Mosaic: From Language Fragments to Program Logics

MAX VISTRUP, ETH Zurich, Switzerland
KLAUS KRASSNITZER, Institute of Science and Technology Austria (ISTA), Austria
MICHAEL SAMMLER, Institute of Science and Technology Austria (ISTA), Austria
RALF JUNG, ETH Zurich, Switzerland

Program logics have proven effective at verifying programs by leveraging modular reasoning. However, the process of building program logics is itself usually not modular: Even though many languages share common features like integers, booleans, heaps, or concurrency, program logics for different languages are usually defined and proven sound independently.

In this paper, we present *Mosaic*, a new framework for building program logics by composition. Mosaic views a programming language as a sum of *fragments*. Each fragment covers a particular feature or aspect of a language, supplying syntax, semantics, program logic, and soundness of said logic. These fragments can be freely combined to obtain a programming language with an accompanying program logic. When realized in a proof assistant, each fragment furthermore comes with a set of tactics that are then automatically available in any language that includes the fragment.

We provide a library of 27 fragments, including fragments for higher-order heaps, concurrency, and atomic blocks. Particularly noteworthy is our fragment for prophecy variables which comes with a reusable, language-generic erasure theorem. To demonstrate the scalability of our approach, we use it to recreate the HeapLang and λ_{Rust} languages from prior work. We show that our composed versions are usable in practice by porting several large proof examples with minimal changes. All of our development is formalized using Rocq and Iris.

1 Introduction

As PL researchers, we often define new programming languages. However, those languages are rarely *entirely* new but usually share many common pieces with other languages (*e.g.,* integer arithmetic or lambda abstractions). And yet, languages are typically defined *monolithically*, which means all those seemingly common pieces have to be re-defined for every new language, again and again. Doing so may not break your back if you are content with just the syntax and operational semantics, but it can become very strenuous if you also want a *program logic*: every time, you must spell out the laws of those common pieces, and every time, you must prove those rules correct—even if they coincide *exactly* with rules proven sound elsewhere! And if you carry out your work in a proof assistant, you have to further build up language-specific tactic infrastructure and other conveniences, over and over again, for each new language.

To escape this cycle of reinvention, we need a way to compose a language from *reusable fragments*. Each language fragment defines a set of syntactic constructs of a language (*e.g.,* integers with their operations or a higher-order heap with heap operations) and equips them with all the necessary bells and whistles: runtime semantics, program logic, theorems relating the two, and (if desired) tactic support for interactive proofs. Additionally, we need a framework to compose a set of such fragments into a language that *automatically* inherits all these aspects of the individual fragments.

Prior work has investigated various principled methods for building syntax and semantics of programming languages from composable fragments [30, 26, 33]. What has been missing thus far is

Authors' Contact Information: Max Vistrup, ETH Zurich, Zürich, Switzerland, max.vistrup@inf.ethz.ch; Klaus Kraßnitzer, Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria, klaus.krassnitzer@ista.ac.at; Michael Sammler, Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria, ...; Ralf Jung, ETH Zurich, Zürich, Switzerland, ralf.jung@inf.ethz.ch.

2026. ACM XXXX-XXXX/2026/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a mechanism to obtain a program logic for the resulting language. In this paper, we present *Mosaic* to close this gap and enable advanced, scalable reasoning in languages built from fragments.

The skeleton of our approach can be summarized as follows. We take the "Data types à la Carte" approach [30] to compose the *syntax* of a language from individually defined fragments. Then, we need to give *semantics* to each of these syntax fragments. To address this problem, we identify Interaction Trees (ITrees) [35] as a particularly well-suited theory. Then, we show how to equip each syntax fragment with *program logic rules*. Each language automatically inherits the program logic rules from its constituent fragments, providing a program logic for free. To prove *soundness* of the program logic rules, we leverage the program logic for ITrees by Vistrup et al. [34] which is based on Iris [17]. This allows us to prove the soundness of each rule individually, which can then be automatically composed into a soundness proof of the full program logic.

An important part of our work is pushing this idea to cover many important programming language features. We provide a large library of fragments that users of Mosaic can import to build new languages. One particular highlight is a fragment for prophecy variables [1, 2]. Previous instantiations of this feature in Iris required a lot of language-specific work [18]. Our prophecy variable fragment can be added to any Mosaic program logic, and we prove its soundness in a language-generic way. This makes it effortless for any language to support prophecy variables.

To show that our framework scales to state-of-the-art formalizations of programming languages, we use it to build multiple program logics. We build models of HeapLang [15] and λ_{Rust} [16] assembled from fragments. As expected, these two models share a number of fragments, reflecting the fact that the languages share a number of features. We develop a compositional theory for proving such models faithful against the operational semantics of the original languages. Beyond recovering all the original proof rules and soundness guarantees, we show that our implementations provide the same ergonomics as the existing implementations by porting several verification results with minimal adaptation. We also show how our approach enables building new languages with minimal effort. In particular, we construct languages that extend HeapLang with a file system and a network, respectively, while reusing all the existing fragments.

Contributions. We present Mosaic, a new approach for building program logics from composable fragments. We make the following contributions:

- A notion of "language fragment" for packaging up syntax, semantics, program logic, and soundness proof, as well as a framework for composing fragments into a language (§2)
- Fragments for concurrency and for atomic blocks (§3)
- A compositional mechanism to relate ITree-based semantics to operational semantics (§4)
- A fragment for prophecy variables, providing for the first time a language-independent reusable account of prophecy variables and associated soundness/erasure theorems (§5)
- A Rocq implementation of the entire framework (§6)
- A large library of fragments that can be composed to build and extend HeapLang and λ_{Rust} (§7)

Limitations. As with any compositional approach to programming languages, the pieces must be "local" in a suitable sense. Fragments cannot globally change the semantics in a way that affects other fragments, nor can they interfere with the internal state of other fragments. Nevertheless, fragments can have shared state. These restrictions are mild enough to still allow us to accommodate a large set of language features (including concurrency, atomic blocks, prophecy variables, and various memory models), as we demonstrate throughout the paper. However, we have not investigated control effects such as exception handling and therefore do not know to what extent those could be supported by our framework. Thread-local state, which appears less demanding in comparison, is also future work. Weak memory is another technically demanding feature we do not support.

2 Overview of Mosaic

What does it take to model a programming language formally? Under the classical view, we write down the syntax of the language all at once, and then we equip it with an operational semantics. For example, we could write the syntax of a small ML-like language as follows:

$$v \in \mathsf{val} ::= () \mid n \mid \texttt{true} \mid \texttt{false} \mid \lambda x.\, e \mid \ell \qquad (n \in \mathbb{Z}, \ell \in \mathbb{N})$$
$$e \in \mathsf{expr} ::= v \mid x \mid e_1 + e_2 \mid e_1 \le e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \lambda x.\, e \mid e_1(e_2) \mid \texttt{ref}(e) \mid \texttt{!}\, e \mid e_1 \leftarrow e_2$$

Then, to endow these terms with computational meaning, we would define a small-step operational semantics. This typically comes in the form of a stepping relation $(e_1, \sigma_1) \to (e_2, \sigma_2)$, which is understood to mean that e_1 with heap σ_1 is able to step to e_2 with heap σ_2. Furthermore, to obtain reasoning principles for showing correctness of programs written in this language, or to prove type soundness using the "logical" approach [32], we have to define a program logic and prove its soundness against the operational semantics. And finally, if we want to carry out our work in an interactive theorem prover, we typically need to develop some amount of language-specific tactic infrastructure to make such proofs less burdensome.

This strategy works but leaves something to be desired. The language above is but a remix of well-understood components: languages with heaps have already been studied, and so have languages with lambdas and with if-conditionals. It therefore seems improper that we should have to re-model every single one of these features, writing down syntax, semantics, and rules again. Of course, this is merely a toy example, and yet it is not hard to imagine the tedium of scaling it to a modern, complex language.

These concerns invite us to think about dissecting the language into an "orthogonal basis" of *fragments*. On the level of the language syntax, we want to take apart the grammar as follows:

$$\begin{aligned}
\mathsf{expr}_{\text{VAL}} &::= v & \mathsf{val}_{\text{UNIT}} &::= () \\
\mathsf{expr}_{\text{INT}} &::= e_1 + e_2 & \mathsf{val}_{\text{INT}} &::= n \\
\mathsf{expr}_{\text{BOOL}} &::= e_1 \le e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \mathsf{val}_{\text{BOOL}} &::= \texttt{true} \mid \texttt{false} \\
\mathsf{expr}_{\text{FUN}} &::= x \mid \lambda x.\, e \mid e_1(e_2) & \mathsf{val}_{\text{FUN}} &::= \lambda x.\, e \\
\mathsf{expr}_{\text{HEAP}} &::= \texttt{ref}(e) \mid \texttt{!}\, e \mid e_1 \leftarrow e_2 & \mathsf{val}_{\text{HEAP}} &::= \ell
\end{aligned}$$

(These are some examples of fragments, but we shall see many more later.) Each of these fragments pertains to just a single feature of the language, and the language is obtained by composition. Crucially, the e and v in this grammar refer to the *entire* language, not just the current fragment.

However, decomposing the grammar is not enough. We want each fragment to be an entire "slice" of a programming language: each fragment should cover only a part of a language but every aspect of that part. Specifically, each fragment contains syntax (§ 2.1), the (dynamic) semantics for that syntax (§ 2.2), a program logic for reasoning about those syntax constructs (§ 2.3), and a soundness proof relating the program logic to the semantics (§ 2.4). In the following, we show how Mosaic models each of those aspects of a language in a compositional way.

2.1 Syntax of a Fragment

The syntax of a fragment is defined using the "Data types à la Carte" approach [30]. This means that each fragment defines type-level functions that take as input the types of expressions and values of the *entire* language, and produce the expressions and values of this *specific fragment*. For

the INT and FUN fragments, that looks as follows (using BNF-style notation to define data types):

$$\begin{aligned}
\mathbf{E}_{\text{INT}}(E, V) &::= e_1 + e_2 && (e_1, e_2 \in E)\\
\mathbf{V}_{\text{INT}}(E, V) &::= n && (n \in \mathbb{Z})\\
\mathbf{E}_{\text{FUN}}(E, V) &::= x \mid \lambda x.\, e \mid e_1(e_2) && (e, e_1, e_2 \in E, x \in B)\\
\mathbf{V}_{\text{FUN}}(E, V) &::= \lambda x.\, e && (e \in E, x \in B)
\end{aligned}$$

Here, B is the set of binders, *i.e.,* variable names. This parametrization by the global expression/value type is crucial in order to allow syntax to be combined across fragments. For example, in spite of $e_1 + e_2$ not belonging to the FUN fragment, $\lambda x.\, x + x$ is a perfectly valid expression in the syntax obtained from combining the INT fragment with the FUN fragment.

Composing a language. To turn a collection of fragments into a closed language (which we will typically abbreviate $\mathcal{L}$), we take a mutual fixed point that satisfies equations like the following:[1]

$$\begin{aligned}
\mathbf{E}_{\mathcal{L}} &= \mathbf{E}_{\text{VAL}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{E}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{E}_{\text{BOOL}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{E}_{\text{FUN}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{E}_{\text{HEAP}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}})\\
\mathbf{V}_{\mathcal{L}} &= \mathbf{V}_{\text{UNIT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{V}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{V}_{\text{BOOL}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{V}_{\text{FUN}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \uplus \mathbf{V}_{\text{HEAP}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}})
\end{aligned}$$

Syntactic fragment inclusion. Having defined a language through the composition of fragments, we want to eventually state general theorems about any language that contains a given fragment. To this end, we have a notion of *inclusion of syntax*. For any fragment FRAG, we define:

$$\text{FRAG} \sqsubseteq_{\text{syn}} \mathcal{L} := \mathbf{E}_{\text{FRAG}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \subseteq \mathbf{E}_{\mathcal{L}} \wedge \mathbf{V}_{\text{FRAG}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \subseteq \mathbf{V}_{\mathcal{L}}$$

With a language defined as above, we automatically obtain that each of the fragments involved in the fixed point is included in the resulting language.

2.2 Semantics of a Fragment

Having split the syntax into fragments, we'd like to do the same to semantics. Operational semantics does not readily lend itself to this purpose. For one, under operational semantics, every expression acts on a global state. In our example language, that would mean even an innocuous step like $(\texttt{if true then } e_2 \texttt{ else } e_3) \to e_2$ must specify how it acts on the heap, even though that operation has nothing to do with the heap. One could try to remedy this by dividing the state into fragment-specific states $S = S_1 \times \cdots \times S_n$, but the states of fragments are not in general easily separable: for example, we might have a HEAP fragment that provides basic heap operations and a separate CAS fragment providing an atomic compare-and-swap operation, both of which access the same heap.

Instead of operational semantics, we will define the semantics of our fragments by means of *definitional interpreters*. As a first approximation, we could consider having each fragment declare its behavior by a simple mathematical function. For instance, the semantics of INT would be given by a function with the following type:

$$[\![\cdot]\!]_{\text{INT}} : \forall \mathcal{L}.\, \mathbf{E}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \mathbf{V}_{\mathcal{L}}$$

Conceptually, this maps expressions from $\mathbf{E}_{\text{INT}}$ to values. However, recall from §2.1 that $\mathbf{E}_{\text{INT}}$ is not a type but a type-level function that receives the E and V that represent the expressions and values of the entire language, respectively. Therefore, the semantics is a polymorphic function: given any language, they define how the expressions added by this particular fragment are evaluated.

However, if we try to define $[\![e_1 + e_2]\!]_{\text{INT}}$, we quickly run into multiple problems: (1) We know nothing about the type $\mathbf{V}_{\mathcal{L}}$ we are supposed to return here, so even if we knew which integer to

[1]The type-level functions for each fragment must be polynomial functors for this fixed point to exist. Syntactically, this amounts to being a *strictly positive inductive*. All our syntaxes are written in BNF-style and such definitions always satisfy this condition.

return, we would have no way to embed that integer into the desired return type. (2) We have to recursively evaluate e_1 and e_2 before we can perform the actual arithmetic. (3) We obviously will not be able to write the definitional interpreter as a *pure* function for all our fragments. The HEAP fragment will have to somehow model its side effects.

To overcome (1), we need to be able to put restrictions on the language $\mathcal{L}$. In particular, the interpreter should only have to deal with languages that actually include our fragment.

To overcome (2) and (3), we make use of a *monadic* interpreter. Specifically, we are using *interaction trees* or *ITrees* [35], which provide a general framework for modeling monadic computations with recursion and arbitrary effects. Before we go on, we briefly explain the basics of ITrees.

Background: ITrees. ITrees [35] are a monad for potentially non-terminating computations with general effects. The type **itree** $E\ R$ is parameterized by a return type R : **Type** and an event type E : **Type** $\to$ **Type**. The type $E\,A$ represents the events ϵ that return an answer of type A. ITrees are defined by coinduction:

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

$\mathsf{Ret}(r)$ returns a value $r : R$ and terminates execution, $\mathsf{Tau}(t)$ takes a silent step and continues the computation t, and $\mathsf{Vis}_A(\epsilon, k)$ emits an event $\epsilon : E\,A$, receives an answer $a : A$, and then continues with $k(a)$. Tau steps "do nothing" but are nevertheless needed for representing infinite loops. **itree** $E\ R$ is a monad in R, and we use notation like $v_1 \leftarrow \cdots ; \mathsf{Ret}(\cdots)$ to construct ITrees. The canonical notion of equality on ITrees is $\approx$ ("equivalence up-to τ"), a bisimulation which says that two ITrees are equal modulo finite sequences of Tau steps.

Using ITrees to model semantic fragments. With ITrees in our quiver, we return to the discussion of semantic fragments. To reveal the full type signature of the semantics of a fragment, we just have to answer one last question: which events will we allow the fragment to use? The answer is, any events that it wants. More specifically, each fragment chooses what events it depends on. For example, INT needs only one event type: **FailE** which lets us abort the execution with a global failure (*i.e.,* Undefined Behavior). Triggering this event corresponds to a stuck execution in a typical operational semantics. In addition to the events chosen, one event is always made available to every fragment: **EvalE** will allow us to recursively evaluate expressions.

With this out of the way, we can finally spell out the semantics of INT:

$$\begin{aligned}
[\![\cdot]\!]_{\text{INT}} &: \forall \mathcal{L}.\ \text{INT} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{E}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \textbf{itree}\ (\textbf{EvalE} \oplus \textbf{FailE})\ \mathbf{V}_{\mathcal{L}} \\
[\![e_1 + e_2]\!]_{\text{INT}} &:= v_2 \leftarrow \mathsf{eval}(e_2); v_1 \leftarrow \mathsf{eval}(e_1); n_1 \leftarrow \mathsf{to_int}(v_1); n_2 \leftarrow \mathsf{to_int}(v_2); \mathsf{Ret}(n_1 + n_2) \\
\mathsf{to_int} &: \forall \mathcal{L}.\ \text{INT} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{V}_{\mathcal{L}} \to \textbf{itree FailE}\ \mathbb{Z} \\
\mathsf{to_int}(v) &:= \text{match } v \text{ with } n \implies \mathsf{Ret}(n) \mid _ \implies \text{fail end}
\end{aligned}$$

We elide the $\mathcal{L}$ argument when invoking these functions. Evaluating an addition $e_1 + e_2$ begins by triggering the eval event to recursively evaluate the subexpressions e_2, e_1, in right-to-left evaluation order.[2] (Rectifying that recursion is handled by Mosaic. We will get back to this at the end of the subsection.) Then we ensure that both subexpressions are actually integers using the to_int helper function. If not, a fail event is triggered. This corresponds to an operational semantics where addition gets stuck if one of the operands is not an integer. Finally, the actual arithmetic operation is defined using arithmetic on meta-level integers, as is usual in definitional interpreters.

But that is just the simplest possible case: a fragment whose semantics can be defined on its own, with only a modicum of effects. Next, we consider the BOOL fragment. This fragment is slightly more complicated because it *depends* on INT: we include integer comparison in BOOL, and somehow

[2]In our mechanization, evaluation order is configurable per language and fragments can be evaluation-order-agnostic (§7.2).

the semantics of BOOL will need to extract integers (supplied by INT) from its subexpressions. To enable this, we allow fragments to declare dependencies on arbitrary other fragments. This is reflected in the type signature for their semantics by adding corresponding inclusion assumptions:

$$[\![\cdot]\!]_{\text{BOOL}} : \forall \mathcal{L}.\ \text{BOOL}, \text{INT} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{E}_{\text{BOOL}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \textbf{itree}\ (\textbf{EvalE} \oplus \textbf{FailE})\ \mathbf{V}_{\mathcal{L}}$$

$$[\![e_1 \le e_2]\!]_{\text{BOOL}} := v_2 \leftarrow \text{eval}(e_2); v_1 \leftarrow \text{eval}(e_1); n_1 \leftarrow \text{to\_int}(v_1); n_2 \leftarrow \text{to\_int}(v_2); \\ \text{if } n_1 \le n_2 \text{ then Ret(}\texttt{true}\text{) else Ret(}\texttt{false}\text{)}$$

$$[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_{\text{BOOL}} := v_1 \leftarrow \text{eval}(e_1); b_1 \leftarrow \text{to\_bool}(v_1); \text{if } b_1 \text{ then eval}(e_2) \text{ else eval}(e_3)$$

We omit to_bool which is analogous to to_int. As before, we define the semantics of these operations using their meta-level equivalents. Note that `if` expressions can return values of *arbitrary* fragments, which explains why the return type of the semantic function is $\mathbf{V}_{\mathcal{L}}$ rather than $\mathbf{V}_{\text{BOOL}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}})$.

With ITree effects at our disposal, defining the semantics for HEAP is really no more complicated than what we have already seen. The main difference is that we depend on the $\mathbf{HeapE}_{\mathbf{V}_{\mathcal{L}}}$ ITree event type, which provides the events for getting and setting the ($\mathbf{V}_{\mathcal{L}}$-valued) heap:

$$[\![\cdot]\!]_{\text{HEAP}} : \forall \mathcal{L}.\ \text{HEAP} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{E}_{\text{HEAP}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \textbf{itree}\ (\textbf{EvalE} \oplus \textbf{FailE} \oplus \mathbf{HeapE}_{\mathbf{V}_{\mathcal{L}}})\ \mathbf{V}_{\mathcal{L}}$$

$$[\![\texttt{ref}(e)]\!]_{\text{HEAP}} := v \leftarrow \text{eval}(e); \text{alloc}(v)$$

$$[\![!\, e]\!]_{\text{HEAP}} := v \leftarrow \text{eval}(e); \ell \leftarrow \text{to\_loc}(v); \text{load}(\ell)$$

$$[\![e_1 \leftarrow e_2]\!]_{\text{HEAP}} := v_2 \leftarrow \text{eval}(e_2); v_1 \leftarrow \text{eval}(e_1); \ell \leftarrow \text{to\_loc}(v_1); \text{store}(\ell, v_2)$$

This uses the alloc, load, and store operations of $\mathbf{HeapE}_{\mathbf{V}_{\mathcal{L}}}$, which have the usual heap semantics. Note that $\mathbf{V}_{\mathcal{L}}$ includes all values available in the full language, including *e.g.*, $\lambda x.\ e$ from FUN, so this heap is fully higher-order.

Dealing with substitution. There is one more fragment we have to show: FUN, the fragment that contains lambda abstractions $\lambda x.\ e$. This fragment is non-trivial as we have to somehow transport the value passed to a function to the places where the argument is used. This could be achieved using a dedicated environment, but to obviate the need to lug around such an environment in the program logic, we instead do as many formalizations do and opt for using β-reduction.

Yet, this creates a new challenge for splitting apart our language into fragments: the semantics of the FUN fragment will have to somehow substitute a variable for a value in some arbitrary expression. To deal with that, we need some infrastructure for substitution. Specifically, fragments can support a substitution operation in addition to their regular semantics. The FUN fragment will then assume that all fragments of the language support that operation.

To define substitution fragment-wise, each syntax fragment specifies how to "lift" a notion of substitution from the global expression type to the fragment. To be more precise, a syntax fragment FRAG with substitution supplies a substitution function with the following signature:

$$\text{subst}_{\text{FRAG}} : \forall \mathcal{L}.\ \text{FRAG}, \text{VAL} \sqsubseteq_{\text{syn}} \mathcal{L} \to B \to \mathbf{V}_{\mathcal{L}} \to (\mathbf{E}_{\mathcal{L}} \to \mathbf{E}_{\mathcal{L}}) \to (\mathbf{E}_{\text{FRAG}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \mathbf{E}_{\mathcal{L}})$$

Just like the semantics from earlier, substitution is parameterized by an arbitrary language. It takes as argument the binder $x \in B$ that is to be replaced and the value $v \in \mathbf{V}_{\mathcal{L}}$ to replace it with. Moreover, it takes a function $f : \mathbf{E}_{\mathcal{L}} \to \mathbf{E}_{\mathcal{L}}$ that performs the substitution of v for x in subexpressions. Finally, it takes an expression e of the fragment and returns the substituted expression. For example, for the FUN fragment, we set

$$\text{subst}_{\text{FUN}}(x, v, f, e_1(e_2)) := f(e_1)(f(e_2))$$

$$\text{subst}_{\text{FUN}}(x, v, f, x') := \begin{cases} v & x = x' \\ x' & \text{otherwise} \end{cases} \qquad \text{subst}_{\text{FUN}}(x, v, f, \lambda x'.\ e) := \begin{cases} \lambda x'.\ e & x = x' \\ \lambda x'.\ f(e) & \text{otherwise} \end{cases}$$

This example also illustrates why the codomain of a substitution function $\text{subst}_{\text{FRAG}}$ is $\mathbf{E}_{\mathcal{L}}$ in lieu of $\mathbf{E}_{\text{FRAG}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}})$: the return value might lie outside the fragment, such as in the variable case above where we return v (an expression of VAL).

Given such a fragment substitution function for each fragment, we define the substitution function for the entire language by taking a fixed point. This assumes a technical condition for the fragment substitution functions (they may only invoke f on structurally smaller terms). The resulting *global* substitution function has the usual signature:

$$\text{subst}_{\mathcal{L}} : B \to \mathbf{V}_{\mathcal{L}} \to (\mathbf{E}_{\mathcal{L}} \to \mathbf{E}_{\mathcal{L}})$$

Now that substitution is available, we can finally define the semantics for FUN:

$$\begin{aligned}
[\![\cdot]\!]_{\text{FUN}} &: \forall \mathcal{L}.\ \text{FUN} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{E}_{\text{FUN}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \mathbf{itree}\ (\mathbf{EvalE} \oplus \mathbf{FailE})\ \mathbf{V}_{\mathcal{L}} \\
[\![x]\!]_{\text{FUN}} &:= \text{fail} \qquad\qquad [\![\lambda x.\ e]\!]_{\text{FUN}} := \text{Ret}(\lambda x.\ e) \\
[\![e_1(e_2)]\!]_{\text{FUN}} &:= v_2 \leftarrow \text{eval}(e_2); v_1 \leftarrow \text{eval}(e_1); (x, e) \leftarrow \text{to_lam}(v_1); \text{eval}(\text{subst}_{\mathcal{L}}(x, v_2, e))
\end{aligned}$$

Evaluating a free variable fails because that means the program did not bind all its variables. Next, lambda-expressions just produce the corresponding lambda-values. The interesting case is the last one. To evaluate an application, we first evaluate the subterms as usual. Then we ensure that the left subterm evaluates to a lambda term: to_lam is a helper similar to to_int that returns the constituent pieces of a lambda value, the binder and the body. Finally, we apply the substitution function to perform β-reduction and evaluate the result.[3]

Note that the substitution function is independent of the FUN fragment, and can hence be reused by other fragments. In our case studies, we use this to define a fragment for n-ary functions that pass multiple arguments at once.

Composing a language. With an interpreter for each fragment at hand, we can now define the semantics of a language that is generated by composing an arbitrary set of fragments. This is a two-step process. In the first step, we define a single big interpreter function that simply determines the fragment the current expression occurs in, and dispatches to the correct interpreter. It has the following signature, where **LangE** is the ITree event type obtained by composing all the event types needed by the fragments:

$$\text{pre\_interp}_{\mathcal{L}} : \mathbf{E}_{\mathcal{L}} \to \mathbf{itree}\ (\mathbf{EvalE} \oplus \mathbf{LangE})\ \mathbf{V}_{\mathcal{L}}$$

However, this pre-interpreter is not particularly useful: for almost all expressions, it will simply stop with an **EvalE** event, indicating the next subexpression to evaluate. To obtain an actual interpreter for the language, we apply an ITree fixed point combinator [35, §4], which replaces each occurrence of the **EvalE** event with a recursive application of the interpreter itself. The upshot is an interpreter for the entirety of our language:

$$[\![\cdot]\!]_{\mathcal{L}} : \mathbf{E}_{\mathcal{L}} \to \mathbf{itree}\ \mathbf{LangE}\ \mathbf{V}_{\mathcal{L}}$$

Semantic fragment inclusion. As before, we will sometimes want to quantify over an *arbitrary* language that includes a certain fragment's semantics. To achieve this, we define a notion of *inclusion of semantics*:

$$\text{FRAG} \sqsubseteq_{\text{sem}} \mathcal{L} := \text{FRAG} \sqsubseteq_{\text{syn}} \mathcal{L} \wedge \forall e \in \mathbf{E}_{\text{FRAG}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}).\ \text{pre\_interp}_{\mathcal{L}}(e) = [\![e]\!]_{\text{FRAG}}$$

[3]Note that the last call to eval is not structurally recursive but is nevertheless valid because of ITree's general fixed point combinators.

2.3 Program Logic

At last, we come to the program logic. The pattern and philosophy remain the same. We want each fragment to define a program logic with its own set of proof rules, and we want to put those pieces together to form a program logic for the entire language.

To achieve this, we take a semantic approach. This avoids the complications of defining an inductive relation (which is essentially a dependently typed data type) in terms of fragments. We instead define once and for all a *model* for a program logic for an arbitrary language, and we let each fragment state properties of this model for its particular part of the language. As the surrounding framework for defining this model, we choose Iris [17]. Being a higher-order separation logic, Iris is a natural choice for reasoning about languages with higher-order mutable state.

The model of a Mosaic program logic is given by a weakest precondition connective with the following signature:

$$\mathsf{wp}_{\mathcal{L}}\ \cdot\ \{\cdot\} : \mathbf{E}_{\mathcal{L}} \to (\mathbf{V}_{\mathcal{L}} \to \mathit{iProp}) \to \mathit{iProp}$$

Here, *iProp* is the type of Iris propositions, *i.e.,* separation logic assertions. The meaning of $\mathsf{wp}_{\mathcal{L}}\ e\ \{\Phi\}$ is that the program e can execute safely (executions will never fail) and if it terminates, the final value will satisfy the predicate Φ. Note that $\mathsf{wp}_{\mathcal{L}}$ is itself a separation logic assertion and as such can own parts of the heap and other resources; the execution of e must only depend on those parts of the heap that are owned by the $\mathsf{wp}_{\mathcal{L}}$. We will discuss the actual definition of $\mathsf{wp}_{\mathcal{L}}$ in the next subsection when we get to soundness of the program logic. Until then, we will treat it as opaque and focus on how fragments can define proof rules for their part of the language.

Stating proof rules for a fragment. The INT fragment can now claim a proof rule for its expressions by stating the following theorem:

$$\forall \mathcal{L}.\ \textsc{int} \sqsubseteq_{\mathsf{sem}} \mathcal{L} \Rightarrow \forall \Phi.\ \Phi(n_1 + n_2) \vdash \mathsf{wp}_{\mathcal{L}}\ n_1 + n_2\ \{\Phi\} \qquad \text{(WpAdd)}$$

In other words, for any language that contains this fragment, the theorem says that we can prove that the result of the expression $n_1 + n_2$ (this is the syntax for addition in our fragment) satisfies postcondition Φ by proving $\Phi(n_1 + n_2)$ (adding the numbers in the meta logic). The $\vdash$ here refers to entailment of the Iris logic.

To make these theorems easier to read, we will state them in inference rule form, with an annotation indicating that these rules require INT to be included in the language:

$$\textsc{WpAdd}\ (\textsc{int} \sqsubseteq_{\mathsf{sem}} \mathcal{L})\ \frac{\Phi(n_1 + n_2)}{\mathsf{wp}_{\mathcal{L}}\ n_1 + n_2\ \{\Phi\}} \qquad \textsc{WpAddBindR}\ (\textsc{int} \sqsubseteq_{\mathsf{sem}} \mathcal{L})\ \frac{\mathsf{wp}_{\mathcal{L}}\ e_2\ \{v_2.\ \mathsf{wp}_{\mathcal{L}}\ e_1 + v_2\ \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L}}\ e_1 + e_2\ \{\Phi\}}$$

$$\textsc{WpAddBindL}\ (\textsc{int} \sqsubseteq_{\mathsf{sem}} \mathcal{L})\ \frac{\mathsf{wp}_{\mathcal{L}}\ e_1\ \{v_1.\ \mathsf{wp}_{\mathcal{L}}\ v_1 + v_2\ \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L}}\ e_1 + v_2\ \{\Phi\}}$$

Aside from the rule that corresponds to the "base" reduction of actually performing arithmetic, we also have two bind rules that let us verify a compound expression by verifying each subexpression.

Bind and evaluation contexts. Stating such bind lemmata quickly gets repetitive, which is why Iris-based logics typically define a single uniform bind rule that works for any *evaluation context* K:

$$\textsc{WpBind}\ \frac{\mathsf{wp}_{\mathcal{L}}\ e\ \{v.\ \mathsf{wp}_{\mathcal{L}}\ K[v]\ \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L}}\ K[e]\ \{\Phi\}}$$

This rule says that if the current program can be decomposed into an evaluation context K filled with expression e, then we can verify that program by first verifying e, and then verifying K filled with the result of e. This generalizes the usual rule of sequential composition.

$$\textsc{WpVal}\ (\textsc{val} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\Phi(v)}{\mathrm{wp}_{\mathcal{L}}\, v\, \{\Phi\}} \qquad \textsc{WpBeta}\ (\textsc{fun} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\mathrm{wp}_{\mathcal{L}}\, e[v/x]\, \{\Phi\}}{\mathrm{wp}_{\mathcal{L}}\, (\lambda x.\, e)(v)\, \{\Phi\}} \qquad \textsc{WpLeq}\ (\textsc{bool} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\Phi(n_1 \le n_2)}{\mathrm{wp}_{\mathcal{L}}\, n_1 \le n_2\, \{\Phi\}}$$

$$\textsc{WpIfTrue}\ (\textsc{bool} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\mathrm{wp}_{\mathcal{L}}\, e_1\, \{\Phi\}}{\mathrm{wp}_{\mathcal{L}}\, \texttt{if true then}\ e_1\ \texttt{else}\ e_2\, \{\Phi\}} \qquad \textsc{WpIfFalse}\ (\textsc{bool} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\mathrm{wp}_{\mathcal{L}}\, e_2\, \{\Phi\}}{\mathrm{wp}_{\mathcal{L}}\, \texttt{if false then}\ e_1\ \texttt{else}\ e_2\, \{\Phi\}}$$

Fig. 1. Proof rules for the remaining pure fragments.

However, to recover the same convenience in a language built from fragments, we need to think fragment-wise again. We must take apart the grammar of evaluation contexts so that each fragment can define its part. For our language, the full grammar would look as follows:

$$K ::= \bullet \mid e + K \mid K + v \mid e \le K \mid K \le v \mid \texttt{if}\ K\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid$$
$$e_1(K) \mid K(v) \mid \texttt{ref}(K) \mid\ !K \mid e \leftarrow K \mid K \leftarrow v$$

Instead of applying the entire "Data types à la Carte" machinery again, we can actually get away with something much simpler: observe that all of the productions in the grammar of evaluation contexts have exactly one recursive occurrence of K, except for the hole $\bullet$. In other words, this is essentially a list![4] We call the individual elements that the list can contain (*e.g.*, $e + \bullet$) *evaluation context items*. To decompose the grammar for K, we simply let each fragment pick a type of evaluation context items, parameterized as usual by the expression and value types for the entire language:

$$\mathbf{K}_{\text{INT}}(E, V) ::= e + \bullet \mid \bullet + v \qquad (e \in E, v \in V)$$
$$\mathsf{fill}_{\text{INT}} : \forall \mathcal{L}.\, \textsc{val} \sqsubseteq_{\text{syn}} \mathcal{L} \to \mathbf{K}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}}) \to \mathbf{E}_{\mathcal{L}} \to \mathbf{E}_{\text{INT}}(\mathbf{E}_{\mathcal{L}}, \mathbf{V}_{\mathcal{L}})$$

Alongside this type, the fragment also defines a corresponding fill function that fills the hole of an evaluation context item with the given expression. Then we define the evaluation context item type for an entire language as a big sum type of all the individual fragments' context item types, and finally $\mathbf{K}_{\mathcal{L}}$ is defined as a list over that sum type. The language-level fill operation is defined by folding the fragment's fill operation over that list. This lets us then state WpBind by taking $K \in \mathbf{K}_{\mathcal{L}}$.

Remaining fragments. Most of the remaining fragments are pure, so their rules are very similar to those for INT. They can be found in Figure 1. The one interesting case that remains is HEAP. Our goal is to obtain the usual separation logic proof rules for heap accesses:

$$\textsc{WpAlloc}\ (\textsc{heap} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell)}{\mathrm{wp}_{\mathcal{L}}\, \texttt{ref}(v)\, \{\Phi\}} \qquad \textsc{WpLoad}\ (\textsc{heap} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \Phi(v))}{\mathrm{wp}_{\mathcal{L}}\, !\ell\, \{\Phi\}} \qquad \textsc{WpStore}\ (\textsc{heap} \sqsubseteq_{\text{sem}} \mathcal{L})\quad \frac{\ell \mapsto v' * (\ell \mapsto v \mathrel{-\!\!*} \Phi(()))}{\mathrm{wp}_{\mathcal{L}}\, \ell \leftarrow v\, \{\Phi\}}$$

These rules are written in weakest precondition style but correspond to the standard Hoare triple rules. To see how, let us take WpStore as an example. Imagine we are proving $\mathrm{wp}_{\mathcal{L}}\, e\, \{\Phi\}$ and the next instruction that executes in e is a store. To make progress, we first use WpBind to "focus" on the store. That turns the goal into $\mathrm{wp}_{\mathcal{L}}\, \ell \leftarrow v\, \{\mathrm{wp}_{\mathcal{L}}\, e'\, \{\Phi\}\}$ where e' is the rest of the program after the store. We can now apply WpStore and have to prove its premise. This is a separating conjunction, which means we have to split our current context into (a) a part that satisfies $\ell \mapsto v'$

[4]This is how evaluation contexts are often defined in Iris-based languages.

for some arbitrary v', and (b) the rest, for which we have to prove $\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{L}}\, e' \{\Phi\}$. This means we get to add $\ell \mapsto v$ to our remaining context, and then we can continue verifying e'. In other words, WpStore forced us through the usual reasoning for stores: give up ownership of ℓ with the old value, and then re-obtain ownership with the new value.

To be able to define these proof rules in a fragment, we make crucial use of Iris's flexible notion of ghost state. The fragment sets up its own ghost state that tracks the current contents of the heap and defines a notion of $\mapsto$ that is tied to that ghost state.

2.4 Soundness

Having introduced the program logic for our fragments and the composed language, we now turn our attention to justifying why that logic is sound. As already mentioned, we will take a semantic approach. This means that we need to give a model for our notion of $\mathsf{wp}_{\mathcal{L}}$ and then show that all our rules hold as theorems about that model.

To build said model, we use the approach described in "Program logics à la Carte" [34]. That paper defined a weakest precondition connective for ITrees with an arbitrary event type, insofar as there is a corresponding *logical effect handler*. The logical effect handler in essence states the pre- and postcondition for each event. We therefore assume that each event type used by a fragment comes with such a logical effect handler, and we compose all those handlers into a single handler $H_{\mathcal{L}}$ for the entire language. This enables us to *define*:

$$\mathsf{wp}_{\mathcal{L}}\, e\, \{\Phi\} := \mathsf{wpi}_{H_{\mathcal{L}}}\, [\![e]\!]_{\mathcal{L}}\, \{\Phi\} \tag{LangWp}$$

We require that every fragment justifies its proof rules according to this definition. For example, this means that rules such as WpStore are proven according to this model and, like the rule, this proof applies to every language containing the heap fragment. These proofs use the underlying ITree program logic [34].

Soundness of WpBind. In order for WpBind to be sound, we need to impose a semantic condition on our evaluation context items K. Namely, the denotation has to satisfy the "semantic bind rule":

$$[\![K[e]]\!] \approx v \leftarrow [\![e]\!]; [\![K[v]]\!].$$

This means that in order to execute $K[e]$, one must first execute e to a value v and then execute the remainder $K[v]$. Every fragment frag is required to prove this property for its evaluation context items K and fill function fill and for any language $\mathcal{L}$ such that $\textsc{frag} \sqsubseteq_{\mathsf{sem}} \mathcal{L}$. Using this and the bind rule [34, Fig. 3] for wpi, WpBind can then be proven for all Mosaic languages.

Soundness of wpi. We have reduced the soundness of $\mathsf{wp}_{\mathcal{L}}$ to that of wpi. Since the latter was established sound in prior work [34], this settles the question of soundness.

That said, typical Iris-style program logics prove soundness in terms of *operational semantics*. We shall see in §4 how to obtain those same guarantees for $\mathsf{wp}_{\mathcal{L}}$.

2.5 Summary: What is a fragment?

Let us summarize what a fragment is. A fragment frag provides:

(1) expressions $\mathbf{E}_{\textsc{frag}}(E, V)$ and values $\mathbf{V}_{\textsc{frag}}(E, V)$,
(2) a substitution function $\mathsf{subst}_{\textsc{frag}}$,
(3) a denotation $[\![\cdot]\!]_{\textsc{frag}}$ (which can declare dependencies on other fragments or on ITree effects),
(4) evaluation context items $\mathbf{K}_{\textsc{frag}}(E, V)$ and a filling function $\mathsf{fill}_{\textsc{frag}}$ satisfying the semantic bind rule, and
(5) program logic rules with a proof of their soundness against the LangWp model.

Multiple fragments can be composed into a language $\mathcal{L}$ with expressions $\mathbf{E}_{\mathcal{L}}$, values $\mathbf{V}_{\mathcal{L}}$, a denotation $[\![\cdot]\!]_{\mathcal{L}}$, evaluation context items $\mathbf{K}_{\mathcal{L}}$, and a sound program logic.

3 Concurrency

The previous section considered a sequential language. This section shows how Mosaic supports concurrency. First, we describe how the framework itself needs to be adjusted for concurrency (§3.1). Then, we present two new fragments: SPAWN for spawning new threads (§3.2) and ATOMIC for atomic blocks (§3.3). These fragments can be freely (and soundly) imported alongside other fragments to create a concurrent language, such as the case studies we consider in §7.

3.1 Extending Mosaic for Concurrency

Concurrency has global ramifications: Each fragment should not only determine how the outputs of each operation are computed, but also describe how *atomic* each operation is, *i.e.,* when the scheduler is allowed to interrupt an operation to schedule a different thread.

To encode this atomicity information, we follow the approach of Vistrup et al. [34]. The ITree semantics of each fragment can (and should) insert calls to a yield function that marks the points where the scheduler can switch to another thread. We follow the scheme for inserting yield calls described in [34, §4.1]: in particular, we redefine the call $\mathsf{eval}(e)$ to implicitly be followed by a yield whenever e is not a value.[5]

Picking the scheduler. What is different from Vistrup et al. [34] is that we do *not* use a fixed interpretation of the yield operation. Instead, each language globally picks its own semantics for yield. To get a standard non-deterministic thread pool semantics, one can define $\mathsf{yield} := \mathsf{yield}_{\mathbf{ConcE}}$ where **ConcE** is the thread pool event type [34]. However, one can also pick $\mathsf{yield} := \mathsf{Ret}()$ to obtain a sequential language. We shall see a third choice of yield in §3.3.

Integrating concurrency reasoning into the program logic. To extend our program logic with support for reasoning about concurrent programs, we add support for Iris invariants [17]. An Iris invariant $\boxed{P}^{\mathcal{N}}$ asserts that P holds whenever control switches from one thread to another, allowing threads to share resources. Invariants are integrated into the program logic by equipping $\mathsf{wp}_{\mathcal{L}}$ with a mask $\mathcal{E}$ that describes the set of currently available invariants. We provide the following rule for opening an invariant around an expression e and accessing its contents:

$$\textsc{WpInv}\ \frac{P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{L},\mathcal{E}\setminus\mathcal{N}}\, e\, \{r.\, P * \Phi(r)\} \qquad \boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\mathsf{wp}_{\mathcal{L},\mathcal{E}}\, e\, \{\Phi\}}$$

Given an invariant $\boxed{P}^{\mathcal{N}}$, WpInv gives access to P as long as the name $\mathcal{N}$ of the invariant is available ($\mathcal{N} \subseteq \mathcal{E}$) and removes $\mathcal{N}$ from $\mathcal{E}$ during e. The absence of an atomicity side condition is inherited from wpi [34, §4.2]. Atomicity of e is instead checked by the bind rule, as we discuss next.

Adapting bind. WpInv is only one half of reasoning about invariants. We also need to ensure that all invariants hold when control switches between threads. That is, when a fragment invokes the yield function, the mask must be $\top$. Thus, non-atomic code can only be verified at $\top$ mask. In

[5]Alongside yield, there is also another event that every fragment invokes: step. This event demarcates what counts as a computational step and thus controls what executions are considered non-terminating. In particular, it determines where in the program logic later modalities ▷ appear. However, it has no bearing on the present discussion so we shall ignore it.

the logic, this shows up in the bind rule:

$$\textsc{WpBindConc}\ \frac{\mathsf{yield} = \mathsf{yield}_{\mathbf{ConcE}} \quad \mathsf{wp}_{\mathcal{L},\top}\, e\, \{v.\, \mathsf{wp}_{\mathcal{L},\top}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L},\top}\, K[e]\, \{\Phi\}} \qquad \textsc{WpBindSeq}\ \frac{\mathsf{yield} = \mathsf{Ret}() \quad \mathsf{wp}_{\mathcal{L},\mathcal{E}}\, e\, \{v.\, \mathsf{wp}_{\mathcal{L},\mathcal{E}}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L},\mathcal{E}}\, K[e]\, \{\Phi\}}$$

With concurrency enabled, the bind rule can only be used with the $\top$ mask since $K[e]$ may be a non-atomic operation. But in the sequential case, the bind rule is available at any mask $\mathcal{E}$.

These changes to the logic are a reflection of changes to the semantics, specifically our aforementioned adjustments to eval. Recall from §2.4 that evaluation contexts K were required to satisfy the semantic bind rule: $[\![K[e]]\!] \approx v \leftarrow [\![e]\!]; [\![K[v]]\!]$. However, in the concurrent setting, another thread may temporarily take control between executing e and $K[v]$. We update the semantic bind rule to account for this: Every fragment has to prove that for all of its evaluation context items K,

$$[\![K[e]]\!] \approx v \leftarrow [\![e]\!]; \mathsf{yield\_if\_not\_val}(e); [\![K[v]]\!].$$

Here, $\mathsf{yield\_if\_not\_val}(v) = \mathsf{Ret}()$, whereas $\mathsf{yield\_if\_not\_val}(e) = \mathsf{yield}$ when e is not a value. In other words, evaluating $K[e]$ is equivalent to first evaluating e to v and *then* evaluating $K[v]$. If e is not already evaluated (*i.e.,* it is not a value), we allow the scheduler to switch control before continuing with the remaining computation $K[v]$.

3.2 Spawn Fragment

Our first concurrency-specific fragment is the SPAWN fragment for spawning new threads. It provides the `spawn {e}` expression:

$$\mathbf{E}_{\textsc{spawn}}(E, V) ::= \mathtt{spawn}\ \{e\} \qquad (e \in E)$$

Its semantics require the concurrency event **ConcE** to be available among the global events since **ConcE** provides the spawn function, which `spawn {e}` relies on to spawn new threads:

$$[\![\mathtt{spawn}\ \{e\}]\!]_{\textsc{spawn}} := \mathtt{spawn}(\mathsf{eval}(e); \mathsf{Ret}()); \mathsf{Ret}()$$

The fragment comes with the standard separation logic rule, and a derived rule demonstrating the usual disjoint splitting of resources that is typical for separation logics:

$$\textsc{WpSpawn}\ \frac{\mathsf{wp}_{\mathcal{L},\top}\, e\, \{\mathsf{True}\}}{\mathsf{wp}_{\mathcal{L},\mathcal{E}}\, \mathtt{spawn}\ \{e\}\ \{v.\, v = ()\}} \qquad \textsc{WpPar}\ \frac{\mathsf{wp}_{\mathcal{L},\top}\, e\, \{\mathsf{True}\} * \mathsf{wp}_{\mathcal{L},\top}\, e'\, \{\Phi\}}{\mathsf{wp}_{\mathcal{L},\top}\, \mathtt{spawn}\ \{e\}; e'\, \{\Phi\}}$$

3.3 Atomic Block Fragment

Our second concurrency-specific fragment is ATOMIC, which introduces *atomic blocks*. Atomic blocks allow executing a number of instructions "in one go" without yielding to another thread. They are useful for modeling atomic operations available on some hardware such as compare-and-swap and fetch-and-X,[6] which can then be written in terms of the more primitive heap operations of load and store. Specifically, the ATOMIC fragment adds one expression:

$$\mathbf{E}_{\textsc{atomic}}(E, V) ::= \mathtt{atomic}\ \{e\} \qquad (e \in E)$$

The behavior of `atomic {e}` is reflected in its program logic rules:

$$\textsc{WpAtomic}\ \frac{☢ \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{L},\mathcal{E}}\, e\, \{v.\, ☢ * \Phi(v)\}}{\mathsf{wp}_{\mathcal{L},\mathcal{E}}\, \mathtt{atomic}\ \{e\}\ \{\Phi\}} \qquad \textsc{WpAtomicBind}\ \frac{\mathsf{yield} = \mathsf{yield}_{\textsc{atomic}} \quad \mathsf{wp}_{\mathcal{L},\mathcal{E}}\, e\, \{v.\, (\mathcal{E} = \top \vee ☢) \wedge \mathsf{wp}_{\mathcal{L},\mathcal{E}}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}_{\mathcal{L},\mathcal{E}}\, K[e]\, \{\Phi\}}$$

[6] In real languages, there exist many variants of this: fetch-and-add, fetch-and-max, fetch-and-xor, …

Executing an atomic block gives access to the ☢ token during the execution of its body e (WpAtomic). This ☢ token is used by the atomic bind rule WpAtomicBind. The latter rule requires atomic blocks to be enabled: yield = $\text{yield}_{\text{ATOMIC}}$ (we shall return to this). WpAtomicBind supports two use-cases (via the disjunction): Either the mask $\mathcal{E}$ is $\top$, in which case the rule is equivalent to the normal concurrent bind rule WpBindConc. Or one can exhibit a ☢ token after e has executed to show that an atomic block is active (thus blocking the possibility of yielding to another thread). In this case, there is no restriction on the mask $\mathcal{E}$ and the rule behaves like the sequential bind rule WpBindSeq, allowing one to keep invariants open while the atomic block is active. The non-separating conjunction obviates the need to give up the ☢ token to use WpAtomicBind.

Semantics of atomic blocks. To give semantics to atomic blocks, we maintain a global counter of open atomic blocks. The counter is incremented when entering an atomic block and decremented when leaving. The atomic yield operation $\text{yield}_{\text{ATOMIC}}$ only yields to another thread if no atomic block is active, *i.e.,* the counter is zero.

The operations for incrementing/decrementing/accessing the atomic counter are encoded using the **StateE** event [34, §3.3]. The atomic token is defined using the ghost theory of the **StateE** event, which we also use to prove the program logic rules above. Here too, we follow the paradigm of building semantics and program logic from more primitive abstractions, reusing their proof rules.

4 From Operational Semantics to ITrees

In this section, we will show that Mosaic-based program logics can provide the very same verification guarantees as traditional program logics. Traditionally, semantics are phrased in terms of operational semantics, which is why we connect the ITree semantics of Mosaic back to the operational semantics that typically underpin program logics. To this end, we present the *compatibility theorem* (Theorem 4.4). This theorem allows us to transform an operational semantics execution into an ITree execution, which can in turn be fed into our soundness statement (Theorem 4.1 and Theorem 5.1). The end result is a soundness result for Mosaic that matches the soundness statement of traditional program logics (Corollary 4.5). We use this result to demonstrate that our program logics for HeapLang [15] and λ_{Rust} [16] provide the same guarantees as the original logics.

Before we continue, we give some background on the Iris language interface (§ 4.1) and on big-step semantics of ITrees (§ 4.2), which are our two reference notions of semantics.

4.1 Background: Iris Language Interface

To state the compatibility theorem, we need a notion of operational semantics. There are many variants of operational semantics that have been used by program logics. In this paper, we use the notion of operational semantics defined by the Iris language interface [31, §8]. We chose this notion of operational semantics because of its widespread use in the Iris ecosystem [16, 13, 18, 28, 14].

The bedrock of the Iris language interface is the *base-step* relation: $(e_1, \sigma_1) \to_{\text{b}} (e_2, \sigma_2, \vec{e}_f)$. A base-step says that the expression e_1 in state σ_1 can step to expression e_2 in state σ_2, spawning new threads with expressions $\vec{e}_f$. As an example, for the language of § 2, the rule for beta reduction would say: $((\lambda x.\, e)(v), \sigma) \to_{\text{b}} (e[v/x], \sigma, [])$.

Base steps are lifted to a *thread step* relation ($\to_{\text{t}}$) and then to a *thread pool step* relation ($\to_{\text{tp}}$):

$$\textsc{ThreadStep}\quad \frac{(e_1, \sigma_1) \to_{\text{b}} (e_2, \sigma_2, \vec{e}_f)}{(K[e_1], \sigma_1) \to_{\text{t}} (K[e_2], \sigma_2, \vec{e}_f)} \qquad \textsc{TpStep}\quad \frac{(e_1, \sigma_1) \to_{\text{t}} (e_2, \sigma_2, \vec{e}_f)}{(\vec{e}_L :: e_1 :: \vec{e}_R, \sigma_1) \to_{\text{tp}} (\vec{e}_L :: e_2 :: \vec{e}_R :: \vec{e}_f, \sigma_2)}$$

$K[e]$ denotes "filling" the hole $\bullet$ in K with an expression e. ThreadStep says that thread steps ($\to_{\text{t}}$) close base-steps ($\to_{\text{b}}$) under evaluation contexts: a thread step corresponds to a base-step inside an

evaluation context K. TpStep shows how $(\rightarrow_{\mathsf{tp}})$ lifts $(\rightarrow_{\mathsf{t}})$ to thread pools by picking an expression e_1 in the thread pool to perform a thread step and appending the newly spawned threads $\vec{e}_f$.

We call a thread *safe* in a given state if it either can take a step, or is already a value and therefore finished executing. Conversely, a thread is *stuck* if it is not safe, *i.e.,* if it cannot take a step and is not a value. A stuck thread represents a buggy execution, such as a dangling memory access. For thread pools, we require *all* threads to be safe, or conversely: $\mathsf{stuck}_{\mathsf{tp}}(\vec{e}, \sigma) := \exists e \in \vec{e}.\, \mathsf{stuck}(e, \sigma)$. In other words, a thread pool is stuck if *any* thread is stuck.

Finally, we can state the notion of safety proven by program logics based on the Iris language interface: e is *operationally adequate* w.r.t. postcondition $\phi : \mathsf{val} \to \mathsf{Prop}$ if

(1) for any $\sigma_1, \sigma_2, \vec{e}_2$ such that $([e], \sigma_1) \rightarrow^*_{\mathsf{tp}} (\vec{e}_2, \sigma_2)$, we have $\neg\mathsf{stuck}_{\mathsf{tp}}(\vec{e}_2, \sigma_2)$, and
(2) for any $\sigma_1, \sigma_2, v, \vec{e}_2$ such that $([e], \sigma_1) \rightarrow^*_{\mathsf{tp}} (v :: \vec{e}_2, \sigma_2)$, we have $\phi(v)$.

4.2 Background: Big-Step Semantics for ITrees

Our objective is to compare operational semantics to ITree semantics. To even state such a result, we will need to explain what the *executions* of an ITree are.

To this end, we rely on the big-step semantics for ITrees introduced by Vistrup et al. [34, §5]. Concretely, they pick a type of states Σ_E and define the following big-step semantics:

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

$(t, \sigma) \Downarrow T$ states that the ITree t in state σ can perform a big-step to an ITree t' and state σ' such that $(t', \sigma') \in T$. Using a set T instead of a single final state follows the style of Omnisemantics [6]. In particular, $(t, \sigma) \Downarrow \emptyset$ if and only if t fails (*i.e.,* triggers Undefined Behavior).

To obtain the big-step semantics of an entire program, we begin by defining an execution handler for each event:

$$(\rightsquigarrow_E) : \forall A.\, E\,A \to \Sigma_E \to (A \to \Sigma_E \to \mathsf{Prop}) \to \mathsf{Prop}$$

This handler describes how the events affect the per-event state Σ_E. For example, the execution handler for the **HeapE**, **DemonicE**, and **FailE** are given as follows:

$$(\mathsf{load}(\ell), \sigma) \rightsquigarrow_{\mathbf{HeapE}} \{(\sigma[\ell], \sigma)\} \qquad (\mathsf{store}(\ell, v), \sigma) \rightsquigarrow_{\mathbf{HeapE}} \{((), \sigma[\ell \mapsto v])\}$$

$$(\mathsf{choice}_A, ()) \rightsquigarrow_{\mathbf{DemonicE}} \{(x, ())\} \qquad (\mathsf{fail}, ()) \rightsquigarrow_{\mathbf{FailE}} \emptyset$$

Σ_E is just the unit type for **DemonicE** and **FailE** as those events have no state. Execution handlers can be composed to obtain handlers for compound event types such as $(\rightsquigarrow_{\mathbf{DemonicE} \oplus \mathbf{FailE}})$. Using $(\rightsquigarrow_E)$, *loc. cit.* defines $(t, \sigma) \Downarrow_E T$ by (co-inductively) applying the corresponding $(\rightsquigarrow_E)$ to each visible event of t. In this way, one can compositionally obtain big-step semantics for ITrees.

Execution handler for concurrency. For expository purposes, the presentation of execution handlers $(\rightsquigarrow_E)$ above is slightly simplified. A more general definition allows defining a handler $(\rightsquigarrow_{\mathbf{ConcE}})$ for the concurrency event **ConcE**. For this, we need the state $\Sigma_{\mathbf{ConcE}}$ to encode the thread pool as a list of suspended threads: $\mathit{List}(\mathbf{itree}\ E\ R)$.

Soundness of wpi. What does the program logic of ITrees prove about executions? The colloquial meaning of $\mathsf{wpi}_H\ t\ \{\Phi\}$ is understood as "every execution of t goes well and the return value satisfies Φ". Soundness amounts to making this imprecise statement precise and proving it.

The connection between executions and program logic is made on the level of event types by the soundness predicate $\mathsf{sound}(H, I, \rightsquigarrow_E)$ that relates a logical effect handler with an execution handler. Informally, it says that H is sound w.r.t. the small-step $(\rightsquigarrow_E)$ while maintaining the invariant $I(\sigma)$ on the state $\sigma \in \Sigma_E$. Crucially, this property is preserved by $\oplus$: the composition of logical effect handlers is sound w.r.t. the composition of the corresponding execution handlers. We

omit the precise definition of $\mathsf{sound}(H, I, \rightsquigarrow_E)$ for brevity. Instead, we focus on the following key consequence, which tells us that wpi_H is sound w.r.t. big-steps ($\Downarrow_E$):

Theorem 4.1 (Soundness of wpi).

$$\frac{\mathsf{sound}(H, I, \rightsquigarrow_E) \qquad (t, \sigma) \Downarrow_E T \qquad I(\sigma) * \mathit{wpi}_H\, t\, \{\Phi\}}{\exists t', \sigma'.\ (t', \sigma') \in T * I(\sigma') * \mathit{wpi}_H\, t'\, \{\Phi\}}$$

This statement says that if we have a big-step $(t, \sigma) \Downarrow_E T$ with $I(\sigma)$ and $\mathsf{wpi}_H\, t\, \{\Phi\}$, we can obtain a final ITree t' and state σ' (*i.e.,* $(t', \sigma') \in T$) that also satisfy $I(\sigma')$ and $\mathsf{wpi}_H\, t'\, \{\Phi\}$. For the special case $T = \{(\mathsf{Ret}(v), \sigma')\}$, this means that if we provide a big-step execution of t resulting in a final value v and prove $\mathsf{wpi}_H\, t\, \{\Phi\}$, we obtain $\Phi(v)$. For the special case $T = \emptyset$, we can prove a contradiction from $\mathsf{wpi}_H\, t\, \{\Phi\}$ (since the empty set is empty). This shows that $\mathsf{wpi}_H\, t\, \{\Phi\}$ rules out all failures/Undefined Behavior in t. This is the sense in which wpi_H is sound.

4.3 Compatibility Theorem

With this background established, we now turn to the compatibility theorem. The basic idea is as follows: We first define a notion of *derived steps* ($\rightarrowtail$), which provides an analogue to thread steps for our fragments. Then, we prove the compatibility theorem, which roughly states that if each base-step of the original language has a corresponding derived step, we can lift thread pool executions from §4.1 to the ITree big-step execution of §4.2.

Derived steps. We start by defining derived steps ($\rightarrowtail$). These are described in terms of executions ($\Downarrow$) of the ITree denotations. $(e_1, \sigma_1) \rightarrowtail (e_2, \sigma_2, \vec{e}_f)$ should be understood as saying that e_1 in state σ_1 steps to e_2 in state σ_2 spawning threads $\vec{e}_f$. It is defined as follows:[7]

$$\begin{aligned} &\forall T, k, \rho.\ (\mathsf{spawn}(\llbracket \vec{e}_f \rrbracket); \mathsf{yield\_if\_not\_val}(e_2); v \leftarrow \llbracket e_2 \rrbracket; k(v), \sigma_2[\mathsf{tp} := \rho]) \Downarrow T \\ &\quad \Longrightarrow\ (v \leftarrow \llbracket e_1 \rrbracket; k(v), \sigma_1[\mathsf{tp} := \rho]) \Downarrow T \end{aligned}$$

The definition encodes the intuitive meaning by giving us a way to construct an execution of e_1 from an execution of e_2. Namely, it states that an ($\Downarrow$) execution of the denotation of e_1 with continuation k in the state σ_1 with thread pool ρ can be constructed from an ($\Downarrow$) execution in the state σ_2 with the same thread pool of the ITree that represents what remains to be done after the step: (1) spawning the new threads $\vec{e}_f$, (2) yielding if e_2 is not a value (*i.e.,* if there is still more to do), (3) continuing with e_2, and (4) finally invoking the continuation k. The definition universally quantifies over the thread pool ρ, reflecting that the step should apply no matter what threads are waiting in the background. It is also closed under an ITree continuation k, because that allows us to use the semantic bind rule to prove that $\rightarrowtail$ is closed under evaluation contexts:

Lemma 4.2 (Context closure). *Let K be an evaluation context as defined in §3.1. Then*

$$(e_1, \sigma_1) \rightarrowtail (e_2, \sigma_2, \vec{e}_f) \implies (K[e_1], \sigma_1) \rightarrowtail (K[e_2], \sigma_2, \vec{e}_f).$$

In this sense ($\rightarrowtail$) can be viewed as an analogue of ($\rightarrow_{\mathrm{t}}$).

Stuckness. Alongside a step relation ($\rightarrowtail$), we also need a notion of a program "going wrong". Under operational semantics, this role is played by $\mathsf{stuck}(e, \sigma)$ which says that (e, σ) has no expression to step to. However, under the big-step semantics for ITrees, this role is instead played by executing to $\emptyset$, which leads us to define:

$$\mathsf{unsafe}(e, \sigma) := (\forall k, \rho.\ (v \leftarrow \llbracket e \rrbracket; k(v), \sigma[\mathsf{tp} := \rho]) \Downarrow \emptyset).$$

Again, the continuation k ensures that unsafe is closed under evaluation contexts:

[7]Here, tp refers to the $\Sigma_{\mathbf{ConcE}}$ part of the state, *i.e.,* the state of ($\rightsquigarrow_{\mathbf{ConcE}}$). See the discussion of concurrency in §4.2.

Lemma 4.3. *Let K be an evaluation context and e a non-value. Then $\mathit{unsafe}(e, \sigma) \implies \mathit{unsafe}(K[e], \sigma)$.*

Compatibility theorem. Finally, we can state the compatibility theorem. The setup is as follows. We have a concurrent language $\mathcal{L}$ with an operational semantics defined using the Iris language interface (§4.1) and a concurrent[8] language $\mathcal{L}_f$ composed from fragments (§2). We want to prove that $\mathcal{L}_f$ simulates $\mathcal{L}$. For this purpose, we require a translation function $\downarrow$ that maps expressions of $\mathcal{L}$ to expressions of $\mathcal{L}_f$, and states of $\mathcal{L}$ to states of $\mathcal{L}_f$ (with empty thread pool). Under this setup, the compatibility theorem says:

Theorem 4.4 (Compatibility theorem). *Given $\mathcal{L}$, $\mathcal{L}_f$, $\downarrow\cdot$, and assuming for all expressions and states of $\mathcal{L}$ that*

(1) (Stuck preservation.) $\mathit{stuck}_{\mathrm{b}}(e, \sigma) \implies \mathit{unsafe}(\downarrow e, \downarrow \sigma)$.
(2) (Simulation.) $(e_1, \sigma_1) \to_{\mathrm{b}} (e_2, \sigma_2, \vec{e}_f) \implies (\downarrow e_1, \downarrow \sigma_1) \rightarrowtail (\downarrow e_2, \downarrow \sigma_2, \downarrow \vec{e}_f)$.

Then the following holds:

(1) If $([e_1], \sigma_1) \to_{\mathrm{tp}}^* (\vec{e}_2, \sigma_2)$ *and* $\mathit{stuck}_{tp}(\vec{e}_2, \sigma_2)$, *then* $([\![\downarrow e_1]\!], \downarrow \sigma_1) \Downarrow \emptyset$.
(2) If $([e_1], \sigma_1) \to_{\mathrm{tp}}^* (v :: \vec{e}_2, \sigma_2)$, *then* $([\![\downarrow e_1]\!], \downarrow \sigma_1) \Downarrow \{(\mathit{Ret}(\downarrow v), \downarrow \sigma_2[tp := [\![\downarrow \vec{e}_2]\!]])\}$.

Here, $\mathrm{stuck}_{\mathrm{b}}(e, \sigma)$ means that (e, σ) is stuck w.r.t. base-step ($\to_{\mathrm{b}}$). Intuitively, this theorem builds an ($\Downarrow$) execution out of an ($\to_{\mathrm{tp}}^*$) execution by simulating one ($\to_{\mathrm{tp}}$) step at a time. In particular, this theorem can be combined with the soundness result of the $\mathcal{L}_f$ program logic to obtain the soundness result typically proven when using the Iris language interface directly:

Corollary 4.5. *Assume the conditions of Theorem 4.4. If* $\mathrm{wp}_{\mathcal{L},\top} \downarrow e \{\phi\}$, *then e is operationally adequate w.r.t. ϕ (§4.1).*

Using Theorem 4.4. To discharge the assumptions of Theorem 4.4 in a compositional manner, we make use of the fragmented nature of $\mathcal{L}_f$: Each fragment library provides various lemmata about ($\rightarrowtail$) and $\mathrm{unsafe}(e, \sigma)$ for the expressions added by that fragment. For example, $((\lambda x.\, e)v, \sigma) \rightarrowtail (e[v/x], \sigma, [])$ holds for any language that contains the FUN fragment. These lemmata can be composed to establish the assumptions of Theorem 4.4. We demonstrate that this approach scales to complex languages by applying it to the HeapLang [15] and λ_{Rust} [16] languages (see §7). These proofs amount to case-by-case analysis where each of the many cases discharges its goal to a lemma in the corresponding fragment library.

5 Prophecy Variables

In the study of concurrent programs, prophecy variables [1, 2] are an important reasoning device that allows one to, at any point in the proof, peek into the future of the execution. This is crucial for proofs of linearizability that rely on "future-dependent" linearization points. However, prophecy variables are a demanding feature for any model: it is easy to accidentally introduce logical inconsistency via circular reasoning [18, §1.1]. In Iris, HeapLang has been equipped with support for prophecy variables [18], but that implementation is specialized to HeapLang, in a way that unfortunately affects even the development of unrelated HeapLang extensions. For this reason, some HeapLang forks such as GooseLang [5, 29] have partially removed support for prophecy variables, and older forks such as λ_{Rust} [16] have never been equipped with prophecy variables.

In this section, we show that we can make prophecy variables a fragment in Mosaic, which means they can be added to any Mosaic language without causing any extra burden for other fragments. But first, we have to recapitulate how prophecy variables work in HeapLang.

[8] By "concurrent", we mean yield = $\mathrm{yield}_{\mathbf{ConcE}}$ or yield = $\mathrm{yield}_{\mathrm{ATOMIC}}$. See §3.

$$\textsc{WpNewProph}\ \frac{\forall p, vs.\ \mathsf{proph}(p, vs) \mathrel{-\!\!*} \Phi(p)}{\mathsf{wp}\ \texttt{new\_proph}\ \{\Phi\}} \qquad \textsc{WpResolve}\ \frac{\mathsf{proph}(p, vs) \qquad \forall vs'.\ vs = v :: vs' \to \mathsf{proph}(p, vs') \mathrel{-\!\!*} \Phi(v)}{\mathsf{wp}\ \texttt{resolve}(p, v)\ \{\Phi\}}$$

Fig. 2. Program logic rules for prophecy variables

5.1 Background: Prophecy Variables in HeapLang

In HeapLang, prophecy variables are implemented by extending the language with special operations to manage prophecies.[9] Specifically, we need two such operations: `new_proph` to create a fresh prophecy variable p, and $\texttt{resolve}(p, v)$ to associate a value v with a previously created prophecy variable p. This is also called "resolving" the prophecy variable to a value (and can occur more than once). The idea is that *logically*, we learn about all the values associated with a prophecy variable the moment it is created. This is captured by the specifications shown in Figure 2: The postcondition of `new_proph` provides ownership of a ghost token $\mathsf{proph}(p, vs)$, where vs is the list of "prophesied" values, *i.e.,* the values that will be associated with p in the remainder of the program's execution. On the other hand, the postcondition of `resolve` assures us that the value v we just passed to `resolve` is indeed, as prophesied, the head of the list vs.

For some proofs, it is necessary to be able to resolve a prophecy *alongside* an atomic operation, together in a single step without yielding to another thread. To achieve this, Jung et al. [18, §2.4] slightly extend the `resolve` operation by adding another argument: $\texttt{resolve}(e_1, e_2, e_3)$ will evaluate e_3 to some value v, e_2 to some prophecy variable p, and then execute e_1 while resolving p to v. Crucially, this resolution occurs immediately after the execution of e_1, in the same atomic step.

Establishing soundness of prophecy variables requires two separate proofs: First of all, one obviously has to prove soundness of the program logic rules. However, this is not enough: this soundness proof will apply to a program that has been decorated with `new_proph` and `resolve`. Meanwhile, the actual, real program we care about, of course, does not contain those "ghost" operations. Jung et al. [18] hence prove an *erasure theorem* which establishes that `new_proph` and `resolve` can be removed from the verified program (replacing all prophecy variables with "poison" values) without invalidating the results of verification.

All this machinery significantly complicates the theory of HeapLang, in a way that affects other, orthogonal aspects of the language. This is especially true for the operation of "resolving alongside an atomic operation", so it is not surprising that HeapLang forks tend to remove this operation to make it easier for them to adjust the language to their needs [29]. Moreover, to add prophecy variables to a language that is not HeapLang, one will have to redo the entirety of the complicated proof of the erasure theorem. This is a painful part of the status quo that motivates Mosaic.

5.2 Prophecy Variables as a Fragment

In the following, we will explain how prophecy variables can be considered as yet another fragment. To ensure soundness, we shall later state and prove a *language-agnostic* erasure theorem once and for all. This makes it effortless for anyone to add prophecy variables to their Mosaic language.

The prophecy variable fragment introduces one kind of value and two kinds of expressions:

$$\mathbf{V}_{\text{PROPH}}(E, V) ::= p \qquad (p \in \mathbb{N})$$
$$\mathbf{E}_{\text{PROPH}}(E, V) ::= \texttt{new\_proph} \mid \texttt{resolve}(e_1, e_2) \qquad (e_1, e_2 \in E)$$

[9] This is rather unusual in the context of Iris; typically, new reasoning principles are introduced entirely logically without impacting the language. For prophecy variables, however, language-level annotations are necessary for soundness.

These operations and their proof rules behave as discussed above, with the one difference that in our framework, vs is an *iterator* (a list that is possibly infinite). This reflects that p can be resolved an arbitrary number of times, even an endless number of times.

Semantics. To define the semantics for the prophecy variable fragment, we introduce the $\mathbf{ProphE}_V$ event type. It has two operations:

$$\begin{aligned} &\mathsf{new\_proph} : \mathbf{itree}\ \mathbf{ProphE}_V\ \mathbb{N} \\ &\mathsf{resolve}(p, v) : \mathbf{itree}\ \mathbf{ProphE}_V\ () \qquad\qquad (p \in \mathbb{N}, v \in V) \end{aligned}$$

$[\![\cdot]\!]_{\text{PROPH}}$ uses these events in a straightforward way to interpret `new_proph` and `resolve`.

Resolving a prophecy alongside an atomic operation. To accommodate the previously discussed three-argument form $\texttt{resolve}(e_1, e_2, e_3)$, we extend the syntax and denotation accordingly:

$$\begin{aligned} [\![\texttt{resolve}(e_1, e_2, e_3)]\!]_{\text{PROPH}} := v \leftarrow \mathsf{eval}(e_3); v_p \leftarrow \mathsf{eval}(e_2); w \leftarrow \mathsf{eval}_{\mathsf{no\ yield}}(e_1); \\ p \leftarrow \mathsf{to\_proph}(v_p); \mathsf{resolve}(p, v); \mathsf{Ret}(w) \end{aligned}$$

As we discussed in § 3, $\mathsf{eval}(e)$ yields after evaluating e whenever e is not a value. However, this denotation explicitly avoids yielding after $\mathsf{eval}(e_1)$ altogether. This reflects the fact that $\texttt{resolve}(e_1, p, v)$ should resolve the prophecy simultaneously with the last step of e_1.

5.3 Prophecy Variables in ITrees

We have successfully reduced the use of prophecy variables in the language to the use of $\mathbf{ProphE}_V$ in ITrees. However, we are not done yet: since we are now defining a new ITree event type rather than using an existing one, it behooves us to build up the requisite infrastructure for this event type. Specifically, we need to establish how to reason about ITrees that use these events (by defining a logical effect handler), we need to establish how to execute such ITrees (by defining an execution handler), and we need to prove a soundness theorem relating the two.

We shall do so by interpreting $\mathbf{ProphE}_V$ into more familiar, lower-level events, and then lifting the existing handlers for those events via the interpretation to obtain handlers for $\mathbf{ProphE}_V$. As we shall see, there are two distinct ways to interpret $\mathbf{ProphE}_V$.

Interpretation 1: Prophetic interpretation. The first interpretation of prophecy variables uses demonic choice: When allocating a prophecy variable, we "guess" all its future resolutions vs. This amounts to demonically choosing the sequence of future resolutions. At the time of a resolution, the value is tested against this sequence: is it the first element in the sequence? If the two do not cohere, that means the guess was incorrect, so the resolution discards the current execution (empty demonic choice, also known as "No Behavior"). This is always safe to do, so we can simply dismiss this branch at verification time.

Using this idea, we make sense of prophecy variables by interpreting them into familiar events:

$$\begin{aligned} \mathsf{interp}_{\mathsf{prophetic}} &: \forall A.\ \mathbf{ProphE}_V\ A \to \mathbf{itree}\ (\mathbf{DemonicE} \oplus \mathbf{FailE} \oplus \mathbf{HeapE}_{\mathsf{iterator}(V)})\ A \\ \mathsf{interp}_{\mathsf{prophetic}}(\mathsf{new\_proph}) &:= vs \leftarrow \mathsf{choice}_{\mathsf{iterator}(V)}; \mathsf{alloc}(vs) \\ \mathsf{interp}_{\mathsf{prophetic}}(\mathsf{resolve}(p, v)) &:= vs \leftarrow \mathsf{load}(p); \text{if } \exists vs'.\ vs = v :: vs' \text{ then } \mathsf{store}(p, vs') \text{ else } \mathsf{halt} \end{aligned}$$

This is the *prophetic interpretation*. Let us summarize the participating events. **DemonicE** supplies demonic choice choice_A and $\mathsf{halt} := \mathsf{choice}_{\emptyset}$. halt can be thought of as safely terminating/discarding the execution. **FailE** supplies the event fail which (unsafely) crashes the program and is invoked by load and store when given an unallocated prophecy variable. $\mathbf{HeapE}_{\mathsf{iterator}(V)}$ represents an $\mathbb{N}$-indexed heap where each cell contains an iterator of V: it supplies a function $\mathsf{alloc}(vs)$ to allocate a new location p with content vs, $\mathsf{load}(p)$ to load, and $\mathsf{store}(p, vs)$ to store.

Obtaining ProphH by lifting. In order to define the logical effect handler $\mathbf{ProphH}_V$ for $\mathbf{ProphE}_V$, we will use the interpretation $\mathsf{interp}_{\mathsf{prophetic}}$. Recall that a logical effect handler specifies the proof obligation that occurs whenever an event is triggered. For the events from $\mathbf{ProphE}_V$, we define that proof obligation to be $\mathsf{wpi}_H\, \mathsf{interp}_{\mathsf{prophetic}}(e)\, \{\Phi\}$, *i.e.,* the prophecy events have to be verified by verifying their interpretation. We call this construction *lifting* of logical effect handlers:

$$\mathbf{ProphH}_V := (\mathsf{interp}_{\mathsf{prophetic}})^*(\mathbf{DemonicH} \oplus \mathbf{FailH} \oplus \mathbf{HeapH}_{\mathsf{iterator}(V)})$$

The wpi rules for $\mathbf{ProphH}_V$ correspond directly to those in Figure 2. To obtain these, we define $\mathsf{proph}(p, vs)$ as the points-to connective $p \mapsto vs$ of the underlying $\mathbf{HeapH}_{\mathsf{iterator}(V)}$. Using standard proof rules for heaps and demonic choice ([34, Fig. 5 and Fig. 6]), we can then easily derive the rules for wpi matching the ones in Figure 2. Thus, even the wpi rules are *derived* rather than *primitive*. We did not have to unravel layers of definitions or define new ghost theory but could simply recombine what is already there in novel ways, building derived abstractions in terms of much simpler ones.

We can also easily define an execution handler $(\rightsquigarrow_{\mathsf{prophetic}})$ for $\mathbf{ProphE}_V$ by applying $\mathsf{interp}_{\mathsf{prophetic}}$ and reducing the problem to the already existing execution handlers $(\rightsquigarrow_{\mathbf{DemonicE}})$, $(\rightsquigarrow_{\mathbf{FailE}})$, and $(\rightsquigarrow_{\mathbf{HeapE}})$. This construction is called *lifting* of execution handlers. However, this handler does not give rise to a very natural notion of execution. In particular, halt is not an executable operation. Executing halt requires (possibly infinite amounts of) backtracking. And, $\mathsf{interp}_{\mathsf{prophetic}}(\mathsf{new\_proph})$ demands that we pick vs up front, and that choice must be compatible with all the future resolutions lest we encounter a halt.

Interpretation 2: Erased interpretation. While $\mathsf{interp}_{\mathsf{prophetic}}$ tells us how to make sense of $\mathbf{ProphE}_V$ events *logically*, the ITrees produced by $\mathsf{interp}_{\mathsf{prophetic}}$ do not explain how to *execute* prophecy events. To this end, we consider a second way to interpret $\mathbf{ProphE}_V$. Prophecy variables are only supposed to be a feature of the *logic*, whereas during *execution*, they ought to do nothing at all: adding prophecy variables to your program should not affect its behavior, only the process of verifying it. This idea is implemented by the *erased interpretation* of $\mathbf{ProphE}_V$:

$$\mathsf{interp}_{\mathsf{erased}} : \forall A.\, \mathbf{ProphE}_V\, A \to \mathbf{itree}\ (\mathbf{HeapE}_{()})\, A$$
$$\mathsf{interp}_{\mathsf{erased}}(\mathsf{new\_proph}) := \mathsf{alloc}()$$
$$\mathsf{interp}_{\mathsf{erased}}(\mathsf{resolve}(p, v)) := \mathsf{load}(p); \mathsf{Ret}()$$

Under this interpretation, the prophecy operations are nearly no-ops. However, we still keep track of what prophecy variables p have been allocated, so that we can guarantee that new_proph always dispenses a new, unique p, and that $\mathsf{resolve}(p, v)$ causes Undefined Behavior when p is dangling. Nevertheless, prophecy variables store no guess about the future: instead they just store unit as a placeholder. In effect, this interpretation "erases" prophecy operations.

Accordingly, $\mathsf{interp}_{\mathsf{erased}}$ gives rise to an execution handler $(\rightsquigarrow_{\mathsf{erased}})$ via lifting.

Soundness of prophecy variables. With $(\rightsquigarrow_{\mathsf{erased}})$, we have defined a sensible notion of execution for $\mathbf{ProphE}_V$. However, this notion of execution has no connection to the rules in Figure 2! Those rules were derived via $\mathbf{ProphH}_V$, which is sound w.r.t. $(\rightsquigarrow_{\mathsf{prophetic}})$ (Lemma 5.2), but *a priori* there is no reason to think that this says anything about $(\rightsquigarrow_{\mathsf{erased}})$.

To bridge this gap, ideally we would prove something like $\mathsf{sound}(\mathbf{ProphH}_V, I, \rightsquigarrow_{\mathsf{erased}})$ and then apply Theorem 4.1. However, that assumption does not hold. So instead, we resort to proving a theorem that has the shape of Theorem 4.1 but is specialized to this particular case:

Theorem 5.1 (Soundness of prophecy variables). *Let E be an event type with a logical effect handler H and an execution handler $(\rightsquigarrow_E)$. Suppose that $(\rightsquigarrow_E)$ is demonic (*i.e., *does not make use of*

angelic choice). Then:

$$\frac{\mathsf{sound}(H, I, \rightsquigarrow_E) \qquad (t, (\sigma, \emptyset)) \Downarrow_{erased} T \qquad I(\sigma) * wpi_{H \oplus \mathbf{ProphH}_V}\ t\ \{\Phi\}}{\exists t', \sigma', \sigma_p.\ (t', (\sigma', \sigma_p)) \in T * I(\sigma') * wpi_{H \oplus \mathbf{ProphH}_V}\ t'\ \{\Phi\}}$$

Here, $(\Downarrow_{\text{erased}})$ resp. $(\Downarrow_{\text{prophetic}})$ are the big-step relations obtained from $(\rightsquigarrow_{\text{erased}})$ resp. $(\rightsquigarrow_{\text{prophetic}})$ and $(\rightsquigarrow_E)$. The key difference to Theorem 4.1 is that we have to rule out the presence of angelic choice in the execution handler $(\rightsquigarrow_E)$ for the remaining events. Prophecy variables are fundamentally incompatible with angelic choice, so this restriction is required for soundness.

To establish this theorem, we first notice that $\mathbf{ProphH}_V$ is sound w.r.t. $(\rightsquigarrow_{\text{prophetic}})$ because both were defined by lifting along $\mathsf{interp}_{\text{prophetic}}$. This is a consequence of the following general lemma about lifting:

Lemma 5.2 (Lifting soundness). *Let $(\rightsquigarrow_E)$ be the lift of $(\rightsquigarrow_{E'})$ along $f : \forall A.\, E\, A \to \textbf{\textit{itree}}\, E'\, A$, let H' be a logical effect handler for E', and let $H := f^* H'$. If* $\mathsf{sound}(H', I, \rightsquigarrow_{E'})$, *then* $\mathsf{sound}(H, I, \rightsquigarrow_E)$.

Therefore, in order to prove Theorem 5.1 (that $\mathbf{ProphH}_V$ is sound w.r.t. erased executions $(\Downarrow_{\text{erased}})$), we need to connect the erased executions $(\Downarrow_{\text{erased}})$ to the prophetic executions $(\Downarrow_{\text{prophetic}})$. This is our version of prophecy erasure.

Erasure theorem. To explain the statement of our erasure theorem, we consider an example:

$$t := p \leftarrow \mathsf{new\_proph}; \mathsf{resolve}(p, 1); \mathsf{resolve}(p, 2); \mathsf{resolve}(p, 3); \mathsf{Ret}()$$

Under an erased execution $(\Downarrow_{\text{erased}})$, prophecy operations are simply skipped; while under a prophetic execution $(\Downarrow_{\text{prophetic}})$, all future prophecy resolutions must be guessed already at the time of the new_proph. The erasure theorem says that every erased execution has a corresponding prophetic execution. For instance, there is an erased execution $t \Downarrow_{\text{erased}} \{\mathsf{Ret}()\}$ (ignoring state) that simply allocates location $p = 0$ and skips over all the prophecy operations. By inspecting that erased execution, we can read off those resolves that we skipped over: $(0, 1), (0, 2), (0, 3)$. Using these resolution values, we can construct a prophetic execution $t \Downarrow_{\text{prophetic}} \{\mathsf{Ret}()\}$ that already at the time of encountering new_proph allocates $0 \mapsto [1, 2, 3]$ on the prophecy heap.

That is the idea behind the erasure theorem. In effect, it guarantees us that there are "enough" prophetic executions. Let us put this in more precise language. Suppose $t : \textbf{itree}\ (E \oplus \mathbf{ProphE}_V)\ R$. Fix an execution handler $(\rightsquigarrow_E)$ for E. Then we have:

Theorem 5.3 (Erasure). *Assume $(\rightsquigarrow_E)$ is demonic (*i.e., *does not make use of angelic choice). Then:*

$$(t, (\sigma, \emptyset)) \Downarrow_{erased} T \implies (t, (\sigma, \emptyset)) \Downarrow_{prophetic} T$$

This theorem is a semantic analogue of the erasure theorem of [18, §3.5]. Our theorem is slightly weaker than theirs in the sense that their theorem erases prophecy variables entirely and replaces them by "poison" values, whereas our theorem leaves the prophecy variables p in place (as nothing but globally unique identifiers).

Unlike their theorem, Theorem 5.3 is not proven with reference to any specific language but is instead *language-agnostic* and applies to any ITree that does not invoke angelic choice. For reasons discussed above, it follows that prophecy variables for ITrees are sound (Theorem 5.1). In turn, our prophecy fragment PROPH is sound and can therefore safely be imported into any Mosaic language. By stating and proving soundness once and for all in a language-generic way, we make it trivial to introduce prophecy variables into any language.

6 Mosaic in Rocq

Mosaic and all the case studies in the following section have been fully implemented in Rocq based on the Iris library and the existing mechanization of wpi [17, 21, 34]. Our Rocq development of Mosaic provides a library of 27 fragments, shown in Figure 3. Note that the FUN fragment from §2 is actually composed of two fragments: VAR contains just variables x, and LAMBDA contains lambda abstractions and applications. Also, integer comparison is split out from BOOL into a separate fragment INTCMP_BOOL. Most of these fragments use event types defined by Vistrup et al. [34]; only PROPH comes with a new event type **ProphE**.

The main departure of Mosaic in Rocq from Mosaic on paper is how we actually take the required fixed points. As mentioned in §2, the type-level functions that define the value and expression constructors of each fragment need to satisfy a technical condition for the fixed point to be well-defined: they need to be *strictly positive*. Rocq has no internalized notion of strict positivity, so it is not possible to write a Rocq construction that takes an arbitrary set of fragments and returns a language. Instead, every Mosaic instance provides its own `Inductive` type. For instance, a language that wants units, variables, and lambdas would define the syntax by writing:

```
Definition val_pre (val expr : Type) : Type := val_unit + val_lambda expr.
Definition expr_pre (val expr : Type) : Type := expr_val val + expr_var + expr_lambda expr.
Inductive expr := | FixE (e : expr_pre val expr)
     with  val := | FixV (v : val_pre val expr).
```

Rocq will automatically inspect the `expr_` and `val_` types defined by the fragments to ensure that this entire definition is strictly positive.

To equip this language with the Mosaic machinery, the user has to instantiate a few typeclasses. Those typeclasses automatically determine what fragments the language consists of, allowing the language to inherit all the notation, lemmata, and tactics from the participating fragments. In the future, we intend to define some macros to simplify this routine boilerplate.

Use of LLMs. Large language models were used to fill out some of the proofs in our Rocq mechanization, primarily the case-heavy compatibility proofs in the case studies (§7). All relevant theorem statements were written by hand, so we incur no risk of the LLM introducing errors.

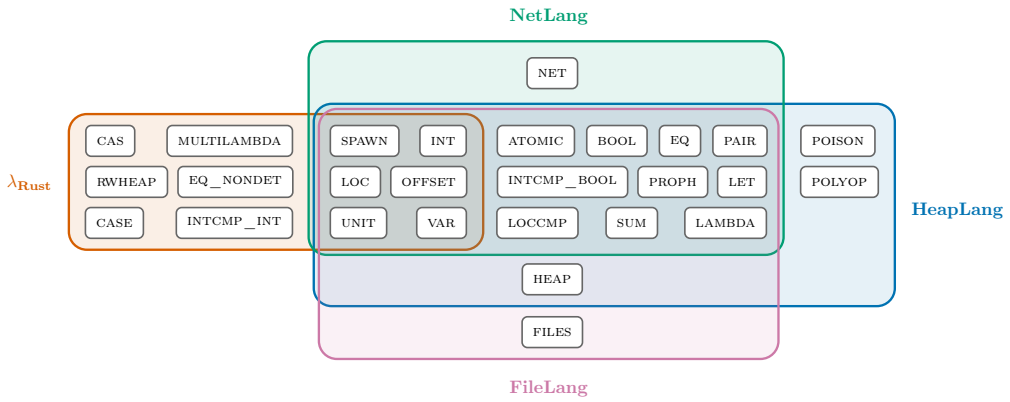

Fig. 3. Languages and their fragments

7 Case Studies

In this section, we demonstrate that Mosaic can be used to build practical program logics. First, we show full-fledged Mosaic-based models of languages from prior work by recreating HeapLang (§ 7.1) and λ_{Rust} (§ 7.2) as a composition of fragments. Then, we show that Mosaic can be used to easily build variants of existing languages by creating two languages based on HeapLang: a language with a file system and a language with a network (§ 7.3). Finally, we evaluate the usability of the created program logics in Rocq (§ 7.4).

7.1 HeapLang

HeapLang [17, §6.1] is an ML-like language that is shipped with the Iris separation logic. It is an untyped lambda calculus with concurrency and a higher-order heap. Many projects have created languages based on HeapLang, either by forking and extending it [5, 29, 22] or by translating code into HeapLang [12, 27, 24]. Its widespread use makes it a fitting case study for testing Mosaic.

We obtain HeapLang by composing 19 different fragments as shown in Figure 3. From this composition, we obtain all proof rules of HeapLang's program logic, subject to the minor changes inherited from the ITree approach [34, §6.4], and some adjustments to the handling of atomic `resolve` as discussed below. These changes do not affect the expressive power of our logic as demonstrated by the evaluation in § 7.4.

The main difference is that our version uses atomic blocks (§ 3.3) to encode atomic operations, while the original version provides a fixed set of atomic operations. In Mosaic, atomic blocks can be added by simply including the ATOMIC fragment, whereas in the original HeapLang this would require fundamental changes to the operational semantics, program logic, and soundness proof.

We leverage the compatibility theorem from § 4.3 to prove that our HeapLang program logic provides the same soundness guarantees as the original HeapLang. While doing so, we discovered that the original semantics of $\texttt{resolve}(e_1, e_2, e_3)$ behaves in unexpected ways in some cases. For example, $\texttt{resolve}((\lambda x.\, x)(1+1+1), p, v)$ cannot step even though $\texttt{resolve}(1+1+1, p, v)$ can. The original HeapLang semantics does not allow the first argument of $\texttt{resolve}(e_1, e_2, e_3)$ to base-step to a non-value, except if the step happens strictly within a non-trivial evaluation context. Our semantics does not exhibit this unexpected behavior and we submitted a merge request against the original HeapLang to fix this issue. We count this discrepancy as evidence towards the simplicity and naturality of our ITree-based semantics for prophecy variables over operational semantics. The issue is caused by an operational semantics rule that looks sensible on the surface, but is problematic due to the subtle definition of stuckness in operational semantics. In contrast, under ITree-based semantics, unsafe failure is always explicit in the denotation. With these patched semantics of HeapLang, we use the compatibility theorem (Theorem 4.4) to show that our version of HeapLang is sound w.r.t. the original semantics, while also providing the same program logic rules, all obtained from fragment composition.

7.2 λ_{Rust}

λ_{Rust} is a language introduced by Jung et al. [16] to model the Rust programming language. We recreate λ_{Rust} in Mosaic by combining 12 fragments shown in Figure 3. Half of these fragments are shared with HeapLang and the other half are new. In the following, we briefly discuss the main differences between λ_{Rust} and HeapLang.

First, λ_{Rust} uses a block-based memory model with non-atomic memory accesses that cause Undefined Behavior on data races. To model this, we replace HeapLang's HEAP fragment with a

new RWHEAP fragment.[10] The original λ_{Rust} encodes non-atomic accesses by adding administrative redexes to the language that represent a half-completed memory operation. In our approach, such administrative redexes are not necessary since we can simply include a yield in the middle of the semantics of non-atomic accesses.

Second, λ_{Rust} uses n-ary functions instead of the unary functions of HeapLang. Since functions are also just modeled as a fragment in Mosaic, changing this amounts to swapping out the LAMBDA fragment for a MULTILAMBDA fragment. This demonstrates that our treatment of substitution does not lock in a particular choice of lambda expression and associated semantics.

Third, λ_{Rust} uses a different notion of values than HeapLang. For example, λ_{Rust} does not have native booleans but encodes them using integers. For this reason, λ_{Rust} does not contain the BOOL fragment, and instead uses the CASE fragment for if-statements.

Finally, λ_{Rust} uses left-to-right evaluation order while HeapLang uses right-to-left evaluation order. To avoid having to bifurcate the theory, we make evaluation order a global flag: every language chooses between left-to-right and right-to-left evaluation orders, and all fragments mentioned so far are defined parametric in this choice. Thus, λ_{Rust} can still import any of the fragments mentioned earlier, but with the appropriate evaluation order.

To demonstrate that our version of λ_{Rust} provides the same program logic as the original, we port several examples to Mosaic (see §7.4). We also use the compatibility theorem (Theorem 4.4) to prove that our version of λ_{Rust} is sound w.r.t. the original operational semantics. Large parts of this proof are shared with HeapLang thanks to fragment-local ($\rightarrowtail$) and $\mathsf{unsafe}(e, \sigma)$ lemmata.

7.3 File System and Network

This section describes two case studies that demonstrate how Mosaic can be used to adjust existing languages. Concretely, we instantiate two variants of HeapLang: (1) FileLang, which extends it with a file system, and (2) NetLang, which replaces the global heap with a network. Neither of these features is natively provided by HeapLang, leading prior work that required them to fork HeapLang [5, 29]. Here, we turn those extensions into the independent fragments FILES and NET that can be freely added to existing languages.

File system. The FILES fragment provides operations FileCreate, FileRead, FileWrite, FileAppend, and FileSize over a file store mapping file identifiers h to contents $d \in \mathsf{list}\ \mathbb{Z}$. On the logic side, the fragment provides a points-to assertion $h \overset{\text{file}}{\mapsto} d$ representing ownership of a particular file. FileRead(h) reads the contents of h into a freshly heap-allocated buffer and returns a reference to said buffer, while FileWrite(h, ℓ, n) writes n bytes stored at location ℓ into the file h. To express these interactions with the heap, FILES depends on the HEAP fragment.

A subtle aspect of these file operations is the atomicity guarantees that they provide. FileWrite performs two kinds of effects: it reads from the heap, and it writes to the file. The intended semantics is that the file is written atomically, but the reads from the heap can be arbitrarily interleaved with operations by other threads. While such partially atomic operations can be difficult to express in operational semantics, our ITree-based denotational semantics makes this straightforward: we insert yields between all loads from memory but do not insert a yield during the write to the file.

Network. Our NET fragment is inspired by Grove [29] and provides the operations Listen, Accept, Connect, Send, and Recv over an unreliable network: messages may be dropped, duplicated, and reordered, and connections may fail. The network state is a map from channels to the set of

[10]The name RWHEAP arises because λ_{Rust} equips every memory cell with a reader-writer lock to ensure that programs get stuck if there is a data race.

lang	file	lemmata	sentences	edit	edit%
HeapLang	`eager_coin`	2	15	0	0.0
	`lazy_coin_one_shot`	2	39	6	15.4
	`treiber`	3	92	8	8.7
	`atomic_snapshot`	4	237	15	6.3
	`herlihy_wing_queue`	3	1687	35	2.1
λ_{Rust}	`new_delete`	2	14	6	42.9
	`memcpy`	1	19	6	31.6
	`swap`	1	19	4	21.1
	`lock`	5	49	8	16.3
	`spawn`	3	86	10	11.6
	`arc`	11	642	47	7.3
Total	11 files	37	2899	145	5.0

Fig. 4. Proof adaptation effort when porting existing verifications. For each file: the number of ported lemmata, the number of tactic sentences in the original proof bodies, the sentences affected overall by the port (edit), and the fraction of affected sentences (edit%).

messages that have been sent on this channel in the past—any of those messages can be delivered arbitrarily often going forward. To model these channels, we can reuse the **HeapE** infrastructure.

7.4 Tactics

Working with a program logic in a theorem prover requires more than just mechanizing the proof rules. Languages like HeapLang and λ_{Rust} come with an extensive collection of customized tactics for symbolic execution such as `wp_pures`, `wp_load`, and `wp_alloc`. These tactics automatically find the right subexpression to reduce, apply the corresponding proof rule, and tidy up the resulting goal. Traditionally, this tactic infrastructure is duplicated for each language. Mosaic enables fragments to define their own tactics, automatically made available for every language that includes the fragment. For example, HEAP contributes reusable analogues of `wp_load` and `wp_store`.

To evaluate this language-agnostic infrastructure, it is necessary to investigate whether the resulting proof experience actually matches that of the original, hand-crafted languages. To answer this question, we ported a body of pre-existing proofs to our instantiations of HeapLang and λ_{Rust}. For HeapLang, we ported a logically atomic specification for a Treiber stack implementation from the Iris examples repository, as well as the atomic snapshot, Herlihy–Wing queue, and prophecy-variable coin examples of Jung et al. [18]. For λ_{Rust}, we ported the verified libraries that ship with the λ_{Rust} development [16] (`arc`, `lock`, `spawn`, and several smaller ones).

The results of this effort are shown in Figure 4. For each example, we report the number of lemmata involving program logic reasoning (column: lemmata), the number of Rocq tactic sentences in the proofs of these lemmata (column: sentences), the number of edited sentences (column: edit), and the edit percentage (column: edit%). Note that helper lemmata and lemmata about custom ghost theory are not counted since they do not involve the `wp_*` tactics.

Across 37 proofs comprising 2899 tactic statements, only 5% of statements needed adjustments; the remaining 95% were copied verbatim. The main causes of these edits were that (1) we define prophecy variables using infinite iterators while the original version uses finite lists, (2) our version of HeapLang uses atomic blocks instead of a fixed set of primitive atomic operations, and (3) small differences in notation since Mosaic does not have a concept of literals (as separate from values).

In total, this demonstrates that our versions of HeapLang and λ_{Rust} provide similar proof ergonomics as the original versions, even though our versions are built from reusable fragments.

8 Related Work

Building the syntax of a programming language from composable fragments and equipping it with a semantics has a long history [30, 10, 20, 11, 26, 4, 33]. However, none of these prior works consider the question of obtaining a program logic for the composed language. We compare our approach for modeling the syntax, semantics, and program logic with that of prior work.

Syntax. "Data types à la Carte" [30] begins this line of work in Haskell by showing how the syntax of a language, and similar data types, can be built up from independent pieces if each piece is a type-level function that supplies new constructors. They define the overall data type as a single big fixed point that includes all these pieces. Later work ported this approach to theorem provers such as Rocq or Agda, which required some adjustments: For reasons of soundness, theorem provers require data type definitions to be *strictly positive*. Therefore, to ensure that we can obtain the syntax as a fixed point of an arbitrary set of fragments, the type-level function has to be encoded in a way that structurally ensures its strict positivity. Possible options for this are Church encodings [10, 11] or "signatures"/"containers" [3, 20, 4, 33]. In Mosaic, each language is its own `Inductive` (§ 6), which side-steps the need for abstracting over an arbitrary strictly positive function, at the expense of more boilerplate for each language instantiation.

An alternative encoding for compositional data types is *Trees That Grow* [26]. This begins with a plain "undecorated" data type that is made extensible by adding a parameter that can add more constructors and decorate existing ones. However, this work focuses more on the type declaration than on defining reusable operations over these types.

Semantics. The standard approach to add *semantics* to syntax obtained in this way is to equip language fragments with interpreters. Initially, those only supported pure languages [10, 20]. Soon thereafter, the approach was extended to monadic interpreters that can model effects [11, 33]. To avoid hard-coding the set of available monads, monad transformers have been used [11], though some work did end up with a fixed set of effects since monad transformers are not sufficiently compositional [33]. Mosaic uses ITrees to achieve this, which avoids these compositionality issues (prior work [33] has already mentioned this as an interesting option).

One particularly subtle aspect of these interpreters is how to deal with *dependencies* among fragments. Most prior work does not discuss this issue. van der Rest et al. [33] deal with it by distinguishing "canons" that add new value forms, and "fragments" that add expressions and can depend on "canons". In Mosaic, fragments can arbitrarily and even mutually depend on each other.

On top of these semantics, prior work is mostly interested in meta-theoretical properties related to substitution [4] or type soundness and strong normalization [10, 20, 11, 33]. In contrast, Mosaic focuses on program logics and their soundness. In future work, it would be interesting to consider whether this could be extended to type soundness using logical relations [32].

The line of work on Modular Structural Operational Semantics [25] achieves modularity by using a compositional notion of state-transition system that avoids having to talk about the entire state in each transition rule. This work has later been mechanized in Rocq [23] and extended to bisimulation reasoning [7] and to expressing (but not proving sound) type systems [8].

Program logics. We use the program logic for ITrees from "Program Logics à la Carte" [34] as the model for our weakest precondition. Their work focuses on using ITrees to define the semantics of a *concrete* language, providing a reusable set of event types so each new language will not have to start from scratch. We make use of those same event types, and take their approach one step further by using ITrees in the semantics of a fragment that can be contained in an *arbitrary* language. We also extended their framework with support for atomic blocks and prophecy variables, as well as a compositional approach for proving compatibility with an operational semantics.

Iris provides a language interface [31] alongside a language-generic program logic for any operational semantics that fits the interface. However, only the language-independent proof rules come "for free"; all the other rules have to be proven against the operational semantics, with many of these proofs being effectively duplicated between similar languages.

GooseLang [5, 29] is a fork of HeapLang used as an underlying model for verifying Go programs. It contains support for *FFI modules* which can add a set of operations and associated semantics to the language. This is used to model system operations such as file system and network access, hence the name. However, there is no notion of composition for FFI modules: each module has to define the full set of operations needed in any particular case study. This ad-hoc implementation of extensible languages shows the practical need for being able to share large parts of a language definition and associated program logic among similar but distinct languages. Our NET and FILES fragments are closely inspired by two of GooseLang's FFI modules.

9 Conclusion

We have shown how to marry the lines of work on "Data types à la Carte" [30] and "Program logics à la Carte" [34] into *Mosaic*, a framework for composable language fragments and associated program logics. Among other things, the framework allows prophecy variables to be modularly integrated into any language, greatly simplifying an otherwise tedious language extension. Our provably faithful models of HeapLang and λ_{Rust} show that Mosaic is versatile enough to support rich languages established in prior work, including their tactic infrastructure.

In future work, it would be interesting to test the boundaries of the kinds of effects the system can handle. Thread-local state is a common feature that we currently do not support; this would also form the foundation for weak memory models in the style of iGPS [19, 9]. More ambitiously, it would be interesting to explore support for control effects such as exceptions or call/cc.

References

[1] Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 165–175. doi:10.1109/LICS.1988.5115

[2] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. doi:10.1016/0304-3975(91)90224-P

[3] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. doi:10.1016/J.TCS.2005.06.002

[4] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.* 31 (2021), e22. doi:10.1017/S0956796820000076

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

[6] 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. doi:10.1145/3579834

[7] Martin Churchill and Peter D. Mosses. 2013. Modular Bisimulation Theory for Computations and Values. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 7794)*. Springer, 97–112. doi:10.1007/978-3-642-37075-5_7

[8] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. *LNCS Trans. Aspect Oriented Softw. Dev.* 12 (2015), 132–179. doi:10.1007/978-3-662-46734-3_4

[9] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *PACMPL* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[10] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *POPL*. ACM, 207–218. doi:10.1145/2429069.2429094

[11] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. 2013. Modular monadic meta-theory. In *ICFP*. ACM, 319–330. doi:10.1145/2500365.2500587

[12] Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated Reasoning About Non-determinism in C Expressions. In *ESOP (LNCS, Vol. 11423)*. 60–87. https://doi.org/10.1007/978-3-030-17184-1_3

[13] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. 442–451. doi:10.1145/3209108.3209174

[14] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. https://doi.org/10.1145/3656422

[15] Ralf Jung. 2020. *Understanding and evolving the Rust programming language*. Ph. D. Dissertation. Universität des Saarlandes, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

[16] 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. doi:10.1145/3158154

[17] 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

[18] 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. doi:10.1145/3371113

[19] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[20] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *WGP@ICFP*. ACM, 13–24. doi:10.1145/2502488.2502491

[21] 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

[22] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP (LNCS, Vol. 12075)*. Springer. doi:10.1007/978-3-030-44914-8_13

[23] Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. 2011. Formal Component-Based Semantics. In *SOS (EPTCS, Vol. 62)*. 17–29. doi:10.4204/EPTCS.62.2

[24] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP (LNCS, Vol. 11423)*. Springer. doi:10.1007/978-3-030-17184-1_1

[25] Peter D. Mosses. 2004. Modular structural operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 195–228. doi:10.1016/J.JLAP.2004.03.008

[26] Shayan Najd and Simon Peyton Jones. 2017. Trees that Grow. *J. Univers. Comput. Sci.* 23, 1 (2017), 42–62. http://www.jucs.org/jucs_23_1/trees_that_grow

[27] João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. 2025. Modular Reasoning about Global Variables and Their Initialization. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 355 (Oct. 2025), 28 pages. doi:10.1145/3763133

[28] 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

[29] 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. doi:10.1145/3600006.3613172

[30] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

[31] The Iris Team. 2026. The Iris 4.5 Reference. https://plv.mpi-sws.org/iris/appendix-4.5.pdf

[32] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[33] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter D. Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1903–1932. doi:10.1145/3563355

[34] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proc. ACM Program. Lang.* 9, POPL (2025), 300–331. doi:10.1145/3704847

[35] 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. doi:10.1145/3371119