}(\ell); v' \leftarrow \mathsf{unwrap}(v_?');\\ & \text{if } v_1 = v' \text{ then } (\mathsf{store}(\ell, v_2); \mathsf{Ret}(\mathtt{true})) \text{ else } \mathsf{Ret}(\mathtt{false})\end{aligned}$$

where we reuse helper functions to_loc, unwrap, and $[\![\_]\!]_{\mathsf{yield}}$ from §2.

To obtain a program logic, we combine the various handlers from §3 to obtain a handler **HeapLangH** for **HeapLangE**. As in §2, we then define $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} := \mathsf{wpi}_{\mathbf{HeapLangH};\mathcal{E}}\, [\![e]\!]\, \{\Phi\}$.

[10] The denotation of a language into ITrees generally ensure that only the main thread ever reaches Ret; all the other threads are ended with endthread.

$\mathrm{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ satisfies the various rules that were discussed in §2. Using the notion of evaluation contexts K [18, §6.1], the bind rules such as WpBindPlusL can be summarized in a single rule:

$$\textsc{HlWpBind}\quad \frac{\mathrm{wp}_{\top}\, e\, \{v.\, \mathrm{wp}_{\top}\, K[v]\, \{\Phi\}\}}{\mathrm{wp}_{\top}\, K[e]\, \{\Phi\}}$$

(The ramifications of the full mask $\top$ were already discussed in §2.5.) This is a consequence of a "semantic bind lemma" which says that the *syntactic* bind $K[e]$ interprets to the *monadic* bind:

Lemma 4.1 (Semantic bind lemma). *If* $K \neq \bullet$ *then* $[\![K[e]]\!] \;\approx\; v \leftarrow [\![e]\!]_{yield};\, [\![K[v]]\!]$.

Reuse in the semantics. The definition of $[\![\_]\!]$ relies on helper functions load and store. We reuse their specifications (Figure 5) when establishing proof rules such as those for CAS:

$$\textsc{HlWpCasSuc}\quad \frac{\ell \mapsto v}{\mathrm{wp}_{\mathcal{E}}\, \mathtt{CAS}(\ell, v, w)\, \{w.\, w = \mathtt{true} * \ell \mapsto w\}} \qquad \textsc{HlWpCasFail}\quad \frac{v \neq v' \qquad \ell \mapsto v}{\mathrm{wp}_{\mathcal{E}}\, \mathtt{CAS}(\ell, v', w)\, \{w.\, w = \mathtt{false} * \ell \mapsto v\}}$$

In the setting of operational semantics, such reuse is generally not possible. One could try to define CAS inside the language instead of having it as a primitive operation:

$$\mathtt{CAS}(x, a, b) := \mathtt{if}\ a = !\,x\ \mathtt{then}\ (x \leftarrow b; \mathtt{true})\ \mathtt{else}\ \mathtt{false}$$

However, this is not equivalent to the intended definition: CAS is supposed to be atomic whereas this definition is not; other threads could take steps between the load and the store. With cooporative concurrency, on the other hand, we can define the helper functions load and store with no yields so that no additional interleavings are introduced. This reuse simplifies both the language semantics and the correctness proof for the program logic.

4.1 Termination-insensitive reasoning

So far, we have only considered a very strong kind of program logic, namely a total weakest precondition that ensures termination. This does not match the usual logic used for HeapLang, which just proves partial correctness.

To match existing HeapLang practice, we also define a partial weakest precondition $\mathrm{wp}^{\triangleright}_{\mathcal{E}}\, e\, \{\Phi\}$ that does not guarantee termination. We can use our existing framework to define this weakest precondition without having to redefine wpi. Iris approaches partial verification by means of the later modality $\triangleright$ which comes with a powerful coinductive reasoning principle: *Löb induction* [18, §5.6]. Typical Iris program logics are set up such that the weakest precondition involves at least one $\triangleright$ per program step, which can be used with Löb induction to derive the usual partial correctness reasoning principle for recursive functions. To achieve the same with our approach, we define an event type **StepE** with a single event, step, with answer type $()$. The handler for **StepE** is then defined as $\mathbf{StepH}(\mathrm{step}, \Phi, \Phi_s) := \triangleright\, \Phi(())$.

We define $\mathbf{HeapLangE}^{\triangleright} := \mathbf{HeapLangE} \oplus \mathbf{StepE}$, and $\mathbf{HeapLangH}^{\triangleright} := \mathbf{HeapLangH} \oplus \mathbf{StepH}$, and finally define $[\![e]\!]^{\triangleright}$ like $[\![e]\!]$ but dredging step events at every point in $[\![e]\!]$ that corresponds to a step in the operational semantics (see the Coq code for the exact placement). The result is a partial program logic $\mathrm{wp}^{\triangleright}_{\mathcal{E}}\, e\, \{\Phi\} := \mathrm{wpi}_{\mathbf{HeapLangH}^{\triangleright};\mathcal{E}}\, [\![e]\!]^{\triangleright}\, \{\Phi\}$ validating the standard HeapLang rules.

In our Coq mechanization, we take this one step further and make **StepH** and wp parametric in whether a later modality is emitted. This allows uniform treatment of partial and total correctness and their proof rules in one, unified framework without duplicated proof effort.

$$\frac{n > 0 \qquad k(()) \downarrow^{n-1}_{\mathbf{StepE}} t'}{\mathsf{Vis}(\mathsf{step}, k) \downarrow^{n}_{\mathbf{StepE}} \mathsf{Tau}(t')} \qquad \mathsf{Vis}(\mathsf{step}, k) \downarrow^{0}_{\mathbf{StepE}} \mathsf{Ret}(\bot_{\mathsf{step\_timeout}}) \qquad \mathsf{Ret}(r) \downarrow^{n}_{\mathbf{StepE}} \mathsf{Ret}(r)$$

$$\frac{t \downarrow^{n}_{\mathbf{StepE}} t'}{\mathsf{Tau}(t) \downarrow^{n}_{\mathbf{StepE}} \mathsf{Tau}(t')} \qquad \frac{\epsilon \notin \mathbf{StepE} \qquad \forall a.\, k(a) \downarrow^{n}_{\mathbf{StepE}} k'(a)}{\mathsf{Vis}(\epsilon, k) \downarrow^{n}_{\mathbf{StepE}} \mathsf{Vis}(\epsilon, k')}$$

StepAdequate

$$\frac{t \downarrow^{n}_{\mathbf{StepE}} t' \qquad \mathsf{wpi}_{\mathbf{StepH} \oplus H; \emptyset}\, t\, \{\Phi\} * £ n}{\mathsf{wpi}_{H;\emptyset}\, t'\, \{x.\, \mathsf{match}\ x\ \mathsf{with}\ \bot_{\mathsf{step\_timeout}} \implies \mathsf{True} \mid r \implies \Phi(r)\}}$$

Fig. 12. Rules defining the coinductive relation $\downarrow^{n}_{\mathbf{StepE}}$, and adequacy for the corresponding handler.

Adequacy for StepH. **StepH** fits into the compositional adequacy story of §2.4 as yet another reusable piece. To describe the semantics of **StepE**, we define an interpretation relation

$$\downarrow^{n}_{\mathbf{StepE}} : \mathbf{itree}\ (\mathbf{StepE} \oplus E)\ R \to \mathbf{itree}\ E\ (R \cup \{\bot_{\mathsf{step\_timeout}}\}) \to \mathit{Prop} \qquad \text{for } n \in \mathbb{N}$$

by coinduction according to the rules in Figure 12. For consistency, we write it as a relation, but it can equivalently be written as a function $f_{\mathbf{StepE}}$. Intuitively, $f_{\mathbf{StepE}}(n, t)$ executes the first n step events in t as no-ops. Once this "fuel" is used up, the next step event terminates program execution by returning $\bot_{\mathsf{step\_timeout}}$.

The adequacy theorem (StepAdequate) turns the wpi of t into a wpi of every partial execution t' of t. It requires later credits $£ n$ (Spies et al. [35]) which provide the right to strip n later modalities. The Iris soundness theorem provides any fixed number of later credits, so this is sufficient to prove that for every n, if the program terminates in n steps, the postcondition holds.

4.2 Comparison to the original HeapLang

We obtain for our program logic nearly the same rules as HeapLang's existing program logic. The only significant difference is the fact that in our logic, invariants can be opened around any block of code, at the expense of the bind rule HlWpBind needing the full mask $\top$; *cf.* §2.5. Furthermore, while we still use the Iris proof mode [22, 20], we have not reimplemented all the additional proof mode integration and automation available in the existing HeapLang implementation. We have also omitted support for the more recent extension of HeapLang introducing prophecy variables [19].

A verified interpreter. As before, we can compose the interpretation functions for the compounding events to obtain an interpreter for HeapLang and an associated correctness proof. Unlike the existing HeapLang interpreter, this does not require a full second, executable definition of the language semantics; we can just reuse the ITree denotation. We also obtain a stronger soundness result for the interpreter, showing in particular that if a program was proven to terminate using the total weakest precondition, then the interpreter eventually terminates.

Correctness of ITree semantics w.r.t. operational semantics. By composing interpretation relations as in §2.4, we obtain an interpretation relation $\downarrow^{n;\sigma}_{\mathbf{HeapLangE}^{\triangleright}}$ which allows us to describe the executions of $[\![e]\!]^{\triangleright}$. To provide assurance that this semantics is meaningful, we prove a result relating it to the more well-established operational semantics of HeapLang [18, Figure 12]. First, we define two notions of "adequacy" for a program w.r.t. a postcondition, one in terms of $\downarrow_{\mathbf{HeapLangE}^{\triangleright}}$ and one in terms of operational semantics:

Definition 4.2. e is *interpretationally adequate* w.r.t. postcondition $\phi : R \to \mathit{Prop}$ if for every σ, n, t', r such that $[\![e]\!]^{\triangleright} \downarrow^{\sigma;n}_{\mathbf{HeapLangE}^{\triangleright}} t' \approx \mathrm{Ret}(r)$, then $r \neq \bot_{\mathsf{fail}}$ and either $r = \bot_{\mathsf{step\_timeout}}$ or $\phi(r)$.

Definition 4.3. A thread pool tp is *progressive* at heap σ if no thread is stuck: for each $e \in \mathrm{tp}$, there is some $e', \sigma', \vec{e}_f$ so that $e; \sigma \to_{\mathrm{t}} e'; \sigma'; \vec{e}_f$.

e is *operationally adequate* w.r.t. postcondition $\phi : \mathrm{val} \to \mathit{Prop}$ if

(1) for any $\sigma, \sigma', \mathrm{tp}'$ such that $[e]; \sigma \to^*_{\mathrm{tp}} \mathrm{tp}'; \sigma'$, tp' is progressive at heap σ', and
(2) for any $\sigma, \sigma', v, \mathrm{tp}'$ such that $[e]; \sigma \to^*_{\mathrm{tp}} [v] +\!\!+ \mathrm{tp}'; \sigma'$, we have $\phi(v)$.

The following chain of implications holds:

$$\mathrm{wp}^{\triangleright}_{\top}\, e\, \{\phi\} \implies e \text{ is interpretationally adequate w.r.t. } \phi \implies e \text{ is operationally adequate w.r.t. } \phi$$

The first implication is obtained modularly by composing the adequacy theorems seen so far.

The second implication relates the ITree semantics and the operational semantics. For reasons of space, we cannot spell out the details here and refer the reader to our Coq formalization. The proof is a simulation involving an intermediate notion of *ITree traces* that we define in our library. For an operational semantics trace of length n starting at e, the proof constructs by induction an ITree trace in $[\![e]\!]^{\triangleright}$. The proof concludes by constructing the relational interpretation $[\![e]\!]^{\triangleright} \downarrow^{\sigma;n}_{\mathbf{HeapLangE}^{\triangleright}} t'$ from this ITree trace. The latter step is entirely modular and reusable: in our Coq formalization, we provide a number of composable trace lemmata that allow the user to construct relational interpretations from ITree traces.

5 Case study: Islaris

As our second large case study, we redefine the program logic used by Islaris [29] using the approach presented in this paper. Islaris provides an Iris-based program logic for traces that describe the semantics of assembly programs based on authoritative models of real-world assembly languages like Armv8 and RISC-V.

Islaris is an interesting case study for this paper since it uses a variety of effects that exercise the ability of our approach to handle non-standard programming languages. While the original work had to rely on various tricks to fit the Islaris language into the fixed interface provided by Iris, we will see how our approach allows a direct encoding of Islaris using a combination of standard and non-standard events. Concretely, Islaris uses the following event type:

$$\mathbf{IslarisE} \coloneqq \mathbf{DemonicE} \oplus \mathbf{StateE}_{S_{\mathrm{Islaris}}} \oplus \mathbf{FailE} \oplus \mathbf{StepE} \oplus \mathbf{HaltE} \oplus \mathbf{SpecE}$$

We first have the standard events for demonic choice, state, and failure introduced in §2. We also use the **StepE** event (§4.1) to obtain partial correctness reasoning principles for recursive programs. Beyond this, Islaris uses two non-standard events that we discuss next: **HaltE** and **SpecE**.

HaltE. The **HaltE** event type provides the $\mathsf{halt} : \mathbf{itree}\ \mathbf{HaltE}\ \emptyset$ operation that (safely) halts the execution and trivially finishes verification (*i.e.,* $\mathsf{wpi}\ \mathsf{halt}\ \{\Phi\} \dashv\vdash \mathsf{True}$). This operation is necessary since the programming language used by Islaris has an unusual way to read values from registers (and memory), akin to prophecy variables: First, it declares a variable that non-deterministically guesses the value that will be read from the register. Then, the read operation prunes all executions that do not correspond to the actual value of the register using halt. (This unusual encoding of reads comes from the fact that the "statements" in the programming language of Islaris are SMT constraints that can only restrict existing variables but not assign them a new value.) Iris's language interface does not provide a dedicated mechanism to support the halt operation, so Islaris uses a notion of a value that represents a halted program. With logical event handlers, we do not need to rely on such encodings since we can just directly encode halt as its own event.

SpecE. The second non-standard event **SpecE** comes from the fact that Islaris does not just prove the standard Iris adequacy that no program gets stuck (*i.e.,* no fail occurs), but also proves that the memory accesses to specially marked memory regions (representing MMIO regions) satisfy a user-defined safety property. To reason about such externally visible events, Islaris uses an encoding based on the observation mechanism that Jung et al. [19] introduced to reason about prophecy variables. Instead, our approach directly supports defining a **SpecE** event type with an operation $\mathsf{emit}(\kappa)$ where κ represents a visible event (*i.e.,* read or write from resp. to MMIO memory) and a handler ensuring the safety properties are upheld.

No concurrency. Note that **IslarisE** does *not* use the **ConcE** event. This is on purpose since Islaris targets a sequential setting (verifying concurrent assembly programs against authoritative semantics is a research topic on its own). However, while Sammler et al. describe the program logic as sequential in the paper, in the actual Coq formalization it is concurrent (with a sequentially consistent semantics that does not match the concurrency of the actual assembly languages) since the Iris language interface only supports concurrent languages. This means that the original work loses the reasoning principles for sequential languages (*e.g.,* those pertaining to invariants). Our formalization of Islaris does not suffer from this drawback since we can easily model a sequential language by simply not using **ConcE**. You only pay for what you use.

Recreating the Islaris program logic. With the **IslarisE** event type and the corresponding handler at hand, it is straightforward to build a semantic interpretation for the SMT traces and derive the Islaris program logic on top of it. We define a $\mathsf{wp}_{\mathsf{asm}}$ for Islaris using wpi and reprove all program logic rules of the original Islaris. Like for the languages we have seen before, proving the rules is a mostly mechanical application of the wpi rules. We also prove that our program logic satisfies the same adequacy statement as the original work. For this, we leverage the state machine adequacy described in §2.6, showing that this adequacy method can also scale to complex programming languages.

6 Related work

We discuss related work along two axes: program logics that permit reuse across languages, and the state of modeling and reasoning about various computational effects with ITrees.

6.1 Program logics with reuse

To our knowledge, the only prior work that defines a "language-agnostic" program logic intended to be instantiated with a wide range of user-defined languages is Iris. The Iris technical manual [39, §8] describes their *language interface*: an arbitrary type of expressions and global state, and an associated per-thread small-step operational semantics—basically, a state transition system. Given an instance of this interface, Iris the provides a weakest precondition connective and associated program logic rules. However, only the basic structural rules such as a bind lemma and a rule of consequence can be shared. Each language needs to re-define almost every aspect of its operational semantics, and then use "lifting lemmas" to provide corresponding reasoning principles in the program logic. Moreover, the language interface can be quite rigid: for instance, the only way a program may terminate is by returning a value, which means supporting an operation that halts the machine abruptly (but safely) from anywhere in the program requires non-trivial modeling. Iris also has to define two entirely separate weakest precondition connectives for total and partial correctness reasoning. There *is* a reusable library for defining the standard points-to connective, but its relation to the operational semantics needs to be re-proven in each new language.

In contrast, our approach allows more flexibility and more reuse: the same basic program logic can support both total and partial correctness reasoning, a **HaltE** effect for safe machine termination is

easily supported, and the **HeapE** effect library can provide a ready-to-use points-to connective that is already integrated with the ITree semantics, to name but a few. However, so far we have not implemented support for HeapLang's prophecy variables [19] in our approach; that remains an interesting candidate for future work.

Abstract Separation Logic [4] defines a separation logic for a language that is denoted into a trace of "local actions" that each describe how the global state is altered. However, these actions do immediately act on the global state; there is no layering that would allow building up the global state from smaller, reusable pieces. In contrast, our approach supports composing **HeapE** with another instance of **StateE** that governs a separate piece of state (such as a file system or a network).

Dijkstra Monads [37, 15, 1, 23] provide a foundation for deriving weakest precondition connectives for arbitrary monadic computations. By applying a suitable sequence of monad transformers (which are in particular endofunctors on the category of monads) to a base effect observation (a morphism from a computation monad to a specification monad), one can build up the weakest precondition connective effect-by-effect. Their theory therefore encapsulates a different flavor of compositionally built program logics: instead of a single weakest precondition connective that generalizes to a wide range of effects, they systematically derive a new weakest precondition connective for each combination of such effects. However, since monads and concurrent computation are ill-fitted, they do not offer support for reasoning about concurrent programs. Furthermore, reasoning about state in this framework involves directly talking about the entire state; there is no separation logic support for local reasoning about memory.

6.2 ITrees

The ITree line of work has so far mostly been focused on being able to formally capture the semantics of languages in a uniform framework and performing equational reasoning based on those semantics. As such, there has not been a lot of work on program logics for ITrees. The most notable exception is the work by Silver and Zdancewic [33] which applies Dijkstra monads to reason about ITrees. Using the recipe set out by Dijkstra monads, they derive a program logic for a specific ITree-based language (a simple imperative language called IMP) but do not discuss the idea of reusing program logic components and rules across languages.

The by far biggest application of ITrees is the VellVM project [42] which models a significant fraction of the LLVM IR specification using ITrees. They use a wide range of effects for that, most of which are also supported by our framework: several kinds of state, non-deterministic choice, fatal failure, and non-fatal machine termination. The one effect we have not implemented is external function calls; this is an interesting direction for future work. That would then allow us to build a program logic for the VellVM semantics of LLVM IR.

Choice Trees [6] extend ITrees with a *native* form of (demonic) non-deterministic choice. This is quite different from VellVM and our own approach where non-deterministic choice is just yet another effect. The payoff for special-casing choice is that the equational theory can be extended to properly support reasoning about choice. Our use of a program logic can be seen as an alternative approach for reasoning about ITrees with non-determinism that avoids special-casing. One case study for Choice Trees is a model of cooperative concurrency very similar to ours: thread forking is defined by duplicating the continuation. They give semantics to this model via a non-deterministic scheduler expressed directly in Choice Trees. We believe that our relational interpretation of the concurrency effects produce the same result. The new contribution of our framework is that we build a fully-featured concurrent separation logic for reasoning about these ITrees, enjoying all the concurrent reasoning principles that Iris provides. As part of our HeapLang case study, we also proved that this cooperative model of concurrency indeed soundly models all possible behaviors of a language defined with a small-step operational semantics and preemptive concurrency. Our

approach is also able to support angelic choice as yet another effect, in contrast to Choice Trees that only support demonic choice.

Guarded interaction trees [11] (GITrees) provide a fully denotational model of a language with higher-order state into a variant of ITrees with support for higher-order events. This is different from our model of HeapLang: our heap stores *syntactic* HeapLang values, representing closures as expressions rather than their denotations. This makes our model much less suited for equational reasoning, but also much less technically demanding. Furthermore, event interpretation in GITrees hard-codes the state monad and therefore does not support other effects such as non-determinism and concurrency. In the future, it would be interesting to combine these lines of work and extend our program logic to support reasoning about GITrees.

References

[1] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *POPL*. ACM, 515–529. https://doi.org/10.1145/3009837.3009878

[2] Andrew W. Appel. 2016. Modular Verification for Computer Security. In *CSF*. IEEE Computer Society, 1–8. https://doi.org/10.1109/CSF.2016.8

[3] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *POPL*. ACM, 339–352. https://doi.org/10.1145/1706299.1706339

[4] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

[5] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[6] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *PACMPL* 7, POPL (2023), 1770–1800. https://doi.org/10.1145/3571254

[7] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1 (2023), 5:1–5:43. https://doi.org/10.1145/3579834

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2016. Using Crash Hoare Logic for Certifying the FSCQ File System. In *USENIX Annual Technical Conference*. USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/chen_haogang

[9] Santiago Cuellar, Nick Giannarakis, Jean-Marie Madiot, William Mansky, Lennart Beringer, Qinxiang Cao, and Andrew W. Appel. 2020. Compiler correctness for concurrency: from concurrent separation logic to shared-memory assembly language. https://www.cs.princeton.edu/~appel/papers/ccc.pdf

[10] Robert W. Floyd. 1967. Nondeterministic Algorithms. *J. ACM* 14, 4 (1967), 636–644. https://doi.org/10.1145/321420.321422

[11] Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *PACMPL* 8, POPL (2024), 332–361. https://doi.org/10.1145/3632854

[12] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

[13] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *PACMPL* 7, OOPSLA2 (2023), 716–744. https://doi.org/10.1145/3622823

[14] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS, Vol. 4960)*. 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

[15] Bart Jacobs. 2015. Dijkstra and Hoare monads in monadic computation. *Theor. Comput. Sci.* 604 (2015), 30–45. https://doi.org/10.1016/J.TCS.2015.03.020

[16] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *PACMPL* 8, POPL (2024), 1385–1417. https://doi.org/10.1145/3632889

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[19] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: Prophecy variables in separation logic. *PACMPL* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[20] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[21] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[22] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

[23] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *PACMPL* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

[24] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. *PACMPL* 1, OOPSLA (2017), 87:1–87:28. https://doi.org/10.1145/3133911

[25] William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *PACMPL* 8, POPL (2024), 148–174. https://doi.org/10.1145/3632848

[26] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A concurrent program logic with a future and history. *PACMPL* 6, OOPSLA2 (2022), 1378–1407. https://doi.org/10.1145/3563337

[27] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/J.TCS.2006.12.035

[28] Ingrid Rewitzky. 2003. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*. LNCS, Vol. 2929. Springer, 256–271. https://doi.org/10.1007/978-3-540-24615-2_12

[29] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI*. ACM, 825–840. https://doi.org/10.1145/3519939.3523434

[30] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI*. 158–174. https://doi.org/10.1145/3453483.3454036

[31] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *PACMPL* 7, POPL (2023), 775–805. https://doi.org/10.1145/3571220

[32] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *SOSP*. ACM, 113–129. https://doi.org/10.1145/3600006.3613172

[33] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *PACMPL* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

[34] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *PACMPL* 7, POPL (2023), 1121–1151. https://doi.org/10.1145/3571232

[35] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *PACMPL* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[36] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library. In *ECOOP (Lecture Notes in Computer Science, Vol. 7920)*. Springer, 327–351. https://doi.org/10.1007/978-3-642-39038-8_14

[37] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *PLDI*. ACM, 387–398. https://doi.org/10.1145/2491956.2491978

[38] The Coq Team. 2024. The Coq proof assistant. https://coq.inria.fr/.

[39] The Iris Team. 2024. The Iris 4.2 Reference. https://plv.mpi-sws.org/iris/appendix-4.2.pdf

[40] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

[41] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *PACMPL* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[42] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *PACMPL* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572