# EXECUTABLE DENOTATIONAL SEMANTICS WITH INTERACTION TREES

Li-yao Xia

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

## ABSTRACT

EXECUTABLE DENOTATIONAL SEMANTICS WITH INTERACTION TREES

Li-yao Xia
Benjamin C. Pierce

Interaction trees are a representation of effectful and reactive systems designed to be implemented in a proof assistant such as Coq. They are equipped with a rich algebra of combinators to construct recursive and effectful computations and to reason about them equationally. Interaction trees are also an executable structure, notably via extraction, which enables testing and directly developing executable programs in Coq. To demonstrate the usefulness of interaction trees, two applications are presented. First, I develop a novel approach to verify a compiler from a simple imperative language to assembly, by proving a semantic preservation theorem which is termination-sensitive, using an equational proof. Second, I present a framework of concurrent objects, inheriting the modularity, compositionality, and executability of interaction trees. Leveraging that framework, I formally prove the correctness of a transactionally predicated map, using a novel approach to reason about objects combining the notions of linearizability and strict serializability, two well-known correctness conditions for concurrent objects.

# Contents

# List of Figures

CHAPTER 1

# Introduction

Machine-checked proofs are now feasible at scale, for real systems, in a wide variety of domains, including programming language semantics and compilers [Leroy, 2009, Kumar et al., 2014], operating systems [Klein et al., 2009, Gu et al., 2016], interactive servers [Koh et al., 2019], databases [Malecha et al., 2010], and distributed systems [Wilcox et al., 2015, Hawblitzel et al., 2015a], among many others. Common to all of these is the need to model and reason about interactive, effectful, and potentially nonterminating computations.

For this, most work to date has relied on *operational semantics*, represented as (small- or big-step) transition relations defined on syntax [Igarashi et al., 2001, Leroy and Blazy, 2008, Leroy, 2009, Milner et al., 1997, Jung, 2020, Wang et al., 2018, Guth, 2013]. These representations have their advantages: they are expressive, since nearly any semantic feature can be modeled by transition systems or traces when combined with appropriate logical predicates; and they fit smoothly with inductive reasoning principles that are well supported by interactive theorem provers. Although the underlying ideas are widely applicable, in practice the common approach is still to reimplement the operational semantics of a new system from scratch.

By contrast, denotational semantics views programs as symbolic representations of abstract objects [Scott and Strachey, 1971]. An early example is the study of continuous functions between lattices as a denotational semantics for the lambda calculus [Scott, 1976], and this formalism was later adopted in the specification of the Haskell programming language [Jones, 2003, Hudak et al., 2007]. Two key characteristics of denotational semantics are *compositionality* and *abstraction*.

Compositionality is the principle that "the whole is the sum of its parts". A compositional semantics defines the meaning of a program by composing the meanings of its components. Hence, the constructs of a programming language (conditionals, loops, applications, etc.) define functions which map denotations of their subphrases to the denotation of the whole, by leveraging operators that make up the algebraic structure of the underlying semantic domain—lattices, categories, monads, etc.

Abstraction is the act of suppressing irrelevant complexity; a semantics should expose only the behavior of a program that is relevant to its clients. This contrasts with the syntactic nature of operational semantics, which includes the code of a program in the state of its abstract machine. For instance, the purely functional programming paradigm views programs as denoting mathematical functions, exposing only relations between input and output while hiding the mechanism by which they are computed. Purity enables one to prove program properties rigorously via equational reasoning. It also motivated monads as an effective solution to reintroduce effects in this paradigm [Wadler, 1992, Jones, 2005]. Purely functional programming is exemplified by Haskell, which paved the way for a family of languages: PureScript, Idris, Agda, Curry, etc., and inspired new features and design patterns in other programming languages.

By deemphasizing its internal state, we are led to focus more on the interactions of the program with its environment. One reasonable approach is to represent the behavior of the program as a set of traces, each recording a possible sequence of interactions between program and environment. As a result, observational equivalence between programs arises as the canonical equivalence between their denotations. To obtain compositional semantics, it's often necessary to require additional structure over plain sets of traces, suggesting alternative representations such as trees and games [Hyland, 1997, Abramsky and McCusker, 1999, Abramsky et al., 1997]. In this dissertation, I develop a library for representing the behavior of programs as a tree datatype, which provides a rich theory of composition, which can be extracted and interpreted for testing, and which can be formalized in a proof assistant such as Coq, Agda, or Lean.

## 1.1. Executable Denotational Semantics

This dissertation presents *interaction trees*, a framework for representing the behavior of recursive and effectful programs. This framework builds upon strong theoretical foundations: the core data structure is a free monad, having deep roots in category theory, and the implementations of the associated operations are largely determined by their types. Furthermore, interaction trees are an executable representation: a denotational semantics using interaction trees yields a working reference interpreter, with testing as one possible use case. Hence, I propose to construct executable denotational semantics formally using interaction trees.

Interaction trees are implemented as a Coq library, named ITree, allowing users to express denotational semantics for effectful and possibly nonterminating computations in Gallina, the specification language of Coq [2018], despite Gallina's strong purity and termination constraints. Interaction trees work well with Coq's extraction capabilities, making them compatible with tools such as QuickChick [Lampropoulos and Pierce, 2018] for testing, and allowing us to easily link the extracted code against non-Coq components such as external libraries, so that we can directly execute systems modeled using interaction trees. This combination of features makes interaction trees a practical foundation for formal verification of interactive systems.

After an overview of the ITree library in Chapter 2, I demonstrate the usefulness and flexibility of interaction trees by presenting two applications, originally from published work in Xia et al. [2019] and Lesani et al. [2022]. Chapter 3 describes a verified compiler from a toy imperative language Imp to a control-flow graph language Asm. Both the source and target language are given denotational semantics in interaction trees. A semantic preservation theorem is then proved by equational reasoning. A key feature of this proof is that it hides coinductive reasoning behind equations for loop combinators. Thus, such a proof only requires familiarity with equational reasoning, turning a technical simulation argument into intuitive rewritings of string diagrams. Furthermore, this equational formulation of semantic preservation is termination-sensitive.

Chapter 4 presents a second application, using interaction trees to implement concurrent and transactional objects. I formalize the standard notions of linearizability and strict serializability, and prove those properties on a number of concrete objects. To define serializability, transactions are represented as interaction trees that transactional objects may instrument to interpret them atomically. Our main case study is a proof of correctness of *transactional predication*, a general technique for constructing a performant transactional map object by combining a concurrent map and a transactional memory object. The core of the proof is again phrased in equational terms, using lemmas from the ITree library to connect the linearizability and serializability assumptions about the individual components, thus establishing the serializability of the composed object.

Chapter 5 discusses related work, including other applications of interaction trees, and a broad literature on modelling effects in type theory that underpins my contributions. Chapter 6 concludes with possible directions for future work.

## 1.2. History and Credits

The earliest version of what is now called interaction trees was due to Joachim Breitner and Dmitri Garbuzov, including the `itree` type definition and an early definition of weak bisimulation. I then started working on it with the objective of specifying a web server [Koh et al., 2019]. I focused on the core library while my coauthors developed the integration with VST, and a harness for randomized testing. Following that, as part of Xia et al. [2019] which properly presents the ITree library (Chapter 2), I worked jointly with Yannick Zakowski and Gregory Malecha on the compiler case study which constitutes Chapter 3 in this dissertation. The trickiness of simulation-based proofs motivated me to add the loop combinators and their associated equations, inspired by arrows [Hughes, 2000, Paterson, 2001]. Chung-Kil Hur suggested redefining weak bisimulation using Paco, and contributed further improvements such as unifying the definitions of strong and weak bisimulation. Paul He formalized the relation between interaction trees and trace semantics in Section 7 of Xia et al. [2019].

The origin of verified transactional objects [Lesani et al., 2022] (Chapter 4) predates ITree, and the project independently built upon the same coinductive type. The proof principle for linearizability in Section 4.5 and its applications—including the verification of the transactional mutex lock in Section 4.6.1—were developed prior to my involvement. I actively participated in writing the expository sections (Sections 4.2 to 4.4 and 4.6), and I subsequently refactored the implementation to align more closely with the presentation of concepts in that paper. My main technical contribution is the transactional predication correctness proof (Section 4.7).

CHAPTER 2

# Interaction Trees

Interaction trees are a datatype for representing computations that can interact with an external environment. We think of such computations as producing a sequence of *visible events*—interactions—each of which might carry a response from the environment back to the computation. The computation may also eventually halt, yielding a final value, or diverge by continuing to compute internally but never producing a visible event.

## 2.1. Definition

Figure 2.1 shows the definition of the type `itree E R`. The parameter `E : Type → Type` is a type of *external interactions*: it defines the interface by which a computation interacts with its environment, as we explain below. The type `R` is the *result type* of the computation: if the computation ever halts, it will return a value of type `R`.

```
CoInductive itree (E: Type → Type) (R: Type): Type :=
| Ret (r: R)                                       (* Terminate with value r *)
| Tau (t: itree E R)                               (* Silent transition with child t
    *)
| Vis {A: Type} (e : E A) (k : A → itree E R). (* Visible event e, answer in A *)
```

FIGURE 2.1. Simplified presentation of interaction trees

ITrees are defined *coinductively* so that they can represent potentially infinite sequences of interactions or divergent behaviors. They are built using three constructors. `Ret r` corresponds to the trivial computation that immediately halts and produces `r` as its result. `Tau t` corresponds to a silent step of computation that does something internal, producing no visible events, and then continues as `t`. Representing silent steps explicitly allows ITrees to represent diverging computations without violating Coq's *guardedness condition* [Giménez, 1995, Chlipala, 2017]. It is a syntactic side-condition on co-recursive definitions which ensures that a finite amount of computation suffices to expose the next constructor of the coinductive type. Concretely, the results of co-recursive calls must occur under constructors and cannot be eliminated by pattern matching.

The final and most interesting way to build an ITree is with the `Vis A e k` constructor (`A` is often left implicit). Here, `e : E A` is a *visible* external event, which consists of data that the computation emits to an external environment, and `A` (for *answer*) is the type of data that the environment provides in response to the event. The constructor also specifies a continuation, `k : A → itree E T`, which produces the rest of the computation given the response from the environment. The tree-like nature of interaction trees stems from the `Vis` constructor, since the continuation `k` can behave differently for different values of type `A`. Importantly, the continuation is represented as a meta-level (*i.e.*, Gallina) function, which means both that we can embed computation in an ITree and that the resulting datatype is extractable and contains executable functions.

The definition shown in Figure 2.1 follows Coq's historical style of using *positive coinductive types*, which emphasizes the tree-like structure via its constructors. This approach is known to break subject reduction [Giménez, 1996]. Our library therefore uses the recommended *negative coinductive* form [Coq development team, 2019, Hagino, 1989] where, rather than defining a coinductive type by providing its constructors, we instead provide its destructors, defining a record. We use "smart constructors" for `Ret`, `Tau`, and `Vis`, which have the types shown in this figure, so the distinction is mostly cosmetic (though it does impact the structure of proofs). We suppress these and similar details throughout this dissertation.

**2.1.1. Alternative Definitions.** Our definition of interaction trees relies on higher-order types and coinductive types, two features readily available in Coq. Nevertheless, it may still be possible to adapt much of the ITree library to another language that does not provide those features.

The type `itree` is higher-order, as it is parameterized by a function on types `E : Type → Type`. Higher-order types are a natural byproduct of dependent types, so they appear completely benign in Coq, Agda, and Lean. Foster et al. [2021] port interaction trees to Isabelle/HOL, which is based on higher-order logic rather than dependent types. Its term language only supports first-order types, so a significant modification of the definition of `itree` is necessary. There, the type parameter `E` ranges over plain types, rather than type functions. This "flattens" the structure of events, as events may no longer be statically associated with specific response types, and Foster et al. [2021] palliate this by enforcing that association dynamically, making the continuation in the `Vis` constructor a partial function which fails when events are answered with the wrong type of response.

While some proof assistants support coinductive types natively [Giménez, 1996, Blanchette et al., 2014], coinductive types are a feature that are not strictly essential to the expressiveness of a theorem prover, as they can be encoded using more primitive concepts [Avigad et al., 2019]. A notable approach is to view a coinductive type as the final coalgebra $\nu F$ of some functor $F$, which can be encoded via an existential type, $\nu F = \exists A, (A \to FA)$.

**2.1.2. Examples.** As a concrete example of external interactions, suppose we choose `E` to be the following type `IO`, which represents simple input/output interactions, each carrying a natural number. Then we can define an ITree computation `echo` that loops forever, echoing each input received to the output:

```
Inductive IO : Type → Type :=    CoFixpoint echo : itree IO void :=
| Input  : IO nat                   Vis Input (fun x ⇒
| Output : nat → IO unit.           Vis (Output x) (fun _ ⇒ echo)).
```

Note that `IO` is indexed by the expected answer type that will be provided by the environment in each interaction. Conversely, its constructors are parameterized by the arguments to be sent to the environment. Hence, an `Input` event takes no parameter and expects a `nat` in return, while an `Output` event takes a `nat` but expects a non-informative answer, represented by the `unit` type. The return type of `echo` is `void`, the empty type, since the computation never terminates.

Similarly, it is easy to define an ITree that silently diverges, producing no visible outputs and never returning a value:

```
CoFixpoint spin : itree IO void := Tau spin.
```

Or one that probes the environment until it receives 9 for an answer, at which point it terminates (returning `tt`, the unique value of type `unit`):

```
CoFixpoint kill9 : itree IO unit :=
  Vis Input (fun x ⇒ if x =? 9 then Ret tt else kill9).
```

The three basic ITree constructors and explicit `CoFixpoint` definitions provide very expressive low-level abstractions, but working with them directly raises several issues. First, Coq's syntactic guardedness check is inherently non-compositional, so it is awkward to construct large, complex systems using it. Second, we need ways of composing multiple kinds of events. Third, we often want to model the behavior of a system by interpreting its events as having *effects* on the environment. For example, a `Write` event could update a memory cell that a `Read` event can later access. Finally, to reason about ITrees and computations built from them as above, we would have to use coinduction explicitly. It is easier to work with loop and recursion combinators that are more structured and satisfy convenient equational reasoning principles that can be expressed and proven once and for all. The ITrees library provides higher-level abstractions that address all three of these concerns. Figure 2.2 provides a synopsis of the library; the details are explained below.

*Notation.* The library makes extensive use of *parametric functions*, which have types of the form `∀ (X:Type), E X → F X`. We write `E ⇝ F` as an abbreviation for such types.

**2.1.3. Composing ITree Computations: ITrees are Monads.** The type `itree E` is a monad [Moggi, 1989, Wadler, 1992], a conventional structure to sequence effectful computations in pure functional programming. Figure 2.3 gives the implementation of the monadic `bind` and `ret` operations. As shown there, `bind t k` replaces each leaf `Ret r` in `t` with the new subtree `k r`. The function `ret` is simply the constructor `Ret`, and we introduce the usual sequencing notation `x ← e ;; k` for `bind`.

We think of the visible events of an ITree as *uninterpreted effects*. In this sense, `itree E` is a *free monad* (in a suitable category: see Chapter 5) where every event of type `E A` corresponds to an effectful (monadic) operation that can be "triggered" to yield a value of type `A`:

```
Definition trigger {E : Type → Type} {A : Type} (e : E A) : itree E A :=
  Vis e (fun x ⇒ Ret x).
```

Using `trigger`, we can rewrite the `echo` example with less syntactic clutter:

```
CoFixpoint echo2 : itree IO void :=
  x ← (trigger Input) ;; trigger (Output x) ;; Tau echo2.
```

*Interaction tree operations*

```
itree E A : Type
Ret  : A → itree E A
Tau  : itree E A → itree E A
Vis  : E R → (R → itree E A) → itree E A
bind : itree E A → (A → itree E B) → itree E B
trigger  : E A → itree E A
```

*Events and subevents*

| | |
|---|---|
| `E, F : Type → Type` | |
| `(e : E R)` | `R` *is the result type of event* `e` |
| `E +' F` | *disjoint union of events* |
| `Class E -< F` | `E` *is a subevent of* `F` |
| `trigger '{E -< F} : E ⤳ itree F` | *overloaded trigger* |

*Heterogeneous weak bisimulation*

```
eutt (r : A → B → Prop) :
itree E A → itree E B → Prop
```

*Strong and weak bisimulation*

```
_ ≅ _ : itree E A → itree E A →
  Prop
_ ≈ _ := eutt eq.
```

*Parametric functions*

```
E ⤳ F  :=  ∀ (X:Type), E X → F X
```

*Monadic interpretation*

```
`{Monad M} `{MonadIter M}
interp : (E ⤳ M) → (itree E ⤳ M)
```

*Standard event types*

| *name* | *events* | *handler type* |
|---|---|---|
| `emptyE` | *none* | `∀ M, emptyE ⤳ M` |
| `stateE S` | `Get Put` | `(stateE S) ⤳ stateT S` |
| `mapDfaultE K V d` | `Insert LookupDfault Remove` | `` `{Map K V map} (mapDfaultE K V d) ⤳ (stateT map) `` |

FIGURE 2.2. Main abstractions of the ITREE library

```
(* Apply the continuation k to the Ret nodes of the itree t *)
Definition bind {E R S} (t : itree E R) (k : R → itree E S) : itree E S :=
  (cofix bind_ u := match u with
                    | Ret r ⇒ k r
                    | Tau t ⇒ Tau (bind_ t)
                    | Vis e k ⇒ Vis e (fun x ⇒ bind_ (k x))
                    end) t.

Notation "x ← t1 ;; t2" := (bind t1 (fun x ⇒ t2)).
Definition ret x := Ret x.
```

FIGURE 2.3. Monadic `bind` and `ret` operators for ITrees

**2.1.4. ITree Equivalences.** Interaction trees admit several useful notions of equivalence even before we ascribe any semantics to the external events. These properties are deceptively simple to state, but the weaknesses of coinduction in Coq make some the proofs quite challenging.

*Strong and Weak Bisimulations.* The simplest and finest notion of equivalence is *strong bisimulation*, written `t1` ≅ `t2`, which relates ITrees `t1` and `t2` when they have exactly the same shape.

The monad laws and many structural congruences hold up to strong bisimulation, but once we introduce loops, recursion, or interpreters, which use `Tau` to hide internal steps of computation, we need to work with a coarser equivalence. We want to equate ITrees that agree on their terminal behaviors (they return the same values) and on their interactions with the environment through `Vis` events, but that might differ in the number of `Tau`'s. This "equivalence up to `Tau`" is a form of *weak bisimulation*: it lets us remove any finite number of `Tau`'s when considering whether two trees are the same, while infinite `Tau`'s must be matched on both sides (*i.e.*, this equivalence is termination-ensitive). We write `t` $\approx$ `u` when `t` and `u` are equivalent up to `Tau`. A key equation distinguishing weak bisimulation from strong bisimulation is `Tau t` $\approx$ `t`. It is crucial for working with general computations modeled as ITrees.

*Heterogeneous Bisimulations.* Both strong and weak bisimulation can be further relaxed to relate ITrees that have different return types, which is needed for building more general simulations, such as the one used in our compiler correctness proof (Chapter 3). If we have `t1 : itree E A` and `t2 : itree E B` and some relation `r : A → B → Prop`, we can define `eutt r` ("equivalence up to `Tau` modulo `r`"), which is the same as $\approx$ except that two leaves `Ret a` and `Ret b` are related iff `r a b` holds. Intuitively, two such ITrees produce the same external events and yield results related by `r`. Indeed $\approx$ is defined as `eutt eq`, where `r` is instantiated to the Leibniz equality relation `eq`. It is straightforward to generalize $\cong$ in the same way.

Figure 2.4 gives the formal definition of `eutt r` as a nested inductive–coinductive structure [Danielsson and Altenkirch, 2009]. The inner inductive `euttF` relation implicitly defines a least fixed point; this results in a relation transformer, parameterized by `sim`, and we take its greatest fixed point `eutt` via the operator `nu`. The constructors of `euttF` can be read as inference *rules and corules*. The `EqRet` rule relates two nodes `Ret a` and `Ret b` when `r a b` holds. The `EqVis` rule relates two `Vis` nodes when they are labeled with identical events `e` and their continuation subtrees `k1 v` and `k2 v` are related by `sim` for every value `v` the environment could return. The rule `EqTau` relates `Tau t1` and `Tau t2` whenever `t1` and `t2` are related by `sim`, while `EqTauL` and `EqTauR` allow to strip off asymmetrically one extra `Tau` on either side. The `EqTau` and `EqVis` appeal to the `sim` relation: they are *corules*, relying on coinduction via the `nu` operator. In contrast, the `EqTauL` and `EqTauR` constructors contain recursive occurrences of the inductively defined `euttF`: they are *rules*. Intuitively, a derivation may consist of infinitely many corules, with only finitely many rules between two corules. This nested inductive-coinductive structure provides a flexible way of defining relations, generalizing inductive relations—which consist only of finite derivations with rules—and coinductive relations—which consist of potentially infinite derivations with corules. In this instance, we use the rules `EqTauL` and `EqTauR` to peel off any finite number of `Tau`'s from one or both trees before we can apply one of the corules `EqVis` or `EqTau`.

It is easy to show that `euttF` acts monotonically on relations, ensuring that a greatest fixed point exists, and it can be constructed using the `nu` operator. This operator and its associated theory are provided by the `paco` library [Hur et al., 2013]—where `nu` is named `paco2`, and more generally `pacoN` for every arity `N` of relations—which streamlines working with coinductive proofs in Coq.

Although the definition of heterogeneous weak bisimulation is fairly straightforward to state, some of its properties—for instance, transitivity and congruence with respect to `bind`—are quite challenging to prove. For these, we need an appropriate strengthening of the coinductive hypothesis that lets us reason about `eutt` up to closure under transitivity and `bind` contexts. Our Coq library actually uses a yet more

```
Context {E : Type → Type} {A B : Type} (r : A → B → Prop).

Inductive euttF (sim : itree E A → itree E B → Prop) : itree E A → itree E B →
   Prop :=
  (* Non-recursive rule *)
| EqRet a b (REL: r a b) : euttF sim (Ret a) (Ret b)
  (* Corules (coinductive): recursion via the sim parameter *)
| EqVis {R} (e : E R) k1 k2 (REL: ∀ v, sim (k1 v) (k2 v)) : euttF sim (Vis e k1) (
  Vis e k2)
| EqTau t1 t2 (REL: sim t1 t2) : euttF sim (Tau t1) (Tau t2)
  (* Rules: recursion via the inductive type euttF *)
| EqTauL t1 ot2 (REL: euttF sim t1 ot2) : euttF sim (Tau t1) ot2
| EqTauR ot1 t2 (REL: euttF sim ot1 t2) : euttF sim ot1 (Tau t2).

Lemma euttF_monotone t1 t2 sim sim' (IN: euttF sim t1 t2) (LE: sim <2= sim') : euttF
   sim' t1 t2.

Definition eutt : itree E A → itree E B → Prop := nu euttF.
```

FIGURE 2.4. Heterogeneous weak bisimulation for ITrees

general definition that subsumes both strong and weak bisimulation and builds in such "up-to" reasoning to make proofs smoother; we omit these details here and refer the interested reader to the Coq development itself. The upshot is that we can prove the following:

(1) $\cong$ *is an equivalence relation.*
(2) *If* `r` *is an equivalence relation, then so is* `eutt r`*.*
(3) $\approx$ *is an equivalence relation (corollary of (2)).*
(4) `t1` $\cong$ `t2` *implies* `t1` $\approx$ `t2`*.*

*Equational reasoning.* Fortunately, clients of the ITrees library can treat the definition of `eutt r` and its instances as black boxes—they never need to look at the coinductive machinery beneath this layer of abstraction. Instead, clients should reason equationally about ITrees. Figure 2.6 summarizes the most frequently used equations, each of which corresponds to a lemma proved in the library. The monad laws, structural laws, and congruences let us soundly rearrange an ITree computation—typically to put it into a form where a semantically interesting computation step, such as the interpretation of an event, takes place. Much of the functionality provided by the ITrees library involves lifting this kind of equational reasoning to richer settings, allowing us to work with combinations of different kinds of events and interpretations of their effects.

One pragmatic consideration is that Coq's `rewrite` and `setoid_rewrite` tactics, which let us rewrite using an equivalence (for instance, replacing the term `C[t1]` with `C[t2]` when we know that `t1` $\approx$ `t2`), only work if the context respects the equivalence. Coq's `Proper` type class registers congruence rules with the rewriting tactics. We prove such congruence rules for all of the operations in the library—some examples are shown in Figure 2.6. Some operators such as `bind` involve functions; they are related

| Symbol | Name | Domain |
|---|---|---|
| $\cong$ | Strong bisimulation | `itree E A` |
| $\approx$ | Weak bisimulation (`eutt`) | `itree E A` |
| $\hat{\approx}$ | Pointwise weak bisimulation | `A → itree E B` |

FIGURE 2.5. Summary of the main ITree relations and their notations

Monad Laws

```
      (x ← ret v ;; k x) ≅ (k v)
        (x ← t ;; ret x) ≅ t
(x ← (y ← s ;; t) ;; u) ≅ (y ← s ;; x ← t ;; u)
```

Structural Laws

```
              (Tau t) ≈ t
  (x ← (Tau t) ;; k) ≈ Tau (x ← t ;; k)
(x ← (Vis e k1) ;; k2) ≈ (Vis e (fun y ⇒ x ← k1 y ;; k2))
```

Congruences

```
                t1 ≅ t2 →      Tau t1 ≅ Tau t2
                k1 ≈̂ k2 →    Vis e k1 ≈ Vis e k2
      t1 ≈ t2 ∧ k1 ≈̂ k2 → bind t1 k1 ≈ bind t2 k2
```

FIGURE 2.6. Core equational theory of ITrees

```
Definition ktree (E : Type → Type) (A B : Type) : Type := A → itree E B.

Definition eq_ktree {E} {A B : Type} : ktree E A B → ktree E A B → Prop
  := fun h1 h2 ⇒ ∀ a, h1 a ≈ h2 a.

Infix "≈̂" := eq_ktree
```

FIGURE 2.7. Definition of KTrees and KTree equivalence

by the relation $\hat{\approx}$, defined in Section 2.1.5 as the pointwise lifting of weak bisimulation. Even so, definitions written in monadic style make heavy use of anonymous functions, which tend to thwart the `setoid_rewrite` tactic's ability to find the correct `Proper` instances. It is therefore useful to further raise the level of abstraction to simplify rewriting, as we show next.

**2.1.5. KTrees: Continuation Trees.** To improve equational reasoning principles and leverage known categorical structures for recursion, the ITrees library provides an abstraction for point-free definitions, centered around functions of the form `_ → itree E _`. We can think of these as *impure* functions that may generate events from `E` or possibly diverge. As we will show, they enjoy additional structure that we can exploit to generically derive more ways of composing ITrees computations.

We call types of the form `A → itree E B` *continuation trees*, or *KTrees* for short (Figure 2.7). Whereas an `itree E R` directly produces an outcome (`Ret`, `Tau`, or `Vis`), a KTree `k : ktree E A B` first expects some input `a : A` before continuing as an ITree `(k a)`. Equivalence on KTrees, written $\hat{\approx}$, is defined by lifting weak bisimulation pointwise to the function space.

```
Definition cat {E} {A B C : Type}
  : ktree E A B → ktree E B C → ktree E A C
  := fun h k ⇒ (fun a ⇒ bind (h a) k).

Infix ">>>" := cat
```

FIGURE 2.8. KTree composition

```
id_   : A → itree E A
cat   : (B → itree E C) → (A → itree E B) → (A → itree E C)
case_ : (A → tree E C) → (B → itree E C) → (A + B → itree E C)
inl_  : A → itree E (A + B)
inr_  : B → itree E (A + B)
pure  : (A → B) → (A → itree E B)
```

FIGURE 2.9. KTree operations

```
                         id_ >>> k ≈ k
                         k >>> id_ ≈ k
                   (i >>> j) >>> k ≈ i >>> (j >>> k)
                 pure f >>> pure g ≈ pure (f ∘ g)
                inl_ >>> case_ h k ≈ h
                inr_ >>> case_ h k ≈ k
(inl_ >>> f) ≈ h ∧ (inr_ >>> f) ≈ k  →  f ≈ case_ h k
```

FIGURE 2.10. Categorical laws for KTrees and handlers

Two KTrees `h : ktree E A B` and `k : ktree E B C` can be composed using `bind`; the result is written `(h >>> k) : ktree E A C` (Figure 2.8). KTree composition has a (left and right) identity, `id_` (equal to `ret`), and is associative; the proof follows from the monad laws for `itree`. Together, these facts mean that KTrees are the morphisms of a category: the *Kleisli category* [Mac Lane, 2013] of the monad `itree E`.

This category has more structure that we expose as part of the ITrees library interface. The `pure` operator lifts a Coq function trivially into an event-free KTree computation. We can also easily define an eliminator for the sum type, `case_` and corresponding left `inl_` and right `inr_` injections (effectful variants of the sum type constructors `inl` and `inr`). The names of those operations are suffixed with an underscore so as not to conflict with `id`, `inl`, and `inr` from the standard library, as well as for the visual uniformity of `case_` with `inl_` and `inr_`. These operations and their types are summarized in Figure 2.9. They satisfy the equational theory given in Figure 2.10.

The laws relating `case_`, `inl_`, and `inr_` mean that KTree is a *cocartesian category*. The Kleisli and cocartesian categorical structures are represented using type classes. These structures allow us to derive, generically, other useful operations and equivalences. For example, the following operations `bimap` and `swap` are defined from `case_`, `inl_`, and `inr_`. The KTree `bimap f g : ktree E (A + B) (C + D)` applies the KTree `f : ktree E A C` if its input is an `A`, or `g : ktree E B D` if its input is a `C`; the KTree `swap : ktree E (A + B) (B + A)` exchanges the two components of a sum. As we will see below, event handlers also have a cocartesian structure, which lets us re-use the same generic metatheory for them.

Similarly, the KTree category is just one instance of a Kleisli category, which can be defined for any monad `M`. Monadic event interpreters, introduced next, build on these structures, letting us (generically) lift the equational theory of KTrees to event interpreters too. This compositionality is important for scaling equational reasoning to situations involving many kinds of events.

### 2.2. Semantics of Events and Monadic Interpreters

To add semantics to the events of an ITree, we define an *event handler*, of type `E ⇝ M` for some monad `M`. Intuitively, it defines the meaning of an event of `E` as a monadic operation in `M`. An *interpreter* folds such an event handler over an ITree; a good interpretation of ITrees is one that respects `itree E`'s monadic structure (*i.e.*, it commutes with `ret` and `bind`).

Events and handlers enjoy a rich mathematical structure, a situation well known from the literature on algebraic effects (see Chapter 5). Our library exploits this structure to provide compositional reasoning principles and to lift the base equational theory of ITrees to their effectful interpretations.

**2.2.1. Example: Interpreting State Events.** Before delving into the general facilities provided by the ITrees library, it is useful to see how things play out in a familiar instance. The code in Figure 2.12 demonstrates how to interpret events into a state monad. The event type `stateE S` defines two events: `Get`, which yields an answer of state type `S`, and `putE`, which takes a new state of type `S` and yields `unit`.

In the figure, the state monad transformer operations `getT` and `putT` implement the semantics of reading from and writing to the state in terms of the underlying monad `M`, using its `ret`. The function `handle_state` is a *handler* for `stateE` events: it maps events of type `stateE S R` into monadic computations of type `stateT S (itree E) R`, *i.e.*, `S → itree E (S * R)`, taking an input state to compute an output state and a result. Given this handler, we define the `interp_state` function, which folds the handler across all of the visible events of an ITree of type `itree (stateE S) R` to produce a semantic function of type `stateT S (itree E) R`. The definition of `interp_state` is an instance of `interp` (see Section 2.2.2 below), specialized to a state monad.

To prove properties about the resulting interpretation, we need to show that `interp_state` is a *monad morphism*, meaning that it respects the `ret` and `bind` operations of the ITree monad.

```
interp_state (ret x) s  ≈  ret (s, x)
interp_state (x ← t ;; k x) s1
  ≈ '(s2, x) ← interp_state t s1 ;; interp_state (k x) s2
```

We next prove that `handle_state` implements the desired behaviors for the `get` and `put` operations, which are shorthands for the `trigger` of the correponding `stateE` events.

```
interp_state get s ≈ ret (s,s)
interp_state (put s') s ≈ ret (s',tt)
```

```
Definition interp {E M} `{MonadIter M} {R : Type} (handler : E ⤳ M)
  : itree E R → M R := iter (fun t : itree E R ⇒
      match t with
      | Ret r   ⇒ ret (inr r)
      | Tau t   ⇒ ret (inl t)
      | Vis e k ⇒ bind (handler _ e) (fun a ⇒ ret (inl (k a)))
      end).
```

FIGURE 2.11. Interpreting events via a handler

```
(* The type of state events *)
Variant stateE (S : Type) : Type → Type :=
| Get : stateE S S
| Put : S → stateE S unit.

(* State monad transformer *)
Definition stateT (S:Type) (M:Type → Type) (R:Type) : Type := S → M (S * R).
Definition getT (S:Type) : stateT S M S := fun s ⇒ ret (s, s).
Definition putT (S:Type) : S → stateT S M unit := fun s' s ⇒ ret (s', tt).

(* Handler for state events *)
Definition h_state (S:Type) {E} : (stateE S) ⤳ stateT S (itree E) :=
  fun _ e ⇒ match e with
            | Get   ⇒ getT S
            | Put s ⇒ putT S s
            end.

(* Interpreter for state events *)
Definition interp_state {E S} : itree (stateE S) ⤳ stateT S (itree E) :=
  interp h_state.
```

FIGURE 2.12. Interpreting state events

These equations allow us to use `put` and `get`'s semantics when reasoning about stateful computations. They are also sufficient to derive useful equations when verifying program optimizations—for instance, we can remove a redundant `get` as follows:

```
  interp_state (x ← get ;; y ← get ;; k x y) s
≈ interp_state (x ← get ;; k x x) s
```

**2.2.2. Monadic Interpreters.** The `interp_state` function above is an instance of a general `interp` function that is defined for any monad `M`, provided that `M` supports an *iteration operator*, `iter`, of type `(A → M (A + B)) → A → M B`. (The first argument is a loop body that takes an `A` and produces either another `A` to keep looping with or a final result of type `B`.) Figure 2.11 shows the definition of `interp`. It takes a `handler : E ⤳ M` and loops over a tree of type `itree E R`. At every iteration, the next constructor of the tree is interpreted, using `handler` for `Vis` constructors, and yielding the remaining tree as a new loop state.

```
id_   : E ⇝ itree E                                        (* trigger *)
cat   : (F ⇝ itree G) → (E ⇝ itree F) → (E ⇝ itree G)    (* interp *)
case_ : (E ⇝ itree G) → (F ⇝ itree G) → (E +' F ⇝ itree G)
inl_  : E ⇝ itree (E +' F)
inr_  : F ⇝ itree (E +' F)
```

FIGURE 2.13. Event handler operations

```
interp h (trigger e)     ≅ h _ e
interp h (Ret r)         ≅ ret r
interp h (x ← t;; k x) ≅ x ← interp h t;; interp h (k x)
```

FIGURE 2.14. Some properties of `interp`

The core properties of `interp`, summarized in Figure 2.14, are generalizations of the laws for `interp_state`. In particular, `interp` preserves the monadic structure of ITrees, and its action on `trigger e` is to apply the handler to the event `e`.

It remains to show how to instantiate the `MonadIter` type class, which provides the `iter` combinator used by `interp`. We defer this discussion to Section 2.3, as it will benefit from a closer look at events and handlers.

### 2.2.3. The Algebra of Events and ITree Event Handlers.

The `handle_state` handler interprets computations with events drawn from the specific type `stateE S`. More generally, we often want to combine multiple kinds of events in one computation. For instance, we might want both `stateE S` and `IO` events, or access to two different types of state at the same time. Fortunately, it is straightforward to define `E +' F`, the disjoint union of the events `E` and `F`. The definition comes with inclusion operations `inl1 : E ⇝ E +' F` and `inr1 : F ⇝ E +' F`.[1] The `emptyE` event type, with no events, is the unit of `+'`.

The corresponding operations on handlers manipulate sums of event types: `case_` combines handlers for different event types into a handler on their sum, while `inl_` and `inr_` are `inl1` and `inr1` turned into event handlers.

```
Definition case_ {E F M} : (E ⇝ M) → (F ⇝ M) → (E +' F) ⇝ M
  := fun f g _ e ⇒ match e with
                    | inl1 e1 ⇒ f _ e1
                    | inr1 e2 ⇒ g _ e2
                    end.

Definition inl_ {E F} : E ⇝ itree (E +' F)
Definition inr_ {E F} : F ⇝ itree (E +' F)
```

Recall that the general type of an event handler is `E ⇝ M`. When `M` has the form `itree F`, we can think of such a handler as *translating* the `E` events into `F` events. We call handlers of this type *ITree event handlers*. Like KTrees, event handlers form

[1]The 1 in `inl1` and `inr1` reminds us that `E` and `F` live in `Type → Type`.

```
iter : (A → itree E (A + B)) → (A → itree E B)
loop : (C + A → itree E (C + B)) → A → itree E B
mrec : (E ⇝ itree (E +' F)) → (E ⇝ itree F)
```

FIGURE 2.15. Summary of recursion combinators

a cocartesian category where composition of handlers uses `interp`, and the identity handler is `trigger`. The interface is summarized in Figure 2.13.

```
Definition cat {E F G}
  : (E ⇝ itree F) → (F ⇝ itree G) → (E ⇝ itree G)
  := fun f g _ e ⇒ interp g (f _ e).

Definition id_ {E} : E ⇝ itree E := @trigger E.
```

The equivalence relation for handlers `h ≊ k` is defined below, as pointwise weak bisimulation. It admits the same equational theory (and derived constructs) as for KTrees, hence we reuse the same notations for the operations (see Figure 2.10).

```
h ≊ k := ∀ A (e: E A), (h A e) ≈ (g A e)
```

*Subevents.* When working with ITrees at scale, it is often necessary to connect ITrees with fewer effects to ITrees with more effects. For instance, suppose we have an ITree `t : itree IO A` and we want to `bind` it with a continuation `k` of type `A → itree (X +' IO +' Y) B` for some event types `X` and `Y`. A priori, this isn't possible, since the types of their events don't match. However, since there is a natural structural inclusion `inc: IO ⇝ X +' IO +' Y` (given by `inl_ ∘ inr_`) we can first interpret `t` using the handler `fun e ⇒ trigger (inc e)` and then bind the result with `k`.

Since the need for such structural inclusions arises fairly often, the ITrees library defines a type class, written `E -< F`, that can automatically synthesize inclusions such as `inc`. It generically derives an instance of `trigger : E ⇝ itree F` whenever there is a structural subevent inclusion `E ⇝ F`. We will see in the case study how this flexibility is useful in practice.

## 2.3. Iteration and Recursion

While Coq does provide support for coinduction and corecursion, its technique for establishing soundness relies on syntactic mechanisms that are not compositional. To make working with ITrees more tractable to clients, our library provides *first-class* abstractions to express corecursion as well as reasoning principles for these abstractions that hide the brittle nature of Coq's coinduction. From the point of view of a library user, recursive definitions using these combinators need only to typecheck, even when they lead to divergent behaviors.

Our library exports two iteration constructs, `iter` and `loop`, and a recursion combinator `mrec` (Figure 2.15). They are mutually inter-derivable, but they permit rather distinct styles of recursive definitions.

```
CoFixpoint iter (body : A → itree E (A + B))
  : A → itree E B :=
  fun a ⇒ ab ← body a ;;
          match ab with
          | inl a ⇒ Tau (iter body a)
          | inr b ⇒ Ret b
          end.

Definition loop (body : C + A → itree E (C + B))
    : A → itree E B :=
  fun a ⇒ iter (fun ca ⇒
            cb ← body ca ;;
            match cb with
            | inl c ⇒ Ret (inl (inl c))
            | inr b ⇒ Ret (inr b)
            end) (inr a).
```

FIGURE 2.16. Iteration combinators: `iter` and `loop`

**2.3.1. Iteration.** The first function is a combinator for *iteration*, `iter`, whose implementation is shown in Figure 2.16. Given `body : A → itree E (A + B)` and a starting state `a:A`, `iter body a` is a computation that produces either a new state from which to iterate the `body` again (after a `Tau`), or a final value to stop the computation. This operator makes no assumption on the shape of the loop body, a marked improvement over the intensional guardedness check required by `cofix`.

Defining fixpoint combinators as functions allows us to prove their general properties *once and for all*. The equations for `iter`, given in Lemma 2.3.1 and graphically in Figure 2.17, imply that continuation trees form an iterative category [Bloom and Ésik, 1993]. The *fixed point identity* unfolds one iteration of the `iter` loop; the *parameter identity* equates a loop followed by a computation with a loop where that computation is part of its last iteration; the *composition identity* equates a loop whose body sequences two computations `f`, `g` with a loop sequencing them in reverse order, prefixed by a single iteration of `f`; and the *codiagonal identity* merges two nested loops into one.

**Lemma 2.3.1** (Iterative category)**.**

```
                iter f  ≈  f >>> case_ (iter f) id_                 (fixed point)
          iter f >>> g  ≈  iter (f >>> bimap id_ g)                   (parameter)
iter (f >>> case_ g inr_)  ≈  f >>> case_ (iter (g >>> case_ f inr_)) id_   (comp.)
         iter (iter f)  ≈  iter (f >>> case_ inl_ id_)              (codiagonal)
```

The proofs of these equations makes nontrivial use of coinductive reasoning for weak bisimulation; carrying them out in a proof assistant is a significant contribution of this work. Nevertheless, that complexity is entirely hidden from users of the library, behind the simple interface exposed by these equations, whose expressiveness we'll demonstrate in our case study in Chapter 3.

The `iter` implementation shown in Figure 2.16 is specialized to the `itree E` monad. However, we can generalize to other monads and characterize the abstraction using the following type class: [2]

```
Class MonadIter (M : Type → Type) `{Monad M} :=
  iter : ∀ A B, (A → M (A + B)) → A → M B.
```

Good implementations of the `MonadIter` interface must satisfy the iterative laws. In a total language such as Coq, this limits the possible implementations. Base instances include ITrees and the predicate monad (`_ → Prop`) where we can tie the knot using the impredicative nature of `Prop`. In addition, we can lift `MonadIter` through a wide variety of monad transfomers, *e.g.*, `stateT S M` where `M` is an instance of `MonadIter`.

*Traced categories.* Figure 2.16 also shows the `loop` combinator, an alternative presentation of recursion that is derivable from `iter`. We can think of the `C` part of the body's input and output types as input and output "ports" that get patched together with a "back-edge" by `iter`. We use `loop` in Chapter 3 to model linking of control-flow graphs. This `loop` combinator equips KTrees with the well-studied structure of a *traced monoidal category* [Joyal et al., 1996, Hasegawa, 1997].

**2.3.2. Recursion.** The ITree library also features combinators for directly expressing recursion, using a technique developed by McBride [2015] to represent recursive calls as events. Figure 2.18 shows the code for a general mutual-recursion combinator, `mrec`. Here, an indexed type `D : Type → Type` gives the signature of a recursive function, or, using multiple constructors, a block of mutually recursive functions. For example, `D := ackermannE` represents a function with two `nat` arguments and a result of type `nat`.

```
Inductive ackermannE : Type → Type :=
| Ackermann : nat → nat → ackermannE nat.
```

A *recursive event handler* for `D` is an event handler of type `D ⇝ itree (D +' E)`, so it can make recursive calls to itself via `D` events, and perform other effects via `E` events. As an example, the handler `h_ackermann` pattern-matches on the event `Ackermann m n` to extract the two arguments of the function, and to refine the result type to `nat`. The body of the function makes recursive calls by `trigger`-ing `Ackermann` events, without any requirement to ensure the well-foundedness of the definition.

```
Definition h_ackermann : ackermannE ⇝ itree (ackermannE +' emptyE) :=
  fun _ '(Ackermann m n) ⇒
    if m =? 0 then Ret (n + 1)
    else if n =? 0 then trigger (inl1 (Ackermann (m-1) 1))
    else (ack ← trigger (inl1 (Ackermann m (n-1))) ;;
          trigger (inl1 (Ackermann (m-1) ack))).
```

[2]In Haskell, there is a class `MonadFix` also associated with recursion and monads, with method `mfix : (A → M A) → M A`. However, `MonadIter` and `MonadFix` have quite different purposes. Whereas `MonadIter` constructs a *computation* as a fixed point, `MonadFix` constructs a *value* as a fixed point, which is both the input and the output of the computation. A computation is *iterated* by `iter`, as exemplified by the unfold law, whereas `mfix` only runs its given computation once, as demonstrated by the law `mfix (const u) = u`.

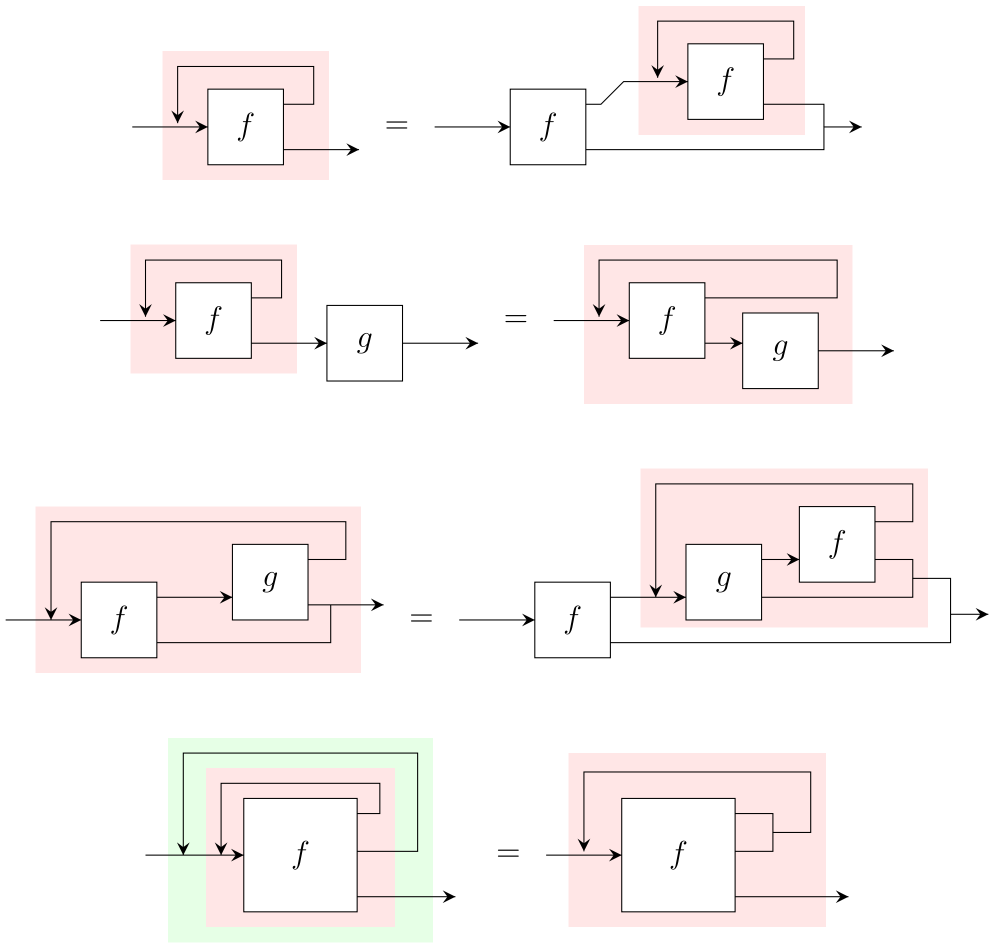

(Colored boxes denote applications of `iter`.)

FIGURE 2.17. Diagrammatic representation of the laws of iterative categories (Lemma 2.3.1)

The `mrec` combinator ties the knot. Given a recursive handler `D ⇝ itree (D +' E)`, it produces a handler `D ⇝ itree E`, where all `D` events have been handled recursively.

```
Definition ackermann : nat → nat → itree emptyE nat :=
  fun m n ⇒ mrec h_ackermann (Ackermann m n).
```

The implementation of `mrec` in Figure 2.18 works similarly to `interp`, applying the recursive handler `rh` to events in `D`. However, whereas `interp` directly uses the ITree produced by the handler as output, `mrec` adds it as a prefix of the ITree to be interpreted recursively: the `inl1 d` branch returns `bind (rh _ d) k`, which will be processed in subsequent steps of the `iter` loop.

For reasoning, `mrec` is also characterized as a fixed point by an *unfolding equation*, which applies the recursive handler `rh : D ⇝ itree (D +' E)` once, and interprets the resulting ITree with `interp`, where `D` events are passed to `mrec` again, and `E` events are passed to the identity handler, *i.e.*, `trigger`, which keeps events uninterpreted.

```
(* Interpret an itree in the context of a mutually recursive definition (rh) *)
Definition mrec {D E} (rh : D ⇝ itree (D +' E)) : D ⇝ itree E :=
  fun R d ⇒ iter (fun t : itree (D +' E) R ⇒
    match t with
    | Ret r ⇒ Ret (inr r)
    | Tau t ⇒ Ret (inl t)
    | Vis (inl1 d) k ⇒ Ret (inl (bind (rh _ d) k))
    | Vis (inr1 e) k ⇒ bind (trigger e) (fun x ⇒ Ret (inl (k x)))
    end) (rh _ d).
```

FIGURE 2.18. Mutual recursion via events

```
mrec rh d ≈ interp (case_ (mrec rh) id_) (rh d)
```

In fact, `mrec` is an analogue of `iter`, equipping event handlers themselves with the structure of an iterative category. It satisfies the same equations as `iter`, relating event handlers instead of KTrees.

## 2.4. Extracting ITrees

One of the big benefits of ITrees is that they work well with Coq's extraction facilities. If we extract the `echo` definition from Section 2, we obtain the code shown at the top of Figure 2.19.[3] The `itree` type extracts to a lazy datatype, and `observe` forces its evaluation.

To actually run the represented computation, we provide a driver that traverses the `itree`, forcing all of its computation and providing handlers for any visible events that remain in the tree. The OCaml function `run` does exactly that, where, for the sake of this example, we interpret each `Input` event as a call to OCaml's `read_int` command and each `Output` event as a call to `print_int`.[4] This kind of simple event handling already suffices to add basic IO and "printf debugging" to Coq programs, which can be extremely handy in practice. We can, of course, implement more sophisticated event handlers, using the full power of OCaml.

ITrees extractability has played a key role in several different parts of an ongoing research project that seeks to use Coq for *Deep Specifications*.[5] In particular, our re-implementation of the VELLVM [Zakowski et al., 2021] formalization of LLVM, which aims to give a formal semantics for the LLVM IR in Coq, heavily uses ITrees exactly as proposed in this dissertation to build a denotational semantics for LLVM IR code. The Vellvm semantics has many layers of events and handlers (for global data, local data, interactions with the memory model, internal and external functions calls, *etc.*), and the LLVM IR control-flow graphs are a richer version of the ASM language (Section 3.2). The flexibility of using ITree-based interpreters means that

[3]For simplicity, here we also extract Coq's `nat` type as OCaml's `int` type.

[4]Thanks to its dependent type, the OCaml extraction of `Vis` uses OCaml's `Obj.t` as the domain of the embedded continuations, so handlers should be written with care, otherwise type-safety could be jeopardized.

[5]`http://www.deepspec.org`.

```
let rec echo =
  lazy (Vis (Input, fun x ->
  lazy (Vis (Output (Obj.magic x), fun _ -> echo))))

(* OCaml handler (manually written, not extracted code) *)
let handle_io e k = match e with
  | Input -> k (Obj.magic (read_int ()))
  | Output x -> print_int x ; k (Obj.magic ())

let rec run t =
  match observe t with
  | Ret r -> r
  | Tau t -> run t
  | Vis (e, k) -> handle_io e (fun x -> run (k x))
```

FIGURE 2.19. OCaml extracted from the `echo` example (top) and OCaml handler and "driver" loop (bottom)

Vellvm can define a relational specification that accounts for nondeterministic features of LLVM (such as `undef`) but that can also be refined into an implementation. We are able to extract an executable interpreter that performs well enough to test small- to medium-sized LLVM code samples (including recursion, loops, *etc.*). All but the outermost `run` driver are extracted from Coq, as in the `echo` example.

ITrees are also used as executable specifications to model the semantics of web servers [Koh et al., 2019, Li et al., 2021]. The ITree representation serves two purposes: (1) ITrees model the interactive operations of the web server in a way that can be connected via Princeton's VST framework [Appel, 2014, 2011] to a C implementation, and (2) the model can also be used for property-based testing with QuickChick [Lampropoulos and Pierce, 2018]. The ability to link against handlers written in OCaml means that the testing framework can be used to test real web servers like Apache across the network, in addition to linking against our own web servers. Here again, the performance of the extracted executable has been good enough that we have felt no need to do any optimization on the ITree representation.

CHAPTER 3

# Application I: Compiling Imp to Asm

To demonstrate the compositionality of ITree-based semantics and the usability of our Coq library, we use ITrees to formalize and verify a compiler from a variant of the Imp language from *Software Foundations* [Pierce et al., 2018] to a simple assembly language, called Asm. Subsequent work in the Vellvm semantics for LLVM [Zakowski et al., 2021] and the Helix compiler to LLVM [Zaliva et al., 2020] are based on the ideas from this chapter.

We begin by explaining the denotational semantics of Imp (Section 3.1) and Asm (Section 3.2). It is convenient to define the semantics in stages, each of which justifies a different notion of program equivalence. The first stage maps syntax into ITrees, thereby providing meaning to the control-flow constructs of the language, but not ascribing any particular meaning to the events corresponding to interactions with the memory. The second stage interprets those events as effects that manipulate (a representation of) the actual program state.

We then give a *purely inductive* proof of the correctness of the compiler (Sections 3.3 and 3.4). The denotational model enables us to state a termination-sensitive bisimulation and prove it purely equationally, naturally extending a straightforward approach for terminating languages with simpler denotational semantics [McCarthy and Painter, 1967]. The correctness proof relates the semantics of Imp to the semantics of Asm *after* their events have been appropriately interpreted into state monads (a necessity, since compilation introduces new events that correspond to reading and writing intermediate values). Since Imp programs manipulate one kind of state (global variables) and Asm programs manipulate two kinds of state (registers and the heap), the proof involves building an appropriate simulation relation between Imp states and Asm states.

To streamline, we identify Imp global variables with Asm heap addresses and assume that Imp and Asm programs manipulate the same kinds of dynamic values. Neither assumption is critical.

## 3.1. A Denotational Semantics for Imp

The syntax and the semantics for Imp is given in Figure 3.1. In the absence of `while`, a denotational semantics could be defined by structural recursion on statements, as a function from an initial environment to a final environment; the denotation function would have type `imp → env → (env * unit)`. However, it is not possible to give a semantics to `while` using this naïve denotation because Gallina's function space is total. The usual solution is to revamp the semantics dramatically, *e.g.*, by moving to a relational operational semantics (Section 5.1.5 discusses other approaches). With ITrees, the denotation type becomes `imp → stateT env (itree F) unit`, or,

```
(* Imp Syntax ------------------------------ *)
Inductive expr : Set := ... (* omitted *)

Inductive imp : Set :=
| Assign (x : var) (e : expr)
| Seq    (a b : imp)
| If     (i : expr) (t e : imp)
| While  (t : expr) (b : imp)
| Skip.

(* Imp Events ------------------------------ *)
Variant ImpState : Type → Type :=
| GetVar (x : var) : ImpState value
| SetVar (x : var) (v : value) : ImpState unit.
```

FIGURE 3.1. Syntax of IMP

equivalently, `imp → env → itree F (env * unit)`, which allows for nontermination. It is also more flexible, since the semantics can be defined generically with respect to an event type parameter `F`, which can later be refined if new effects are added to the language or if we want to compose ITrees generated as denotations of IMP programs with ITrees obtained in some other way.

Figure 3.2 shows how the IMP semantics are structured. We first define `denote_expr` and `denote_imp`, which result in trees of type `itree E unit`. The type class constraint `ImpState -< E` indicates that `E` permits `ImpState` actions, a refinement of `stateE` that provides events for reading and writing *individual* variables; we would follow the same strategy to add other events such as IO. The meanings of expressions and most statements are straightforward, except for `While`. This relies on the `iter` combinator (see Section 2.3) to first run the guard expression, then either continue to loop (by returning `inl tt` to the `iter` combinator) or signal that it is time to stop (by returning `inr tt`).

The second stage of the semantics is `interp_imp`, which takes ITrees containing `ImpState` events and produces a computation in the state monad. It first invokes a handler for `ImpState`, `h_imp_state`, to translate the IMP-specific `GetVar` and `SetVar` events into the general-purpose `mapE` events provided by the ITrees library (the `bimap` operator propagates other events untouched). It then uses `interp_map` to define their meaning in terms of actual lookup and set operations on the type `env`, a simple finite map from `var` to `value`. The final semantics of an IMP statement `s` is obtained simply by composing the two functions: `interp_imp (denote_imp s)`.

Factoring the semantics this way is useful for proofs. For instance, to prove the soundness of a syntactic program transformation from `s` to `s'` it suffices to show that `denote_imp s ≈ denote_imp s'`; we need not necessarily consider the impact of `interp_imp`. We will exploit this semantic factoring in the compiler proof below by reasoning about syntactic "linking" of ASM code before its state-transformer semantics is considered.

```
Context {E : Type → Type} `{ImpState -< E}.

(* Imp Denotational semantics ------------------- *)
(* ITree representing an expression *)
Fixpoint denote_expr (e:expr) : itree E value :=
 match e with
 | Var v     ⇒ trigger (GetVar v)
 | Lit n     ⇒ ret n
 | Plus a b  ⇒ l ← denote_expr a ;;
               r ← denote_expr b ;; ret (l + r)
 | ...
 end.

(* Imp Denotational semantics cont'd --------------------- *)
(* ITree representing an Imp statement *)
Fixpoint denote_imp (s : imp) : itree E unit :=
  match s with
  | Assign x e   ⇒ v ← denote_expr e ;; trigger (SetVar x v
    )
  | Seq a b      ⇒ denote_imp a ;; denote_imp b
  | If i t e     ⇒ v ← denote_expr i ;;
      if is_true v then denote_imp t else denote_imp e
  | While t b    ⇒
      iter (fun _ ⇒ v ← denote_expr t ;;
                      if is_true v
                      then denote_imp b ;; ret (inl tt)
                      else ret (inr tt))
  | Skip         ⇒ ret tt
  end.

(* Imp state monad semantics ----------------------------- *)
(* Translate ImpState events into mapE events *)
Definition h_imp_state {F: Type → Type} `{mapE var 0 -< F}
  : ImpState ⇝ itree F := ...(* omitted *)

(* Interpret ImpState into (stateT env (itree F)) monad *)
Definition interp_imp {F A} (t : itree (ImpState +' F) A)
  : stateT env (itree F) A :=
  let t' := interp (bimap h_imp_state id_) t in
    interp_map t'.
```

FIGURE 3.2. Semantics of IMP

This style of denotational semantics avoids defining a syntactic representation of machines, which often comes with cumbersome administrative reduction rules. Instead, we directly manipulate denotations to implement the control-flow structures of the language.

```
(* Asm syntax ----------------------------------- *)
Variant instr : Set := ... (* omitted *)

Variant branch {label : Type} : Type :=
| Bjmp (_ : label)                (* jump to label *)
| Bbrz (_ : reg) (yes no : label) (* cond. jump *)
| Bhalt.

Inductive block (label : Type) : Type :=
| bbi (_ : instr) (_ : block)   (* instruction *)
| bbb (_ : branch label).        (* final branch *)

Definition bks A B := fin A →  block (fin B).

(* Control-flow subgraph: entries A and exits B. *)
Record asm (A B : nat) : Type :=
{ internal : nat
; code      : bks (internal + A) (internal + B) }.

(* Asm events ----------------------------------- *)
Variant Reg : Type →  Type :=
| GetReg (x : reg) : Reg value
| SetReg (x : reg) (v : value) : Reg unit.

Inductive Memory : Type →  Type :=
| Load   (a : addr) : Memory value
| Store (a : addr) (v : value) : Memory unit.
```

FIGURE 3.3. Syntax of ASM

## 3.2. A Denotational Semantics for Asm

The target of our compiler is ASM, a simple assembly language that represents computations as collections of basic blocks linked by conditional or unconditional jumps. Figure 3.3 gives the core syntax for the language, which is split into two levels: basic blocks and control-flow subgraphs.

A *basic block* (`block`) is a sequence of straight-line instructions followed by a branch that transfers control to another block indicated by a label. As with IMP expressions, the denotation of instructions is mostly uninteresting, so they are omitted in the figure.

A *control-flow subgraph*, or "sub-CFG" (`asm` in Figure 3.3), represents the control flow of a computation. These are *open* program fragments, represented as sets of labeled basic blocks. The labels in a sub-CFG are separated into three groups: *entry labels*, from which the code in a sub-CFG can start executing; *exit labels*, where the control flow leaves the sub-CFG; and *internal labels*, which are invisible outside of the subgraph. A sub-CFG has a block of code for every entry and internal label; control leaves the subgraph by jumping to an exit label. Labels are drawn from finite domains,

```
Context {E : Type → Type} `{Reg -< E} `{Memory -< E}.

(* Asm denotational semantics --------------------- *)
Definition denote_instr (i:instr) : itree E unit
  := ... (* omitted *)
(* Asm denotational semantics cont'd -------------------- *)
Definition denote_br (b:branch (fin B)):itree E (fin B) :=
  match b with
  | Bjmp l ⇒ ret l
  | Bbrz v y n ⇒
      val ← trigger (GetReg v) ;;
      if val ?= 0 then ret y else ret n
  | Bhalt ⇒ exit
  end.

Fixpoint denote_bk {L} (b : block L) : itree E L :=
  match b with
  | bbi i b ⇒ denote_instr i ;; denote_bk b
  | bbb b    ⇒ denote_br b
  end.

Definition denote_bks (bs:bks A B):ktree E (fin A) (fin B)
  := fun a ⇒ denote_bk (bs a).

Definition den_asm {A B}:asm A B → ktree E (fin A) (fin B)
  := fun s ⇒ loop (denote_bks (code s)).

(* Asm state monad semantics ---------------------------- *)
Definition h_reg {F: Type → Type} `{mapE reg 0 -< F}
  : Reg ⇝ itree F := (* omitted *)
Definition h_mem {F: Type → Type} `{mapE addr 0 -< F}
  : Memory ⇝ itree F := (* omitted *)

Definition interp_asm {F A} (t:itree (Reg+'Memory+'F) A)
  : memory → registers → itree F (memory*(registers*A)) :=
  let h := bimap h_reg (bimap h_mem id_) in
  let t' := interp h t in
  fun mem regs ⇒ interp_map (interp_map t' regs) mem.
```

FIGURE 3.4. Semantics of ASM

*e.g.*, `fin A` and `fin B` where `A` and `B` are `nat`s, as seen in `bks`. The finiteness of labels is useful for ASM program transformations and is faithful to "real" assembly code, but this restriction is not actually necessary. The correctness proof is independent of this choice, so we sweep the details under the rug.

Figure 3.4 presents the denotation of ASM programs, which factors into two parts, just as we saw for IMP. The `GetReg` and `SetReg` events represent accesses of register state, and Load and Store accesses of memory.[6] Once again, we give meaning to the control-flow constructs of the syntax independently of the state events. The result of denote_bks is a ktree that maps each entry label to an itree that returns the label of the next block to jump to. The den_asm function first computes the denotation of each basic block and then wires the blocks together using loop, hiding the internal labels in the process.

The stateful semantics of Asm programs is given by interp_asm, which, like interp_imp, realizes the register and memory as finite maps using interp_map. As a result of this nesting, the "intermediate state" of an Asm computation is a value of type memory * (register * A). Because they are built compositionally from interpreters, it is very easy to prove that both interp_imp and interp_asm are monad morphisms in the sense that they commute (up to Tau) with ret and bind, a fact that enables proofs by rewriting.

## 3.3. Linking of Control-Flow Subgraphs

We now turn to the compilation of Imp to Asm. The compiler and its proof are each split into two components. The first phase handles reasoning about control flow by embedding sub-CFGs into KTrees. In the second phase, we perform the actual compilation and establish its functional correctness by reasoning about the quotienting of the local events.

For the first phase, we first implement a collection of reusable combinators for linking sub-CFGs. These combinators correspond to the operations on KTrees described in Section 2.1.5, which can be seen in this context as presenting a theory of graph linking at the denotational level. Here are the signatures of the four essential ones (their implementations are straightforward):

```
Definition app_asm       (ab : asm A B) (cd : asm C D) : asm (A + C) (B +
   D).
Definition loop_asm      (ab_ : asm (I + A) (I + B))      : asm A B.
Definition pure_asm      (f : A → B)                       : asm A B.
Definition relabel_asm  (f : A → B) (g : C → D) (bc : asm B C) : asm A D
    .
```

Two sub-CFGs can be placed beside one another while preserving their labels, via app_asm. *Linking* of compilation units is performed by loop_asm: it connects a subset of the exit labels I as back edges to the imported labels, also named I, and internalizes them. Visible labels can be renamed with relabel_asm. Finally, pure_asm creates, for every label a : A, a block that jumps immediately to f a. Together, relabel_asm and pure_asm provide the plumbing required to use the combinators app_asm and loop_asm effectively.

The correspondence between these core Asm combinators and operations on KTrees is given by the following equations, which commute the denotation function inside the combinator.

[6]An additional Done event (not shown) represents halting the whole program for blocks terminated by a Bhalt instruction via exit, but we omit it for the purposes of this exposition.

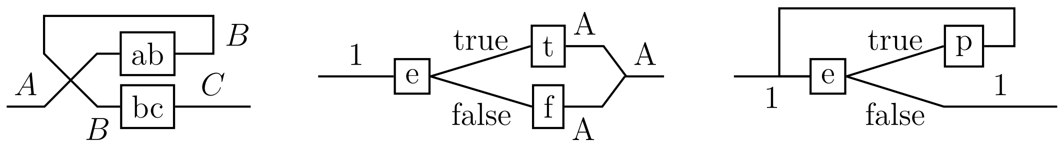
```
Definition seq_asm {A B C} (ab : asm A B) (bc : asm B C): asm A C
  :=  loop_asm (relabel_asm swap id_ (app_asm ab bc)).

(* Auxiliary for if_asm *)
Definition cond_asm (e : list instr) : asm 1 (1 + 1)
  := ... (* omitted *)

Definition if_asm {A} (e : list instr) (t : asm 1 A) (f : asm 1 A)
  : asm 1 A
  :=  seq_asm (cond_asm e) (relabel_asm id_ merge (app_asm t f)).

Definition while_asm (e : list instr) (p : asm 1 1) : asm 1 1
  :=  loop_asm (relabel_asm id_ merge
                 (app_asm (if_asm e (relabel_asm id inl_ p) (pure_asm inr_))
                          (pure_asm inl_))).
```

(A) Coq definitions



(B) Graphical representation

Figure 3.5. High-level control flow in Asm

```
   den_asm (app_asm ab cd)  ≈̂  bimap (den_asm ab) (den_asm cd)
      den_asm (loop_asm ab)  ≈̂  loop (den_asm ab)
den_asm (relabel_asm f g bc)  ≈̂  (pure f >>> den_asm bc >>> pure g)
       den_asm (pure_asm f)  ≈̂  pure_ktree f
```

Equipped with these primitives, building more complex control-flow graphs becomes a diagrammatic game. Figure 3.5 shows how to use the primitives to build linking operations for sub-CFGs that mimic the control-flow operations provided by Imp. For instance, sequential composition of `asm A B` with `asm B C` places them in parallel, swaps their entry labels to get a sub-CFG of type `asm (B+A) (B+C)`, and then internalizes the intermediate label `B` via `loop_asm`.

We emphasize that while these control-flow graphs are specific to Imp, their definitions do not depend on Imp's or even Asm's state-transformer semantics. We can reason about control-flow independently of other events. For instance, the denotation of the `seq_asm` combinator is indeed the sequential composition of denotations of its arguments (up to `Tau`):

```
Lemma seq_asm_correct {A B C} (ab : asm A B) (bc : asm B C) :
  (den_asm (seq_asm ab cd)) ≈̂ (den_asm ab >>>  den_asm bc).
```

The `while_asm` combinator is more involved, naturally. As illustrated in Figure 3.5, it constructs the control-flow graph of a `while` loop given the list of instructions for the test condition and the compilation unit corresponding to the body of the loop. The type of `p` represents a compilation unit with a single imported label (the target to jump to when the loop body finishes) and a single exported label (the entry label for the top of the loop body). The correctness of the combinator establishes that its denotation can be viewed as an entry point that runs the body if a variable `tmp_if` is non-zero after evaluating the expression `e`. This is expressed at the level of KTrees via the `loop` operator. In the code below, `label_case l` analyzes the shape of the label `l`, and `l1` and `l2` are two distinct label constants corresponding to the loop entry or exit, respectively.

```
Lemma while_asm_correct (e : list instr) (p : asm 1 1)
  : denote_asm (while_asm e p)
  ≈ loop (fun l:fin (1 + 1) ⇒
       match label_case l with
       | inl _ ⇒   denote_list e ;; v ← trigger (GetReg tmp_if) ;;
                   if (v:value) then Ret l2 else (denote_asm p l1;; Ret l1)
       | inr _ ⇒  Ret l1
       end).
```

Most importantly, the proof of `while_asm_correct` is again purely equational, relying solely on the theory of KTrees and correctness equations of the low-level linking combinators (`app_asm_correct`, `seq_asm_correct`, *etc.*).

## 3.4. Compiler Correctness

The compiler itself is entirely straightforward. It compiles Imp statements using the linking combinators along with `compile_assign` and `compile_expr`, both of which are simple (and omitted). We pass to `compile_expr` the name of a target register (here, just `0`), into which the value of the expression will be computed; it stores intermediate results in additional Asm registers as needed.

```
Fixpoint compile (s : stmt) {struct s} : asm 1 1 :=
  match s with
  | Skip         ⇒  id_asm
  | Assign x e   ⇒  raw_asm_block (after (compile_assign x e) (Bjmp l1))
  | Seq l r      ⇒  seq_asm (compile l) (compile r)
  | If e l r     ⇒  if_asm (compile_expr 0 e) (compile l) (compile r)
  | While e b    ⇒  while_asm (compile_expr 0 e) (compile b)
  end.
```

The top-level compiler correctness theorem is phrased as a bisimulation between the Imp program and the corresponding Asm program, which simply requires them to have "equivalent" behavior.

```
Theorem compile_correct (s : stmt) :  equivalent s (compile s).
```

Figure 3.6 unpacks the definition of `equivalent`, which requires the ITree denotations of `s` and its compilation to be bisimilar. Two ITrees `t1` and `t2`, representing Imp and Asm computations respectively, of types `itree (ImpState +' E) A` and `itree (Reg +' Memory +' E) B`, are bisimilar if, when run in `Renv`-related initial states, they produce computations that are equivalent up to `Tau` and both terminate in states related by `state_invariant TT`. The relation `Renv` formalizes the assumption that the Imp environment and Asm memory have the same contents when viewed as maps from Imp variables / Asm addresses to values, and it is implied by `state_invariant`. Here, `TT` is the trivial relation on the output label of Asm, since a statement has a unique exit point; in general the `RAB` relation parameter in `state_invariant` is used to ensure that both computations jump to the same label, which is needed to prove that loops preserve the state invariant.

Since the compiler introduces temporary variables, the bisimulation does not hold over the uninterpreted ITrees. To prove that expressions are compiled correctly, we need to explain how reads and writes of Asm registers relate to the computations done at the Imp level. The relation `sim_rel` establishes the needed invariants, which ensure that the code generated by the compiler (1) doesn't corrupt the Asm memory, (2) uses registers in a "stack discipline," and (3) computes the Imp intermediate result `v` into the target register `n`. These properties are used to prove the correctness of `compile_expr`.

Crucially, despite correctness being termination sensitive, the proofs follow by structural induction on the Imp terms: all coinductive reasoning is hidden in the library. As in the first phase, the reasoning here follows by rewriting, this time using the `bisimilarity` relation and equations about `interp_imp` and `interp_asm` that are induced by virtue of being compositionally defined from `interp_state`.

Setting aside the usual design of the simulation relation, the resulting proofs are slightly verbose, but extremely elementary. They mostly consist in successive rewrites to commute the denotation with the various combinators, and some elementary semantic reasoning where events are reached to prove that the simulation relation is preserved. We believe that these kind of equational proofs can be automated to a large degree, a perspective we would like to explore in further works.

```
(* Relate an Imp env to an Asm memory *)
Definition Renv (g_imp : Imp.env) (g_asm : Asm.memory) : Prop :=
  ∀ k v, alist_In k g_imp v ↔ alist_In k g_asm v.

Definition sim_rel l_asm n: (Imp.env * value) → (Asm.memory * (Asm.registers * unit))
     → Prop :=
    fun '(g_imp', v) '(g_asm', (l_asm', _)) ⇒
      Renv g_imp' g_asm' ∧            (* we don't corrupt any of the Imp state *)
      alist_In n l_asm' v ∧           (* we get the right value in register n *)
      (∀ m, m < n → ∀ v,   alist_In m l_asm v ↔ alist_In m l_asm' v).
                                      (* we don't mess with anything on the "stack" *)

Context {A B : Type}.              (* Imp / Asm intermediate result types *)
Context (RAB : A → B → Prop).  (* Parameter that relates intermediate results *)

(* Relate Imp to Asm intermediate states. *)
Definition state_invariant (a : Imp.env * A) (b : Asm.memory * (Asm.registers * B))
    :=
    Renv (fst a) (fst b) ∧ (RAB (snd a) (snd (snd b))).

Definition bisimilar {E} (t1 : itree (ImpState +' E) A)
                            (t2 : itree (Reg +' Memory +' E) B)  :=
    ∀ g_asm g_imp l, Renv g_imp g_asm
    → eutt (state_invariant RAB) (interp_imp t1 g_imp) (interp_asm t2 g_asm l).

(* Imp / Asm program equivalence *)
Definition TT : unit → fin 1 → Prop := fun _ _ ⇒ True.
Definition equivalent (s:stmt) (t:asm 1 1) : Prop
  := bisimilar TT (denote_stmt s) (den_asm t f1).
```

FIGURE 3.6. Simulation relations for the compiler correctness proof

CHAPTER 4

# Application II: Verified Transactional Objects

## 4.1. Introduction

From multiprocessors to distributed systems, concurrency plays a key role in modern computing. Concurrent algorithms exhibit complex behaviors and are prone to subtle bugs, thus making concurrency a valuable target for formal verification. The literature on verifying concurrent systems has explored a diversity of consistency models and verification techniques.

**4.1.1. Consistency Models.** Concurrent systems consist of individual agents—which may be threads, processes, machines, etc.—communicating with each other. Through their interactions, agents gain their own partial views of the world, which may correlate with each other more or less strongly, for example depending on whether communication channels may delay, reorder, lose, or duplicate messages or memory instructions. A consistency model is a description of these modalities of communication. Two standard and closely related consistency models are sequential consistency and linearizability.

Sequential consistency [Lamport, 1979] requires that operations appear to execute atomically in some global order, called a *linearization*, consistent with the order observed locally by each process. Linearizability [Herlihy and Moss, 1993] is a stronger model, augmenting sequential consistency with the constraint that the global ordering of non-overlapping operations be also preserved by the linearization. The connections between sequential consistency and linearizability run deep [Attiya et al., 2017, Filipovic et al., 2009]. To give a concrete example distinguishing these two conditions, consider an execution of three processes concurrently accessing a shared reference, illustrated in Figure 4.1. Processes invoke methods to read and write the reference; these methods are represented here by the interval of time between their invocation and their completion. Processes $A$ and $B$ write values 1 and 2 to the reference, while process $C$ reads from it. Under sequential consistency, any permutation of those operations is a valid linearization. Process $C$ may thus read either 1 or 2, or the initial value 0 if it is scheduled before $A$ and $B$. Under linearizability, process $B$ must appear to be scheduled after process $C$ in order to preserve the ordering of their non-overlapping operations, thus $C$ is guaranteed not to read 2.

On the one hand, since weaker consistency models impose fewer synchronization constraints, they generally enable more performant implementations. Relaxed consistency models, which are weaker than sequential consistency, are also of notable interest as they accurately describe the behavior of common hardware architectures [Steinke and Nutt, 2004]. On the other hand, weaker consistency guarantees put more burden on the correct and effective usage of data structures in higher-level applications.

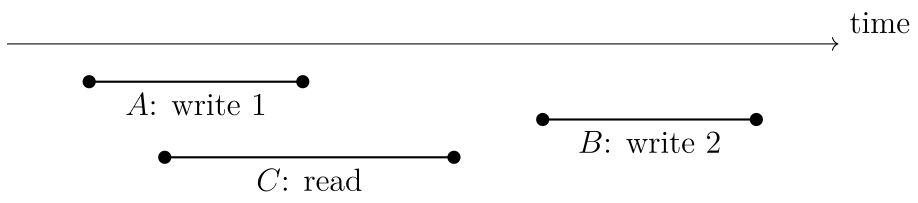


FIGURE 4.1. Concurrent history of a shared mutable reference

One benefit of linearizability over weaker consistency models is compositionality: complex linearizable objects can be constructed from smaller linearizable objects. Thus, linearizability aligns well with the themes of this dissertation. This chapter presents a framework of verified linearizable objects leveraging the compositional aspects of interaction trees.

**4.1.2. Verification for Concurrency.** Among the many approaches to formally verifying concurrent data structures, one principal distinguishing factor is the degree of automation. Model checking [Clarke et al., 2018, 1994, Černý et al., 2010] is a common basis for automated solutions. Implementations are expressed as finite state machines and specifications as first-order logical formulas, so that the resulting verification problem can be handled by standard model checking techniques and tools. A general limitation of model checking is the need to restrict the search space, to make the problem tractable at the cost of soundness, *e.g.*, by bounding the length of executions or the number of threads considered [Burckhardt et al., 2010]. For that reason, automated verification techniques are most often applied to validate high-level models of concurrent algorithms, before developing their final implementations separately.

In contrast, this work targets the practice of interactive theorem proving. The general approach is to design semantic models or program logics to be implemented in a proof assistant. What is lost in automation is gained in expressiveness. High-level specifications can be written in an expressive, higher-order language, and we can construct formal proofs that apply to all possible executions of a given program. In this context of interactive theorem proving for concurrency, a central topic is separation logic [Reynolds, 2002]. It is an extension of Hoare logic, viewing imperative programs in terms of assertions on state. Separation logic introduces a *separating conjunction*, joining together assertions on disjoint parts of the program, which is key to the ubiquitous problem of *aliasing*. Separation logic is a foundation for many more concepts for dealing with concurrency: atomic triples [Rocha Pinto et al., 2014], ghost state [Jung et al., 2016b], prophecy variables [Jung et al., 2019], etc. VST [Appel, 2014] and Iris [Jung et al., 2015] are two notable implementations of concurrent separation logic in Coq. Separation logic is an *axiomatic* approach to program verification, investigating the structure of logical predicates on state. This chapter presents a more *algebraic* approach, viewing linearizable objects as first-class entities. Our main case study leverages this aspect to verify a transactionally predicated map object, built as a nontrivial compositon of individual linearizable (and serializable) objects.

**4.1.3. From Linearizability to Serializability.** Two styles of concurrency have been studied intensively from a formal-methods perspective. On the one hand, we have classic data structures (e.g., stacks, queues, dictionaries) that rely on primitives like locks and compare-and-set instructions to guarantee atomicity of their methods; we might call these *single-method-atomic* data structures. On the other hand, there is a separate tradition of transaction-based APIs, which guarantee atomicity of chains of method calls; let us call them *multi-method-atomic*. In broad strokes, single-method atomicity is appealing for its superior performance, while multi-method atomicity is appealing for its simpler programming model.

Sometimes it is useful to break a complex concurrent application into pieces written in *both* of these styles [Spiegelman et al., 2016, Assa et al., 2020, Elizarov et al., 2019], but no one had previously shown how to prove functional correctness of such applications. We remedy this gap with a new framework for modular proofs of programs that mix the two styles.

As a motivating case study, we focus on *transactional predication* [Bronson et al., 2010]. The idea behind data structures implemented in this style—typically finite sets or maps—is to combine software transactional memory (STM) [Shavit and Touitou, 1995, Harris et al., 2005], which provides good compositionality and reasoning properties but is relatively inefficient, with a concurrent data structure that exhibits good performance but provides a noncomposable interface. Instead of storing the entire data structure in transactional memory, we store *predicates* about the data structure—Boolean-valued mutable references indicating the membership of particular elements in the set. Updates to the predicates reflect actual changes to the data structure, while a nontransactional concurrent object manages the mapping between keys and predicates; this split reduces conflict detection to the STM's detection of write-write and read-write collisions. The challenge is finding a framework that supports ergonomic proofs involving both kinds of mechanisms.

A growing community of "programmer-provers" have learned to be effective at proving properties of code written in the native programming languages of various proof assistants. These native languages tend to be purely functional, but frameworks like interaction trees have demonstrated how to extend the lightweight combination of programming and proof that arises naturally for pure functional programs to situations involving a range of computational effects. Unfortunately, when we include concurrency the complexity of reasoning increases dramatically.

To add concurrency on top of a proof assistant's native functional language and proof tools, we organize code into layered collections of concurrent objects whose specifications force all methods to behave atomically; in this setting we support a variety of concurrency styles, principally those associated with traditional linearizable concurrent data structures and serializable transactional memory. To support transactions, we introduce a novel technique of *passing code reified as interaction trees* and applying instrumentation functions to transform those trees.

For example, consider a concurrent stack object with methods *push* and *pop*. If such an object has been proven *linearizable* against a natural specification, clients can treat calls to *push* and *pop* as atomic, ignoring their actual implementations, which might use tricky fine-grained primitives like compare-and-swap. However, one level up, clients *do* still need to reason about the ways that sequences of calls to *push* and *pop* can nondeterministically interleave. For instance, if one wanted to transfer the elements of one concurrent stack to another, a naive implementation may be the following `moveStack` function:

$$\begin{aligned} \mathit{moveStack}(\mathit{from}, \mathit{to}) := {} & \mathit{vo} \leftarrow \mathit{from.pop}(); \\ & \mathsf{match}\ \ \mathit{vo}\ \ \mathsf{with}\ \mathsf{None} \Rightarrow \mathsf{return}\ () \\ & \qquad\qquad\qquad \mid \mathsf{Some}(v)\ \ \Rightarrow \mathit{moveStack}(\mathit{from}, \mathit{to});\ \ \mathit{to.push}(v) \end{aligned}$$

Multiple calls to `moveStack` from different client threads on the same `from` stacks may interleave their invocations of `push` and `pop`, resulting in two subsets of the initial stack being sent to different destinations. In particular, if the desired behavior of `moveStack` were to transfer the elements from one stack to another *atomically*, it would be necessary to employ a more sophisticated implementation involving synchronization mechanisms external to the stack.

The complexity of reasoning about concurrent objects is one reason why *transactions* [Papadimitriou, 1979, Herlihy and Moss, 1993, Shavit and Touitou, 1995, Harris et al., 2005] have become a popular concurrency abstraction. Transactions allow the programmer to declare that a particular block of code must be run atomically, leaving it to the compiler and/or runtime protocol to figure out how to provide this atomicity both soundly and efficiently.

Software transactional memory is probably the best-known realization of this idea in the functional-programming world. It allows programmers to write code like

$$\mathit{moveStackAtomically}(\mathit{from}, \mathit{to})\ \ :=\ \ \mathit{asTransaction}(\lambda\_.\ \mathit{moveStack}(\mathit{from}, \mathit{to}))$$

where the call to *moveStack* is wrapped in a thunk and passed to a library procedure that promises to run such thunks atomically.

Our goal is to blend transactions and classical concurrent objects into a unified concurrent-object framework. In particular, we want to capture transaction protocols as objects that accept user transactions—thunks that chain calls together—and execute them atomically. Further, we want to express the correctness of such objects in terms of linearizability, so that we can compose our verified transactional objects with other verified linearizable objects.

Our key technical advance is making the structure of concurrent programs more syntactic by using interaction trees (Chapter 2) and executing them later with an explicit interpreter. The previous chapters have demonstrated interaction trees for representing sequential programs, and I will now show how to adapt them into a framework for linearizability proofs of concurrent objects. Interestingly, in this setting, serializability (the classical transaction correctness condition [Papadimitriou, 1979]) turns out to be literally a special case of linearizability for objects whose higher-order methods take code (represented as interaction trees) as arguments. Note the contrast with familiar correctness criteria for transactions, which are typically stated as ad-hoc conditions [Papadimitriou, 1979, Guerraoui and Kapalka, 2008, Doherty et al., 2013, Scott, 2006, Jagannathan et al., 2005].

To make this framework more efficient, our library methods perform *syntactic transformations* on their interaction-tree arguments. For example, consider how *asTransaction* might transform the tree of method calls *moveStack* wishes to perform. After inlining a bit of library code, we first see insertion of code for initialization (*beginTransaction*) and finalization (*commitTransaction*, etc.).

```
$moveStackAtomically(from, to) :=$
    $\_ \leftarrow \mathsf{beginTransaction};$
    $r \leftarrow moveStack'(from, to);$
    $\mathsf{match}\ r\ \mathsf{with\ None} \Rightarrow \mathsf{abortTransaction}$
                 $\mid \mathsf{Some}(r) \Rightarrow \_ \leftarrow \mathsf{commitTransaction};\ \mathsf{return}(r)$
```

The library builds an instrumented routine *moveStack*′, which looks just like *moveStack* but with calls to instrumented methods like *from.pop*′() instead of *from.pop*(), where *from.pop*′() is a modified version of *from.pop*() whose *concurrent read* and *write* methods have been replaced with *transactional trans_read* and *trans_write* methods, and any sequence of transactional methods between a beginTransaction call and a commitTransaction call shall appear to execute atomically as a whole. Concurrent transactions may conflict with each other, in which case subsequent transactional method calls may no longer return consistent results. Instead, those methods fail by returning a None value, and all the previous operations of the ongoing transaction are cancelled.

```
$pop() := ...;$    (* Original code: *)        $pop'() := ...;$    (* Instrumented code: *)
    $v \leftarrow read(i);\ k(v)$                          $vo \leftarrow trans\_read(i);$
                                                  $\mathsf{match}\ vo\ \mathsf{with\ None} \Rightarrow \mathsf{return(None)}$
                                                                 $\mid \mathsf{Some}(v) \Rightarrow k'(v)$
```

Crucially, the free monad represents programs explicitly as trees of method calls and responses to their return values, allowing us to traverse these trees syntactically and add uniform instrumentation.

Recapping, we adopt linearizable objects as the foundation of our framework. An object packages concurrent code with private state to implement public methods, and it is verified with respect to a sequential specification. The framework supports modular implementation and verification of concurrent libraries that include both classic linearizable data structures and serializable transactional objects. It formulates serializability in terms of linearizability and offers a unified proof technique where all library modules are proved linearizable against straightforward sequential specifications.

The modular nature of the framework fits our goal of verifying transactional predication. The basic idea, presented in more detail in the next section, is to combine a concurrent map with a transactional-memory library, yielding a higher-level *transactional concurrent map* abstraction. While the map we build on provides atomicity only at the level of single-key reads and writes, the higher-level map allows grouping of several dependent reads and writes in transactions. The underlying concurrent map is used to associate keys with references to mutable memory cells managed by the transactional-memory system. As a result, key lookups have the performance of the concurrent map, while the values associated with several keys can be read or written within a single atomic transaction. Each of these main ingredients is itself constructed atop more primitive library modules with their own sequential specifications and is separately verified.

In summary, our key contributions are:

- We present a core formal framework for *verified linearizable objects* whose methods are expressed as interaction trees (Chapter 2). The framework includes powerful composition combinators to define implementations and specifications and verify implementations (Section 4.2 to Section 4.4), modularly. A key component of the framework is a unified proof principle for linearizability (Section 4.5), which we demonstrate by verifying a concurrent hash-map object (Section 4.5.1) and a concurrent histogram (Section 4.5.2).
- Within this framework, we introduce *verified transactional objects*. The key idea is to view transactions as first-class entities, represented as interaction trees, and use *interaction-tree rewriting* to instrument transactions with the bookkeeping calls required to ensure atomic execution. We use the composition operations defined earlier to state transaction serializability as an instance of linearizability. We then apply the proof principle for linearizability to prove the serializability of a transactional-memory object based on transactional mutex locks (TML) [Dalessandro et al., 2010] (Section 4.6).
- We use this framework to carry out a significant case study, leveraging its support for compositional reasoning to verify a concurrent-set object implemented using *transactional predication* (Section 4.7). To our knowledge, this correctness proof is the first rigorous one—mechanized or otherwise—for a concurrent object that encapsulates both a conventional concurrent data structure and a transactional-memory implementation.
- We package our verification framework as a *reusable Coq library*, proved correct from first principles. The Coq library[7] mechanizes everything presented in the thesis, including the main case study and all its dependencies.

The work presented in this chapter, originally published in Lesani et al. [2022], was started independently by my coauthors prior to my work on interaction trees. Significant portions of this chapter present work that was completed prior to my involvement, namely: the verification principle for linearizability (Section 4.5), its application to verifying hash-map and histogram objects, and the verification of the transactional mutex locking protocol (Section 4.6.1). They are preserved in this dissertation for they help to illustrate the scope of this project. However, I was actively engaged in writing the expository sections (Sections 4.2 to 4.4 and 4.6), requiring a non-trivial effort in untangling the concepts relevant to the framework for a clear presentation.

It is notable that the coinductive type of interaction trees and associated key definitions such as the `interp` combinator appeared early on this project. Nonetheless, the originality of the ITree library is rather to be found in the surrounding equational theory up to weak bisimulation, and I've made use of it in the verification of transactional predication (Section 4.7), which is my main technical contribution.

[7]Source available at https://zenodo.org/record/6342476

```
inc := i ← r.read();
       ok ← r.cas(i, i + 1);
       if ok then i + 1 else inc
```

FIGURE 4.2. *inc* method

## 4.2. Overview

We begin by fixing terminology and notation for some standard concepts, using the example of a simple concurrent counter; then we review the core idea of transactional predication.

### 4.2.1. Objects.

We speak of *objects* encapsulating some *state* which is accessed through *methods* grouped into *interfaces*. An object implements a *high-level interface* by issuing a sequence of calls to a *low-level interface*. A core assumption, which significantly simplifies our composition laws, is that object methods may not spawn new threads—i.e., concurrency arises at the level of applications, not internally in libraries.

Consider an object with just one method, *inc*, which increments an abstract counter and returns its new value. We can implement this object in terms of a compare-and-swap (CAS) register interface, with methods *read*, *write*, and *cas* (Figure 4.2). Multiple application threads may call *inc* simultaneously. For example, Figure 4.3(a) shows a possible execution history starting with two calls to *inc* in two concurrent threads, displaying interactions through the counter's high- and low-level interfaces. Time flows vertically. Horizontal arrows are *events*—either method calls (rightward arrows) or returns (leftward arrows). Calls from the external environment to the counter object are on the left; calls from the counter to the underlying CAS register object are in the middle of the diagram. Events in thread 1 are shown in red (and normal font); events of thread 2 are blue (and italic). In this execution, both threads first execute *read* and get the same value back. Both then try to compare-and-swap, but only one of them (thread 2) succeeds, incrementing the counter once. Thread 2 returns the value of the counter, now at 2. Thread 1 tries again and finally succeeds, incrementing the counter to 3. The counter object translates single calls from its own clients (on the left) into multiple calls to its low-level interface (on the right).

### 4.2.2. Objects and Linearizability.

A *sequential specification* describes the high-level behavior of an object when its methods are executed sequentially, waiting for each to return before calling the next one. Formally, a sequential specification for a given interface is defined as a labeled state-transition system, where the labels are pairs of method calls and return values. Alternatively, a sequential specification can itself be viewed as an idealized "atomic object," whose methods execute fully and return their results as soon as they are called. We say that an object is *linearizable* [Herlihy and Wing, 1990] with respect to a sequential specification when every method invoked on the object appears to execute atomically at some point between its call and its return, matching the behavior of the method call from the sequential specification. (We will formally define linearizability and its properties in Section 4.4.) For example, Figure 4.3(b) shows how the high-level history of the counter object in Figure 4.3(a)

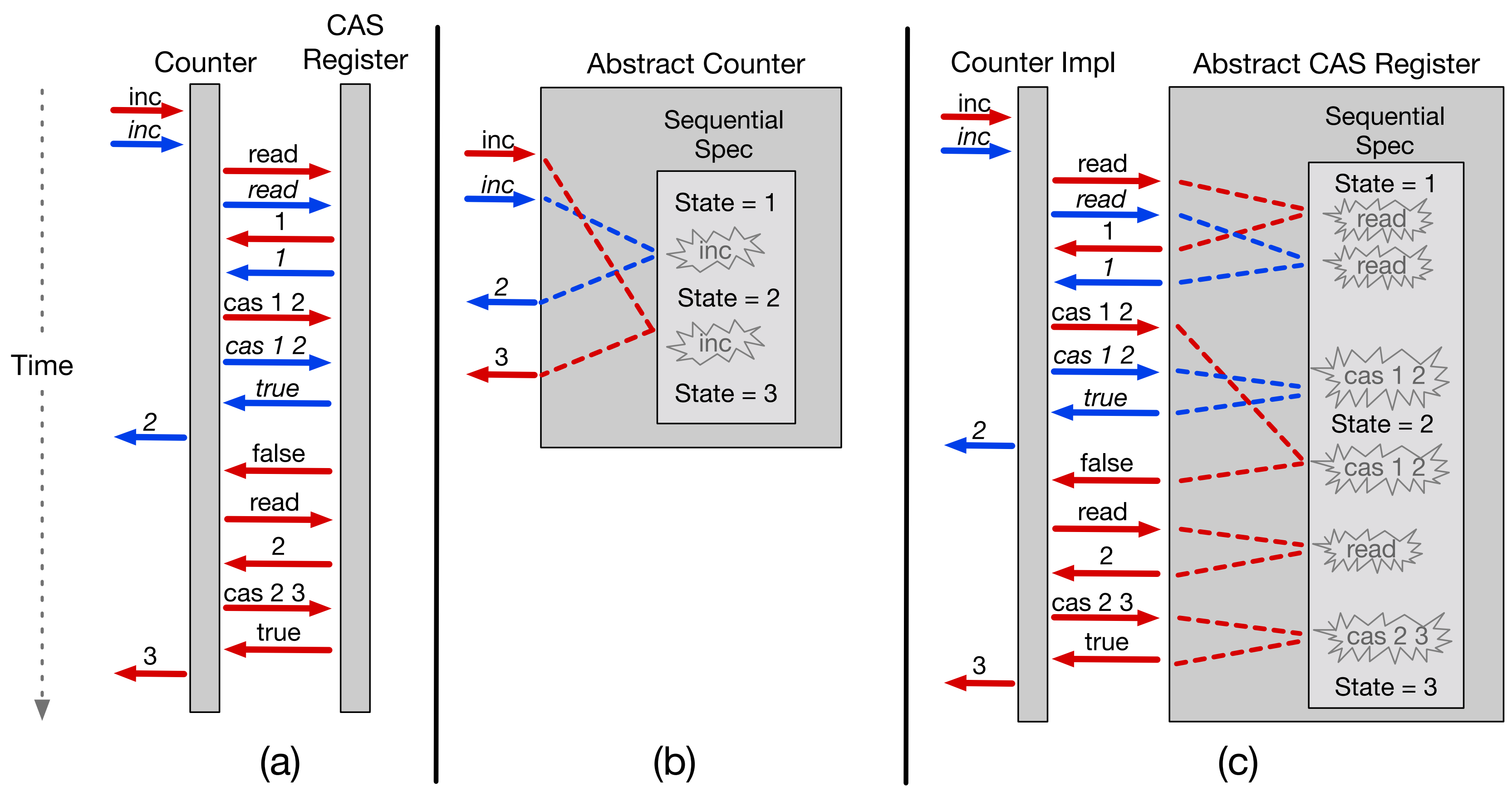


FIGURE 4.3. Execution of a linearizable counter object

(a) Possible execution history of a counter object implemented using a CAS register object.
(b) Linearization of the high-level history, observed externally by callers of the counter object.
(c) Linearization of the low-level history, observed internally by the counter object when calling the CAS register object.

can also be produced by a sequence of atomic interactions with an idealized counter shown in lighter gray: the idealized counter responds first to thread 2 and then to thread 1.

So far, we have focused on one side of interfaces, where an object acts as the *callee* of its high-level interface (here, *inc*); the object must satisfy the high-level sequential specification that is associated with the interface. The object also acts as the *caller* of its low-level interface (here, *read* and *cas*), which is itself associated with a low-level sequential specification that the object can rely on in order to satisfy its high-level specification. In the example, our counter object interacts, through its low-level interface, with a CAS-register object that it assumes is linearizable with respect to its own sequential specification. Figure 4.3(c) shows a linearization of the low-level history from Figure 4.3(a).

**4.2.3. Transactions and Serializability.** An appealing feature of linearizable objects is that, in every layer of a hierarchical system and its proofs, every object is proved against a straightforward sequential specification of the objects it depends on. In the example, when proving the linearizability of the counter object, we can restrict attention to histories where low-level method calls return immediately, allowing us to reason about many fewer interleavings than if we had to consider the actual implementations of *read* and *cas*. However, while the methods of a linearizable object are logically atomic, clients must still reason about possible interleavings of *sequences* of calls.

A more user-friendly model of concurrency is offered by transactional memory, which allows client programmers to choose the granularity of atomic actions, delimiting blocks of client code—possibly containing multiple calls—that must execute atomically as wholes, a requirement called serializability. We will model transactional memory's correctness as a special case of linearizability: atomic transactions can be viewed as calls to the methods of a linearizable object, where the arguments to those methods consist of *programs* to be interpreted (atomically). A program is an interaction tree that may include method calls on an interface. One familiar way to implement such an object is to interpret transactions in an environment where conflicts can be detected and transactions can be rolled back. We will formalize this approach (in Section 4.6) as a form of "transaction instrumentation," inserting "transaction life-cycle" method calls.

**4.2.4. Transactional Objects.** The fine-grained conflict detection of transactional-memory protocols can hinder performance. Implementations of high-performance transactional objects [Bronson et al., 2010, Spiegelman et al., 2016, Assa et al., 2020, Elizarov et al., 2019] use TM (transactional memory) only sparingly, to reduce the frequency of "false conflicts," instead delegating most memory accesses to more efficient concurrent data structures. They achieve the best of both worlds: composability from TM and performance from concurrent data structures.

An elegant realization of this idea is the technique called *transactional predication* [Bronson et al., 2010]. Figure 4.4 shows the internal structure of a transactional-set object built in this style. The transactional set is vertically composed on top of the horizontal composition of two lower-level objects: a *locator* (which wraps a concurrent

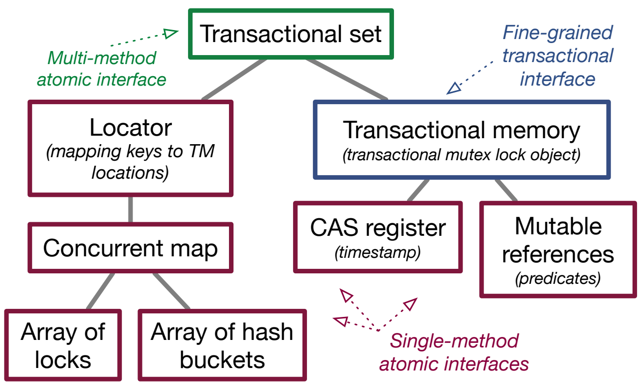

Transactional set
Multi-method atomic interface
Fine-grained transactional interface
Locator
(mapping keys to TM locations)
Transactional memory
(transactional mutex lock object)
Concurrent map
CAS register
(timestamp)
Mutable references
(predicates)
Array of locks
Array of hash buckets
Single-method atomic interfaces

FIGURE 4.4. Architecture of transactional predication

map) and a TM. The locator maps each element that has ever been in the set (whether or not it is still in the set at the moment) to a *location* (a mutable cell) managed by the TM. A location stores a mutable Boolean (called a *predicate*, hence the technique's name) that represents whether the element is currently in the set. The locations themselves are managed by the TM. A locator is a simple concurrent object that is built on top of a concurrent-map object. (We will later implement a concurrent-map object on top of arrays of locks and buckets and implement the TM object on top of a register and a map object.) Given an element, the locator checks whether the element is already in the map. If the element is present, the locator returns the location that the element is mapped to. If it is not, the locator puts the element and a fresh TM location in the map.

The interface of a transactional set accepts user programs on the set interface and executes them atomically. Given a user program, it inserts TM life-cycle calls such as TM initialization and commitment calls into the program. For each set method call on an element in the user program, the locator is called to find the location corresponding to that element. A TM read or write method on the Boolean value stored in that location is performed depending on the set method call (membership, insert, or delete). A single transaction may access several elements of the set, leading to accesses to several locations in the TM. Since the TM enforces atomicity of all such accesses to the locations, the transactional-set object inherits the same atomicity. Accesses to these locations track conflicts only at the semantic level for the set interface. Contending accesses inside the locator do not lead to conflicts. By contrast, a set implemented purely based on TM tracks conflicts on the low-level reads and writes and aborts more transactions.

**4.2.5. Verified Transactional Objects.** The use of a concurrent object (the locator) together with a TM raises significant challenges for verification. The TM guarantees the atomicity of transactions that use just its own interface, but the methods of the transactional set call methods on *both* the locator and the TM. How can the atomicity guarantees of the TM be used to prove the atomicity guarantees of the transactional set? An important observation is that the locator object behaves like a *pure function*, as far as its callers can tell: although the mapping from the keys

```
Definition Interface := Type → Type.
```

(A) Type of interfaces

```
Inductive Map {K V : Type} : Interface :=
| Get : K → Map V
| Put : K → V → Map unit.
```

(B) Map interface

Figure 4.5. Definition of interfaces in Coq

to the locations is actually decided dynamically, once a mapping is made, it stays unchanged. We prove simulation relations that let us substitute method calls on the locator with ordinary function calls in the metalanguage. This substitution reduces method calls on the transactional set to transactions on the TM interface, allowing us to apply the atomicity guarantees of the TM directly. (See Section 4.7 for more detail.)

This proof style assigns each component a natural specification, without anticipating how other parts of the hierarchy will work. For instance, the specifications of classical concurrent data structures need say nothing about transactions. Also, the approach is modular: each library component can be proved separately against its natural specification.

This transactional set can be composed on top of any concurrent map and transactional memory implementing the given specifications. As concrete examples, this dissertation also presents concrete implementations of those specifications to illustrate the core concepts of our framework. We show a concurrent map using lock striping, which uses an array of locks to protect an array of buckets; and a transactional mutex lock (TML) object, which uses a compare-and-swap register to increment timestamps that control concurrent accesses to mutable references.

Now we are ready to dive into the details. The next section formalizes the ideas of objects, interfaces, and sequential specifications. Section 4.4 and Section 4.5 formalize linearizability, together with the related concepts of simulation and composition, and Section 4.5.1 and Section 4.5.2 apply them to verify concurrent data structures. Section 4.6 and Section 4.7 formalize transactions and transactional predication.

## 4.3. Concurrent Objects

**Interfaces.** An *interface* $M$ is a collection of *method calls*. A method call, for example $\mathsf{lookup}(k)$, intuitively consists of a method name ($\mathsf{lookup}$) paired with its arguments ($k$)—*e.g.*, $\mathsf{lookup}(1)$ and $\mathsf{lookup}(2)$ are different method calls. The arguments to methods typically consist of first-order values (integers, strings), but they may also be *programs* (described below); indeed, this possibility will be key to our treatment of transactions in Section 4.6.

**Definition 4.3.1.** *An* interface *is a type constructor* $M : \mathrm{Type} \to \mathrm{Type}$. *The type* $M\ R$ *is the type of methods with return type* $R$ *(Figure 4.5a).*

```
Definition Impl (M N : Interface) := (∀ (R : Type), M R → Prog N R).
Record Spec (M : Interface) :=
  { State : Type
  ; Init : State
  ; Transitions : State → ∀ (R : Type), M R → State → R → Prop }.
Record Object (M : Interface) :=
  { LowM : Type → Type
  ; ObjImpl : Impl M LowM
  ; ObjLowSpec : Spec LowM }.
```

FIGURE 4.6. Implementations, sequential specifications, and objects in Coq

For instance, the map interface map associating keys of type $K$ with values of type $V$ can be defined as follows (Figure 4.5b): The type constructor map has two data constructors get and put. For any key $k$, there is a method call $\mathsf{get}(k) : \mathsf{map}\,(\mathsf{option}\ V)$, where the result type is $\mathsf{option}\ V$. Similarly, for any key $k$ and value $v$, there is a method call $\mathsf{put}(k, v) : \mathsf{map\ unit}$.

**Programs.** *Programs* are data structures that describe chains, or more generally trees, of method calls. They play two roles in our framework: as the bodies of methods associated with objects and as the bodies of transactions. We represent them as interaction trees.

This encoding of programs is important for the flexibility of our framework. First, interaction trees are potentially infinite, as required, for example, to implement an algorithm that retries some action until it succeeds, like the counter in Section 4.2. Second, programs are *first-class, syntactic* objects that can be manipulated by other programs, which allows encoding a variety of useful code transformations, including what we need to implement transactional memory cleanly (Section 4.6).

**Definition 4.3.2.** *An* implementation *of a high-level interface $M$ in terms of a low-level interface $N$ is defined as a* handler *(Section 2.2):*

$$\mathsf{Impl}\, M\, N = (M \rightsquigarrow \mathtt{itree}\ N)$$

*i.e., implementations are polymorphic functions $\forall R, M\ R \to \mathtt{itree}\ N\ R$, ensuring that a high-level method $\mathsf{m}$ is mapped to a program impl$(\mathsf{m})$ over the low-level interface with the same result type $R$.*

For example, Figure 4.7 shows an implementation of a counter interface using a CAS register interface, which encodes in Coq the pseudocode from Figure 4.2.

The identity implementation, $\mathsf{id} : \mathsf{Impl}\, M\, M$, maps any method $\mathsf{m}$ to the program $(x \leftarrow \mathsf{m}; x)$, which calls the same method and returns its result. It will serve as the identity of vertical composition (in Section 4.4).

To define the semantics of an implementation, we must first define the semantics of its low-level interface by specifying the values its methods may return when called sequentially.

**Definition 4.3.3.** *A* sequential specification *spec of an interface $M$, written spec* : $\mathsf{Spec}\, M$*, is a labeled state-transition system: it is a tuple $(S, s_0, \Delta)$, comprising a type $S$*

```
Inductive Counter : Interface :=
| Incr : Counter nat.

Inductive Cas : Interface :=
| Read : Cas nat
| CompareAndSwap : nat → nat → Cas bool.

CoFixpoint counter_impl : Impl Counter Cas := fun R (m : Counter R) ⇒
  match m with
  | Incr ⇒ n ← trigger Read ;;
            ok ← trigger (CompareAndSwap n (n+1)) ;;
            if ok then ret i+1 else counter_Impl _ Incr
  end.
```

FIGURE 4.7. Implementation of counter interface using CAS register interface

*of abstract states, an initial state $s_0$ of type $S$, and a relation $\Delta : \forall R, \mathcal{P}(M\ R \times S \times R \times S)$ between a method call, a current state, a result, and a next state, where $\mathcal{P}(U)$ denotes the powerset of $U$. (The result type $R$ is another parameter of that relation, which we leave implicit in the examples in the rest of this chapter.)*

In practice, interfaces are designed with a particular sequential specification in mind, so one might consider merging the two notions as one. Interfaces are a kind of *shallow* specifications, only indicating the types of inputs and outputs, whereas our sequential specifications are a kind of *deep* specifications, describing the desired relations between inputs and outputs. However, keeping these notions separate lets us structure the process of verification in a fine-grained way. Implementations are raw, unverified code, under a lightweight well-typedness constraint depending only on interfaces. Objects (defined later in this section) pair up implementations with a specification of their low-level interface, this induces some observable behavior through the high-level interface. Verifying an object is to prove some relation between the object's observable behavior and a sequential specification of its high-level interface, such as linearizability or serializability.

As an example, consider a sequential specification of locks, call it lock-spec. Its transition system, shown in Figure 4.8a, has a state of type bool that represents whether the lock is acquired, with initial state false. The two methods of the interface are lock and unlock (with result type unit): the semantics of the former is to toggle the state to true, and the latter to set it to false. This simple specification indeed captures the behavior of a lock: if more than one thread simultaneously try to lock, only one will be allowed to take the transition from the initial false to true, and other threads will have to wait for it to unlock the state back to false to have another chance at taking the lock. As another example, the sequential specification of a register, reg-spec, is a tuple of the type $\mathbb{N}$ (numbers), the initial value 0, and the obvious transition system (Figure 4.8b) for the methods read, write, and cas. The method cas takes two arguments, changes the value of the register to the second argument if its current value is equal to the first argument, and returns a Boolean indicating whether

$$\Delta_{\textsf{lock-spec}} = \{(\textsf{lock}, \textsf{false}, (), \textsf{true}), (\textsf{unlock}, \textsf{true}, (), \textsf{false})\}$$

(A) Lock specification

$$\begin{aligned}\Delta_{\textsf{reg-spec}} &= \{(\textsf{read}, n, n, n) \mid n \in \mathbb{N}\}\\ &\cup \{(\textsf{write}(n), m, (), n) \mid n, m \in \mathbb{N}\}\\ &\cup \{(\textsf{cas}(n, m), n, true, m) \mid n, m \in \mathbb{N}\}\\ &\cup \{(\textsf{cas}(n, m), p, false, p) \mid n, m, p \in \mathbb{N}, n \neq p\}\end{aligned}$$

(B) Register specification with compare-and-swap

$$\begin{aligned}\Delta_{\textsf{map-spec}} &= \{(\textsf{get}(k), f, f(k), f) \mid f \in K \to \textsf{option}\, V\}\\ &\cup \{(\textsf{put}(k, v), f, (), f[k \mapsto v]) \mid f \in K \to \textsf{option}\, V, k \in K, v \in V\}\end{aligned}$$

(C) Map specification

FIGURE 4.8. Examples of sequential specifications

$$\textsc{CallStep}\quad \frac{(\textsf{m}, s, r, s') \in \Delta}{(s \mid x \leftarrow \textsf{m}; p(x)) \rightarrow_{spec} (s' \mid p(r))} \qquad \textsc{TauStep}\quad \frac{}{(s \mid \tau; p) \rightarrow_{spec} (s \mid p)}$$

FIGURE 4.9. Program transition rules

it succeeded. Finally, a sequential specification for maps, $\textsf{map-spec}$, can be defined as follows. The state is a mapping from keys $K$ to values $\textsf{option}\, V$, with an initial state that maps any keys to the "none" value $\perp$. The interface is given by two methods (with the expected transitions, Figure 4.8c): $\textsf{get}(k)$ with the return type $\textsf{option}\, V$ for lookup and $\textsf{put}(k, v)$ with the return type $\textsf{unit}$ for insertion.

A sequential specification of an interface naturally provides a stateful interpretation of programs over the same interface, represented as a *transition relation* $\rightarrow_{spec}$, defined in Figure 4.9, between pairs $(s \mid p)$ of abstract states and programs, where finished programs do not step.

**Definition 4.3.4.** *An* object *obj with interface $M$, written obj* : $\textsf{Object}\, M$*, is a triple $(N, impl, spec)$ of low-level interface $N$, implementation impl* : $\textsf{Impl}\, M\, N$*, and sequential spec* : $\textsf{Spec}\, N$*.*

Multiple threads may make method calls on the high-level interface of an object; the corresponding programs will run concurrently, interleaving calls to the object's low-level interface. The semantics of the low-level interface is given by a sequential specification, and every low-level method call operates atomically on the shared low-level state according to that specification.

The observable behavior of an object is characterized by the methods that the clients may call concurrently and the responses that they receive. Formally, behavior is modeled by the trace semantics of the following state-transition system. Let

$$\textsc{Spawn}\quad \frac{\theta \notin \Theta}{(s \mid \Theta) \xrightarrow{\theta,\mathsf{m}}_{\mathit{obj}} (s \mid \Theta; \theta \mapsto \mathit{impl}(\mathsf{m}))} \qquad \textsc{Step}\quad \frac{(s \mid p) \to_{\mathit{spec}} (s' \mid p')}{(s \mid \Theta; \theta \mapsto p) \to_{\mathit{obj}} (s' \mid \Theta; \theta \mapsto p')}$$

$$\textsc{Return}\quad \frac{}{(s \mid \Theta; \theta \mapsto v) \xrightarrow{\theta,v}_{\mathit{obj}} (s \mid \Theta)}$$

FIGURE 4.10. Operational semantics of concurrent objects

$\mathit{obj} = (N, \mathit{impl}, \mathit{spec})$ be an object with high-level interface $M$ and low-level interface $N$. In a concurrent execution, the state of the object consists of a pair of (1) a shared *low-level state* $s$ of the sequential specification *spec* and (2) the states of all the threads, represented as a finite map $\Theta$ from *thread identifiers* $\theta$ (drawn from some infinite supply of names) to programs $p$ over the low-level interface $N$, each representing the remainder of a method that a thread has yet to finish executing. (We write $\Theta; \theta \mapsto p$ to denote a map $\Theta$ extended with the mapping from $\theta$ to $p$.) The initial state of the object is the pair $(s_0 \mid \emptyset)$, containing the initial state $s_0$ of *spec* and the empty thread state $\emptyset$.

The *operational semantics* of the object *obj*, shown in Figure 4.10, has three kinds of transitions, all written $\to_{\mathit{obj}}$, with different labels above the arrow.

(1) A new thread can be *spawned* to execute a high-level method call $\mathsf{m}$. Such a step is represented by a transition labeled with a fresh thread identifier $\theta$ (not already in $\Theta$) as well as the method (and arguments) $\mathsf{m}$. The thread state $\Theta$ is extended with a mapping from the new identifier $\theta$ to the program that the implementation *impl* associates to the method $\mathsf{m}$.
(2) A thread can step *internally*, based on the program transition relation defined above, and mutate the internal state. The environment cannot observe this transition, so it is unlabeled.
(3) A thread can *return* once it is done, *i.e.*, when the remaining program is a leaf $v$. This transition is labeled with the thread identifier (to identify the transition that initially spawned this thread) and its result $v$. The transition takes the thread out of the state. (This removal lets us model the possibility for one thread $\theta$ to perform multiple calls sequentially, waiting for each call to return before making the next call.)

A history (or trace) is a sequence of events $e$—either call events $(\theta, \mathsf{m})$ or return events $(\theta, v)$. The *observable behavior* $\mathsf{beh}(\mathit{obj})$ of an object is the set of histories produced by the above transition system ($\xrightarrow{e}_{\mathit{obj}}$) starting from the initial state, i.e., $\mathsf{beh}(\mathit{obj}) = \{(e_0 \dots e_n) \mid \exists\, s\, \Theta,\ (s_0 \mid \emptyset) \to^{\star}_{\mathit{obj}} \xrightarrow{e_0}_{\mathit{obj}} \cdots \to^{\star}_{\mathit{obj}} \xrightarrow{e_n}_{\mathit{obj}} (s \mid \Theta)\}$. Note that multiple low-level internal steps ($\to^{\star}_{\mathit{obj}}$, the transitive closure of $\to_{\mathit{obj}}$) may happen between two labeled steps, but they are not recorded in the history $(e_0 \dots e_n)$.

## 4.4. Linearizability and Composition

Having thus defined the semantics of individual objects, we now formalize the familiar notions of simulation, linearizability, and composition, their properties, and their relation with each other in a novel unified framework. We first define a notion of refinement between objects based on trace inclusion. This notion, in turn, will allow us formally to capture linearizability between an object and a specification. Then, we state properties of linearizability that support its hierarchical verification by successive refinements. Finally, we define composition patterns for objects and state properties of compositions that support modular verification of linearizability for composed objects.

**Simulation.** We define simulation relations for specifications and objects in three steps. We first define simulation between two specifications. Then, on top of that notion, we define simulation between an object and a specification when the object is executed sequentially. Finally, we define simulation between two objects when they execute concurrently.

Intuitively, a specification simulates another specification with the same interface iff (1) every value that can be returned by a method call on the former can also be returned by the same method call on the latter, and (2) this property continues to hold for the resulting post-states.

**Definition 4.4.1** (Specification simulation). *The* similarity relation *between two sequential specifications* $spec_1 \lesssim_{\mathbb{SP}} spec_2$*, where* $spec_1 = (S_1, s_1, \Delta_1)$ *and* $spec_2 = (S_2, s_2, \Delta_2)$ *are specifications of the same interface* $M$*, is the greatest simulation relation, i.e., a relation satisfying the following condition: for all* $\mathsf{m}$*,* $r$*, and* $s'_1$*, if* $(\mathsf{m}, s_1, r, s'_1) \in \Delta_1$*, then there exists* $s'_2$ *such that* $(\mathsf{m}, s_2, r, s'_2) \in \Delta_2$ *and* $(S_1, s'_1, \Delta_1)$ *simulates* $(S_2, s'_2, \Delta_2)$*.*

Any object *obj* can be interpreted in a sequential manner: the bodies of the methods are interpreted atomically as state transitions over the states of the low-level specification, by iterating the program transition relation $\rightarrow_{spec}$ until we reach a value. This convention associates *obj* to a sequential specification whose interface is the high-level interface of *obj*.

**Definition 4.4.2** (Interpreted sequential specification). *The* interpreted sequential specification *for the object* $obj = (N, impl, spec)$ *with respect to the low-level specification* $spec = (S, s_0, \Delta)$ *is the specification* $\textsf{interp-as-spec}(obj) = (S, s_0, \Delta')$ *where, for all* $\mathsf{m}$*, we have* $\Delta' \ni (\mathsf{m}, s, v, s')$ *iff* $(s \mid impl(\mathsf{m})) \rightarrow^*_{spec} (s' \mid v)$*.*

**Definition 4.4.3** (Sequential object simulation). *An object obj* sequentially simulates *a sequential specification spec, written* $obj \lesssim_{\mathbb{S}} spec$*, iff the interpreted sequential specification of obj simulates spec—i.e.,* $\textsf{interp-as-spec}(obj) \lesssim_{\mathbb{SP}} spec$*.*

Intuitively, if an object sequentially simulates a specification, it behaves according to the specification when its methods are executed sequentially.

**Definition 4.4.4** (Concurrent object refinement). *An object* $obj_1$ (concurrently) refines *another object* $obj_2$ *written* $obj_1 \lesssim_{\mathbb{C}} obj_2$*, if the observable behavior of* $obj_1$ *is included in that of* $obj_2$*, i.e.,* $obj_1 \lesssim_{\mathbb{C}} obj_2$ *iff* $\mathsf{beh}(obj_1) \subseteq \mathsf{beh}(obj_2)$*.*

| | |
|---|---|
| Specification simulation | $\lesssim_{\mathbb{SP}} : \mathsf{Spec}\ M \to \mathsf{Spec}\ M \to \mathsf{Prop}$ |
| Sequential object simulation | $\lesssim_{\mathbb{S}} : \mathsf{Object}\ M \to \mathsf{Spec}\ M \to \mathsf{Prop}$ |
| Concurrent refinement | $\lesssim_{\mathbb{C}} : \mathsf{Object}\ M \to \mathsf{Object}\ M \to \mathsf{Prop}$ |
| Linearizability | $\lesssim_{\mathbb{L}} : \mathsf{Object}\ M \to \mathsf{Spec}\ M \to \mathsf{Prop}$ |
| Horizontal composition | $+ : \mathsf{Object}\ M \to \mathsf{Object}\ N \to \mathsf{Object}\ (M + N)$ |
| Vertical composition | $\triangleright_O : \mathsf{Impl}\ M\ N \to \mathsf{Object}\ N \to \mathsf{Object}\ M$ |

FIGURE 4.11. Signatures of main definitions in Section 4.4

The objects $obj_1$ and $obj_2$ are called *concrete* and *abstract* objects, respectively. Note that concurrent refinement relates objects with a common high-level interface, while their low-level interfaces may differ: these are viewed as internal to each object.

Concurrent refinement is both reflexive and transitive, allowing verification of objects to proceed in steps. Transitivity allows us to decompose refinement proofs of an object into steps. Reflexivity allows us to refine only parts of a composite object.

**Linearizability.** We define linearizability of an object with respect to a specification as a concurrent refinement between the object and an "atomic object" associated with the specification.

**Definition 4.4.5** (Atomic object)**.** *Any sequential specification spec of an interface $M$ can be associated with an* atomic object $\mathsf{atomic}(spec) = (M, \mathsf{id}, spec)$*. (Recall the definition of the identity implementation:* $\mathsf{id} = (\mathsf{m} \mapsto (x \leftarrow \mathsf{m}; x))$*.)*

The atomic object trivially "wraps" the sequential specification as its low-level interface, delegating every method call to the corresponding method of the sequential specification. The intuition is that this object behaves atomically because every method call executes in just a single low-level step, modifying the low-level state according to the sequential specification and immediately returning a value. (Clients of such an object can still see some nondeterminism because it takes separate steps to call the method [spawn a thread], execute it, and return a value, and these steps may be interleaved with the steps of other method calls.)

This definition allows us to connect specification simulation and concurrent refinement. If one specification simulates another, then the former's atomic object refines the latter's atomic object.

**Proposition 4.4.6.** $spec \lesssim_{\mathbb{SP}} spec' \;\Rightarrow\; \mathsf{atomic}(spec) \lesssim_{\mathbb{C}} \mathsf{atomic}(spec')$.

**Definition 4.4.7** (Linearizability)**.** *An object obj is* linearizable *with respect to a sequential specification spec iff obj concurrently simulates the atomic object associated with spec, i.e.,* $obj \lesssim_{\mathbb{C}} \mathsf{atomic}(spec)$*. We then write* $obj \lesssim_{\mathbb{L}} spec$*.*[8]

[8]This simple presentation of linearizability in terms of a refinement relation from *obj* to (an object derived from) *spec* is equivalent to the original definition due to Herlihy and Wing [1990]. In an execution of the object $\mathsf{atomic}(spec)$, while clients observe the history $h$ of high-level method calls and returns, each method call is performed in a single internal step according to the sequential

The proof of linearizability of an object *obj* with respect to a specification *spec* can be carried out in two steps using an intermediate specification *spec*′: first prove that *obj* is linearizable with respect to *spec*′, then prove that *spec*′ simulates *spec*.

**Proposition 4.4.8.** $obj \lesssim_{\mathbb{L}} spec' \;\wedge\; spec' \lesssim_{\mathbb{SP}} spec \;\Rightarrow\; obj \lesssim_{\mathbb{L}} spec.$

Alternatively, linearizability of an object *obj* with respect to a specification *spec* can be proved hierarchically using an intermediate object *obj*′: first prove linearizability of *obj* with respect to the interpreted specification of *obj*′, then prove sequential simulation from *obj*′ to *spec*. We will use this decomposition later to verify a concurrent hash-map data structure.

**Proposition 4.4.9.** $obj \lesssim_{\mathbb{L}} \mathsf{interp\text{-}as\text{-}spec}(obj') \;\wedge\; obj' \lesssim_{\mathbb{S}} spec \;\Rightarrow\; obj \lesssim_{\mathbb{L}} spec.$

**Composition.** Objects support two fundamental composition patterns: *horizontal composition* corresponds to the union of interfaces, while *vertical composition* interprets the low-level calls of one implementation in terms of another implementation.

*Horizontal composition.* Given two interfaces $M_1$ and $M_2$, we write their disjoint union as $M_1 + M_2$. We can then define the horizontal composition of implementations and sequential specifications as follows. Given two implementations $impl_1 : \mathsf{Impl}\, M_1\, N_1$ and $impl_2 : \mathsf{Impl}\, M_2\, N_2$, their horizontal composition $impl_1 + impl_2 : \mathsf{Impl}\, (M_1 + M_2)\, (N_1 + N_2)$ simply implements high-level methods from $M_1$ using $impl_1$ and methods from $M_2$ using $impl_2$. This composition is defined formally as `bimap` in the category of event handlers (Sections 2.1.5 and 2.2.3). We can similarly define horizontal composition for sequential specifications $spec_1 + spec_2$ and objects $obj_1 + obj_2$. Horizontal composition satisfies the following property with respect to concurrent refinement, allowing us to refine each summand independently. Herlihy and Wing [1990] call this property the compositionality of linearizability:

**Proposition 4.4.10.** $obj_1 \lesssim_{\mathbb{C}} obj_1' \wedge obj_2 \lesssim_{\mathbb{C}} obj_2' \;\Rightarrow\; (obj_1 + obj_2) \lesssim_{\mathbb{C}} (obj_1' + obj_2').$

Reflexivity of $\lesssim_{\mathbb{C}}$ allows keeping parts of a composite object the same while others are rewritten.

*Vertical composition.* The vertical composition $\triangleright$ of an implementation $impl_1 : \mathsf{Impl}\, M_1\, M_2$ on top of another $impl_2 : \mathsf{Impl}\, M_2\, M_3$, written $impl_1 \triangleright impl_2 : \mathsf{Impl}\, M_1\, M_3$, is defined by interpreting the low-level calls in the body of $impl_1$ using the methods of $impl_2$. The operation $\triangleright$ is the categorical composition in the category of event handlers (Section 2.2.3). We can then define the vertical composition $\triangleright_O$ of an implementation $impl_1 : \mathsf{Impl}\, M_1\, M_2$ on top of an object $obj_2$ with interface $M_2$, which is an object $impl_1 \triangleright_O obj_2$ with interface $M_1$, by composing $impl_1$ with the implementation contained in $obj_2$.

Vertical composition satisfies the following property with respect to concurrent refinement, allowing us to refine the internal object *obj*′ to *obj*:

specification *spec*, in the interval between its initial call and final return. The sequence $s$ of those internal steps is a (sequential) trace of *spec*. In fact, $s$ is a linearization of the observed history $h$, in the sense of Herlihy and Wing [1990]. The set $\mathsf{beh}(\mathsf{atomic}(spec))$ is exactly the set of histories $h$ that can be linearized to sequential traces of *spec*. An object *obj* is linearizable with respect to a specification *spec* iff all of its histories $h \in \mathsf{beh}(obj)$ can be linearized to sequential traces of *spec*. Thus, an object *obj* is linearizable iff $\mathsf{beh}(obj)$ is a subset of $\mathsf{beh}(\mathsf{atomic}(spec))$.

**Proposition 4.4.11.** $\mathit{obj} \lesssim_{\mathbb{C}} \mathit{obj}' \;\Rightarrow\; (\mathit{impl} \rhd_O \mathit{obj}) \lesssim_{\mathbb{C}} (\mathit{impl} \rhd_O \mathit{obj}')$.

It will be convenient to write the construction of an object $(N, \mathit{impl}, \mathit{spec})$ as $\mathit{impl} \rhd_S \mathit{spec}$, leaving implicit the low-level interface $N$. Then $\rhd_O$ is defined by $\mathit{impl} \rhd_O (\mathit{impl}' \rhd_S \mathit{spec}) = (\mathit{impl} \rhd \mathit{impl}') \rhd_S \mathit{spec}$. Furthermore, because $\rhd$, $\rhd_O$, and $\rhd_S$ expect different types of right operand, we can unambiguously omit parentheses in expressions involving only these three operators, and thanks to the associativity of $\rhd$, we can even freely reassociate parentheses, modulo changing the operators so that the resulting expression is still well-typed, *e.g.*, $(f \rhd g) \rhd_O \mathit{obj} = f \rhd_O (g \rhd_O \mathit{obj})$.

Two implementations $\mathit{impl}_1$ and $\mathit{impl}_2$ are considered equal when they map methods to the same programs, up to ignoring finite runs of silent steps $\tau$. This notion of equality is a congruence with respect to $+$ and $\rhd$. Proposition 4.4.12 summarizes equations relating $+$ and $\rhd$ that will be needed in the proof in Section 4.7. Those equations involve the following identity elements for $+$ and $\rhd$. As we saw before, for any interface $M$, there is a trivial implementation $\mathsf{id} : \mathsf{Impl}\, M\, M$ which maps a method to a program that simply calls the same method. Such a trivial implementation is an identity for vertical composition $\rhd$. The empty interface $\mathsf{Empty}$, containing no methods, is the identity for horizontal composition $+$, in the sense of monoidal categories: there is a left unitor, that is to say, an implementation $\mathsf{empty_L} : \mathsf{Impl}\,(\mathsf{Empty} + M)\, M$ for every interface $M$, satisfying various equations; there are also a right unitor and an associator, omitted for brevity.

**Proposition 4.4.12.** *Vertical and horizontal composition define a monoidal category whose objects and morphisms are respectively interfaces and implementations. In other words, they satisfy certain equations, including (among others),*

$$\begin{array}{ll} (1)\ (f \rhd g) \rhd h = f \rhd (g \rhd h) & (3)\ (f + g) \rhd (h + k) = (f \rhd h) + (g \rhd k) \\ (2)\ \mathsf{id} \rhd f = f \rhd \mathsf{id} = f & (4)\ (id + g) \rhd \mathsf{empty_L} = \mathsf{empty_L} \rhd g \end{array}$$

*for any implementations $f$, $g$, $h$ and $k$ with interfaces that make the equations well-typed.*

## 4.5. Verification of Linearizability

We now present a novel proof principle for concurrent refinement. It allows reasoning about the method bodies as programs (interaction trees) and does not require them to be translated to low-level labeled transition systems. Further, instead of stating and proving a simulation relation on the global states of a pool of concurrent programs, it factors and decomposes the simulation relation into separate and simpler invariants on the object state and programs, and it factors the simulation proof into separate and simpler proof obligations. The proof technique is a general method that supports verification of both lock-based and lock-free algorithms. In Section 4.5.1, we apply this principle and the hierarchical proof techniques that we saw in Section 4.4 to verify the linearizability of a concurrent hash-map object. We use the same principle in Section 4.5.2, to prove the linearizability of a concurrent histogram, and in Section 4.6.1, to prove the correctness of Transactional Mutex Locks (TML) [Dalessandro et al., 2010], an implementation of transactional memory.

$$\begin{aligned} inc' := \; & i \leftarrow r.read(); \\ & r.write(i+1); \\ & i+1 \end{aligned}$$

FIGURE 4.12. $inc'$ method

The work presented in this section—the proof principle and the examples of objects verified using it—was completed by my coauthors prior to my joining this project [Lesani et al., 2022].

In a concurrent execution (Section 4.3), the state of an object is a pair $(s \mid \Theta)$ of a data state $s$ and a thread pool $\Theta$. Stating invariants about these pairs is complicated and distracts from interesting similarities between the concrete and abstract objects themselves. Further, reasoning about the pool of threads involves boilerplate steps such as reasoning about spawning and returning threads and stating and proving conditions for every thread or pair of threads in the pool.

**Invariant relations.** To prove that a concrete object refines an abstract object, we need to define three relations and prove five obligations. Start with a concrete object $(N, impl, spec)$ on the low-level specification $spec = (S, s_0, \Delta)$ and an abstract object $\left(N', impl', spec'\right)$ on the low-level specification $spec' = (S', s'_0, \Delta')$. The *data relation* $\mathrm{R_D}$ captures the relation between the concrete and abstract data. The *program relation* $\mathrm{R_P}$ captures the relation between the corresponding concrete and abstract programs. Finally, the *interprogram relation* $\mathrm{R_I}$ captures the mutual relation between pairs of concrete and abstract programs.

For example, consider the *inc* method that we saw in Figure 4.2 of a counter object $c$, compared with the $inc'$ method of a counter object $c'$ in Figure 4.12. The concrete counter $c$ is linearizable with respect to the interpreted specification of the abstract counter $c'$. Informally, the proof principle captures the following invariants. (1) The *data relation* $\mathrm{R_D}$ says that the states of the two base objects $r$ are equal. (2) The *program relation* $\mathrm{R_P}$ captures a correspondence between the intermediate concrete and abstract programs. It is defined based on the linearization point, a low-level method call in the concrete program where the abstract program should be executed instantaneously. When the intermediate concrete program has not yet reached the linearization point, the corresponding intermediate abstract program is not executed yet. After the linearization point, the corresponding abstract program is already evaluated to a value. In this example, the linearization point is reached when the *cas* call succeeds. The entire abstract method $inc'$ is executed then. (3) In this example, the *interprogram relation* $\mathrm{R_I}$ is trivially `true`. Our framework provides simple instantiations of the proof technique as well. For example, an instance does not require the specification of the interprogram relation (i.e., it is simply instantiated as true).

As another example, consider the programs in Figure 4.14. (To simplify the example, we show only the bodies of the methods.) On the right, we have the implementation $p'$ of a sequential object $o'$ with two method calls on a base object $b$. On the left, we have the implementation $p$ of a concurrent object $o$ that protects the same calls with a lock $l$. The concrete object $o$ is linearizable with respect to

$$\textsc{InitObl}:\ \mathrm{R_D}(s_0,\ s'_0)$$

$$\begin{aligned}\textsc{PPSymObl}:\ & \mathrm{R_I}(p_1,\ p_2,\ s,\ p'_1,\ p'_2,\ s') \Rightarrow \\ & \mathrm{R_I}(p_2,\ p_1,\ s,\ p'_2,\ p'_1,\ s')\end{aligned}$$

$$\begin{aligned}\textsc{CallObl}:\ & \mathrm{R_D}(s,\ s') \Rightarrow \\ & \mathrm{R_P}(\mathit{impl}(\mathsf{m}),\ s,\ \mathit{impl}'(\mathsf{m}),\ s') \\ & \wedge (\mathrm{R_P}(p,\ s,\ p',\ s') \Rightarrow \\ & \quad \mathrm{R_I}(\mathit{impl}(\mathsf{m}),\ p,\ s,\ \mathit{impl}'(\mathsf{m}),\ p',\ s'))\end{aligned}$$

$$\textsc{RetObl}:\ \mathrm{R_D}(s,\ s')\ \wedge\ \mathrm{R_P}(v,\ s,\ p',\ s')\ \Rightarrow\ p' = v$$

$$\begin{aligned}&\textsc{StepObl}: \\ &\quad \mathrm{R_D}(s_1,\ s'_1) \wedge \\ &\quad \mathrm{R_P}((x \leftarrow \mathsf{n}; f(x)),\ s_1,\ p'_1,\ s'_1) \wedge \\ &\qquad (\mathsf{n}, s_1, v, s_2) \in \Delta \Rightarrow \\ &\qquad \exists\ p'_2\ s'_2, \\ &\qquad\quad (s'_1, p'_1) \rightarrow^*_{\Delta'} (s'_2, p'_2) \\ &\qquad\quad \wedge \mathrm{R_D}(s_2,\ s'_2) && (1) \\ &\qquad\quad \wedge \mathrm{R_P}(f(v),\ s_2,\ p'_2,\ s'_2) \\ &\qquad\quad \wedge\ (\forall\ p\ p', && (2) \\ &\qquad\qquad \mathrm{R_P}(p,\ s_1,\ p',\ s'_1) \wedge \\ &\qquad\qquad \mathrm{R_I}((x \leftarrow \mathsf{n}; f(x)),\ p,\ s_1,\ p'_1,\ p',\ s'_1) \Rightarrow \\ &\qquad\qquad \mathrm{R_P}(p,\ s_2,\ p',\ s'_2) \\ &\qquad\qquad \wedge \mathrm{R_I}(f(v),\ p,\ s_2,\ p'_2,\ p',\ s'_2)) \\ &\qquad\quad \wedge\ (\forall\ p_1\ p'_1\ p_2\ p'_2, && (3) \\ &\qquad\qquad \mathrm{R_P}(p_1,\ s_1,\ p'_1,\ s'_1) \wedge \\ &\qquad\qquad \mathrm{R_P}(p_2,\ s_1,\ p'_2,\ s'_1) \wedge \\ &\qquad\qquad \mathrm{R_I}(p_1,\ p_2,\ s_1,\ p'_1,\ p'_2,\ s'_1) \Rightarrow \\ &\qquad\qquad \mathrm{R_I}(p_1,\ p_2,\ s_2,\ p'_1,\ p'_2,\ s'_2))\end{aligned}$$

The concrete object is $(N, \mathit{impl}, \mathit{spec})$ on the low-level specification $\mathit{spec} = (S, s_0, \Delta)$, and the abstract object is $\left(N', \mathit{impl}', \mathit{spec}'\right)$ on the low-level specification $\mathit{spec}' = (S', s'_0, \Delta')$. Unprimed and primed variables are used for the concrete and abstract objects respectively. The data relation is $\mathrm{R_D}(s, s')$, the program relation is $\mathrm{R_P}(p, s, p', s')$ and the interprogram relation is $\mathrm{R_I}(p_1, p_2, s, p'_1, p'_2, s')$. The free variables are universally quantified.

FIGURE 4.13. Proof principle for concurrent refinement

the interpreted specification of the abstract object $o'$. Informally, the proof principle captures the following invariants. (1) The *data relation* $\mathrm{R_D}$ says that, when the lock $l$ is in the released mode, the states of the two base objects $b$ are equal. (2) The *program relation* $\mathrm{R_P}$: In this example, the linearization point is reached when the lock is released. When the intermediate concrete program has not yet reached the

|  | $p :=$ | $p' :=$ |
|---|---|---|
| 1 | $l.\mathsf{lock}\,();$ |  |
| 2 | $b.\mathsf{m}_1();$ | $b.\mathsf{m}_1();$ |
| 3 | $x \leftarrow b.\mathsf{m}_2();$ | $x \leftarrow b.\mathsf{m}_2();$ |
| 4 | $l.\mathsf{unlock}\,();$ |  |
|  | $x$ | $x$ |

FIGURE 4.14. Proof principle example

linearization point, the corresponding intermediate abstract program is not executed yet. However, it holds that if part of the abstract program that corresponds to the executed part of the concrete program is executed, then the state of the abstract base object will be the same as the concrete base object. When the concrete program $p$ reaches line 4, the entire abstract program $p'$ is executed and results in the same state for the base objects. (3) *Interprogram relation* $\mathrm{R_I}$: No two concrete programs can be in the critical section (at lines 2 and 3).

More precisely, the data relation $\mathrm{R_D}(s, s')$ defines an invariant between the low-level concrete state $s$ and the abstract state $s'$. The program relation $\mathrm{R_P}(p, s, p', s')$ defines an invariant between the concrete program $p$ paired with data state $s$ and the corresponding abstract program $p'$ paired with data state $s'$. The interprogram relation $\mathrm{R_I}(p_1, p_2, s, p'_1, p'_2, s')$ defines a mutual relation among two running concrete programs $p_1$ and $p_2$ (with data state $s$) and two corresponding abstract programs $p'_1$ and $p'_2$ (with data state $s'$).

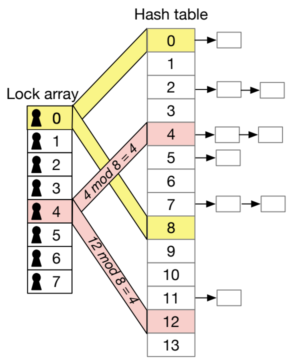
The overall simulation relation is simply defined as the conjunction of the data relation $\mathrm{R_D}$ between the concrete and abstract data states, the program relation $\mathrm{R_P}$ for each program in the thread pool (and its corresponding abstract program), and the interprogram relation $\mathrm{R_I}$ for each pair of programs in the thread pool (and their corresponding abstract programs).

**Proof obligations.** Figure 4.13 summarizes the proof obligations. These proof obligations imply that the overall simulation relation is preserved. The obligation INITOBL states that the data relation $\mathrm{R_D}$ holds between the initial concrete state $s_0$ and abstract state $s'_0$. PPSYMOBL states that the interprogram relation $\mathrm{R_I}$ is symmetric over programs.

The obligation CALLOBL states that when a method is called, the invariants are preserved. Let us consider a method $\mathsf{m}$ implemented by the program *impl*($\mathsf{m}$) in the concrete object and by *impl'*($\mathsf{m}$) in the abstract object. The new pair of concrete *impl*($\mathsf{m}$) and abstract *impl'*($\mathsf{m}$) programs should be in the program relation $\mathrm{R_P}$ with every concrete state $s$ and abstract state $s'$ that are in the data relation $\mathrm{R_D}$. In addition, the two programs *impl*($\mathsf{m}$) and *impl'*($\mathsf{m}$) should be in the interprogram relation $\mathrm{R_I}$ with any pair of concrete $p$ and abstract $p'$ programs that are in the program relation $\mathrm{R_P}$. When the concrete object returns a value, the obligation RETOBL requires the abstract object to return the same value. If a concrete leaf value $v$ is in program relation $\mathrm{R_P}$ with an abstract program $p'$ and states $s$ and $s'$ that are also in the data relation $\mathrm{R_D}$, then the abstract program $p'$ should be the same leaf value $v$.

```
      m(k, x) :=
L_1     s ← size();
        let i := (hash k) modulo s in
L_2     y ← arrayCall(i, m(k, x));
        y
```

(a) hash-map



(b) Concurrent hash-map

```
      m(k, x) :=
L_1     ls ← locks.size();
        let li := (hash k) modulo ls in
L_2     _ ← locks.arrayCall (li, lock);
L_3     bs ← buckets.size();
        let bi := (hash k) modulo bs in
L_4     y ← buckets.arrayCall (bi, m(k, x));
L_5     _ ← locks.arrayCall (li, unlock);
L_6     y
```

(c) conc-hash-map

In (a), the metavariable m stands for either the get or put method on the map interface. The variable $x$ is the second argument, either nothing (for get) or the value to write (for put). In (c), locks.m$'$() and buckets.m$'$() refer to methods m$'$ of the lock and bucket arrays respectively.

FIGURE 4.15. Implementations of sequential and concurrent hash-map objects

The obligation StepObl states that the invariants are preserved by program steps: when the program relation $R_P$ holds for the pair of a concrete program $x \leftarrow \mathsf{n}; f(x)$ and an abstract program $p'_1$ and a pair of concrete and abstract data states $s$ and $s'$ that are in the data relation $R_D$, if the concrete program steps, then the abstract program $p'_1$ can take steps such that (1) both the data and program relations are preserved after the step for the resulting states and programs. In addition, it states that (2) if before the step, a pair of concrete and abstract programs $p$ and $p'$ were in the program relation $R_P$ and also in the interprogram relation $R_I$ with the concrete call program $x \leftarrow \mathsf{n}; f(x)$ and its corresponding abstract program $p'_1$, then after the step, the program and interprogram relations are preserved. Further, it states that (3) if before the step, two pairs of concrete and abstract programs $p_1, p'_1$ and $p_2, p'_2$ were each in the program relation $\mathrm{R_P}$ and also in the interprogram relation $\mathrm{R_I}$ with each other, then after the step, their interprogram relation is preserved with the new data states. This proof principle is for forward simulation, and the StepObl condition states proof obligations for a method call as a forward step.

For our first example above, the three predicates of StepObl hold as follows: (1) After the step, $R_D$ holds, because the value of the register is incremented by either both or neither of the concrete and abstract programs. Further, $R_P$ is trivially preserved as the abstract steps are taken according to the $R_P$ relation. (2) The $R_P$ relation for another pair of concrete and abstract programs is preserved since $R_P$ is independent of the data state. (3) $R_I$ trivially holds.

In our second example: (1) After the step, $R_D$ holds because if the lock is released, the abstract program takes a step as well, and the concrete and abstract post-states become the same. Further, $R_P$ is trivially preserved as the abstract steps are taken according to the $R_P$ relation. (2) When a program steps, the $R_P$ relation for any other program is preserved as follows. The definition of $R_P$ allows the concrete program to be in an intermediate step of the critical section only when the lock is in the acquired state. By the mutual-exclusion property of $R_I$, at most one of the stepping program or the other program is in the critical section. If the other program is in the critical section, the stepping program is not in the critical section and cannot release the lock to break the $R_P$ relation for the other program. If the other program is not in the critical section, any change that the stepping program makes to the state of the lock does not affect the $R_P$ relation for the other program. Further, the mutual-exclusion relation $R_I$ between the two is preserved. If the other program is already in the critical section, the state of the lock is acquired, and the stepping program cannot acquire the lock and step into the critical section. (3) The relation $R_I$ for two other programs is preserved trivially because a stepping program cannot affect the mutual-exclusion property between them.

**4.5.1. Hierarchical Verification of Linearizability.** We saw the sequential specification of maps, map-spec, in Section 4.3. We next implement both sequential and concurrent hash-map objects, hash-map and conc-hash-map. The former uses just an array of buckets; the latter, in addition, uses an array of locks. We show that the concurrent hash-map object conc-hash-map is linearizable with respect to the sequential specification of maps map-spec. The proof is divided into two steps, by Lemma 4.4.9, with the sequential object hash-map as an intermediate specification: we first show that hash-map sequentially simulates map-spec, and then we show that conc-hash-map is linearizable with respect to the interpreted sequential specification of hash-map.

*Sequential hash-map.* The sequential hash-map object hash-map is implemented as a vertical composition on top of an array object, which represents an indexed sequence of other objects. (Formally, the array object type is parametric—at the metalevel—in the cell object type: for each type of cells, there is a separate type of arrays of those cells.) The array interface provides two methods: a method arrayCall that, given an index and a method call, calls the method on the object stored at that index; and a method size that returns the size of the array. Each cell of the array refines the object that the array is instantiated with.

We use a closed-address hash-map, where each bucket refers to a set of items. In the hash-map object, the array elements are map objects that represent buckets with colliding keys. The bucket map bmap can be any simple map object (thus, the map interface plays both high- and low-level roles in the implementation of the hash-map). Figure 4.15a presents the implementation of methods of the hash-map. Map methods are parametrized by the key $k$ being accessed, which we hash to obtain the index $i$ of the corresponding bucket. We then call the input method on the map object contained in that bucket and return its result.

If the given bucket-map object sequentially simulates the map specification, the hash-map object sequentially simulates the map specification as well.

**Lemma 4.5.1** (Sequential simulation of map object)**.**

$$\mathsf{bmap} \lesssim_{\mathbb{S}} \mathsf{map\text{-}spec} \;\Rightarrow\; \mathsf{hash\text{-}map} \lesssim_{\mathbb{S}} \mathsf{map\text{-}spec}$$

*Concurrent hash-map.* We implement a striped concurrent hash-map as an object conc-hash-map. The structure of the buckets is similar to the sequential hash-map that we saw above. The bucket map bmap is not linearizable (not thread-safe). As Figure 4.15b shows, the implementation uses lock striping to protect access to buckets. An array of locks is used, and the lock at index $i$ protects all the buckets at indices $j$ such that $j$ is $i$ modulo the size of the lock array. Thus, the concurrent hash-map object is a vertical composition on the horizontal composition of two objects: an array of bucket maps (bmap) and an array of locks (lock). Figure 4.15c presents the implementation of the methods of the concurrent hash-map. It gets the size of the lock array, gets the input key of the call, computes the lock index as the hash value of the input key modulo that size, acquires the lock at the lock index, then gets the size of the bucket array, computes the bucket index as the hash value of the input key modulo that size, performs the input method on the bucket index, releases the lock at the lock index, and returns the resulting value.

To prove the correctness of the concurrent hash-map, we use the sequential hash-map as an intermediate specification. The following lemma states that the concurrent hash-map object is linearizable with respect to the interpreted specification of the sequential hash-map.

**Lemma 4.5.2** (Linearizability of lock object)**.**

$$\mathsf{lock} \lesssim_{\mathbb{L}} \mathsf{lock\text{-}spec} \;\Rightarrow\; \mathsf{conc\text{-}hash\text{-}map} \lesssim_{\mathbb{L}} \mathsf{interp\text{-}as\text{-}spec}(\mathsf{hash\text{-}map})$$

Finally, we apply the hierarchical technique of Lemma 4.4.9 to Lemmas 4.5.2 and 4.5.1. We conclude that the concurrent hash-map object is linearizable with respect to the map specification.

**Corollary 4.5.3** (Linearizability of concurrent hash-map)**.**

$$\mathsf{bmap} \lesssim_{\mathbb{S}} \mathsf{map\text{-}spec} \;\wedge\; \mathsf{lock} \lesssim_{\mathbb{L}} \mathsf{lock\text{-}spec} \;\Rightarrow\; \mathsf{conc\text{-}hash\text{-}map} \lesssim_{\mathbb{L}} \mathsf{map\text{-}spec}$$

```
      hist-imp :=
          get (k) :=
$G_1$         $r \leftarrow$ map.get $(k)$;
$G_2$         $r$

          inc (k) :=
$I_1$         $r \leftarrow$ map.get $(k)$;
$I_2$         match $r$ with
$I_3$         | $\lceil v \rceil \Rightarrow$
$I_4$             $s \leftarrow$ map.replace $(k, v, v + 1)$;
$I_5$             if $(s) v + 1$
$I_6$             elseinc $(k)$
$I_7$         | $\perp \Rightarrow$
$I_8$             $r \leftarrow$ map.putIfAbsent$(k, 1)$;
$I_9$             match $r$ with
$I_{10}$            | $\lceil \_ \rceil \Rightarrow$ inc $(k)$
$I_{11}$            | $\perp \Rightarrow 1$
```

FIGURE 4.16. Implementation of a histogram object on a map object

**4.5.2. Linearizability of Vertical Compositions.** Linearizability ensures that concurrent method calls on the object appear to execute atomically and behave according to the sequential specification of the object. This guarantee is only provided for individual method calls on the object. However, methods of a vertical composition on top of the object may make multiple calls to the object; therefore, they are not necessarily atomic. Studies of production code [Shacham et al., 2011] show that atomicity bugs are prevalent in vertical compositions. In this section, we see how a concurrent histogram can be implemented as a vertical composition on top of a concurrent hash-map and how its linearizability can be verified using the same proof technique that we saw in Section 4.5.

A histogram is a data structure that represents values for a set of bars. We first consider the sequential specification of histograms hist-spec. The interface of a histogram is parametric in terms of the key type $K$ for the bars, and it provides two methods: get$(k)$ and inc$(k)$. The state is a mapping $m$ from keys $K$ to optional natural values option $\mathbb{N}$. The transition system of hist-spec makes a transition on get$(k)$ that keeps the state the same and returns the value of $k$ in the current state $m$. It makes two different transitions on inc$(k)$ depending on whether $k$ is in the domain of the current state map $m$. If $k$ already exists, it increments its value in $m$; otherwise, it adds to the map $m$ a mapping from $k$ to the "some" value of 1. In both transitions, it returns the value of $k$ in the post-state.

We will see a histogram object that is implemented by a vertical composition on top of a map object. This map object provides an extended interface: in addition to the methods get$(k)$ and put$(k, v)$, it provides putIfAbsent$(k, v)$ and replace$(k, v, v')$. Similar to the basic map, the sequential specification of the extended map emap-spec stores a mapping $m$ from keys to optional values. Its transition system makes two different transitions on $\mathsf{putIfAbsent}(k, v)$: if $k$ is not in the domain of the state map $m$, it extends $m$ with a mapping from $k$ to $v$; otherwise, it leaves $m$ the same. In both cases, it returns the optional value of $k$ in the prestate. Similarly, there are two possible transitions on $\mathsf{replace}(k, v, v')$: if the value of $m$ for $k$ is $v$, it replaces $v$ with $v'$ and returns true; otherwise, it leaves the state unchanged and returns false.

The implementation hist-imp of the histogram object is presented in Figure 4.16. The get method is simply delegated to the underlying map object (at $G_1$). On the other hand, a naive implementation of the inc method, which would simply get the current value and put back an incremented value, is not atomic, and concurrent calls on it can easily violate the sequential specification. In the implementation of inc, the current value of the key $k$ is first obtained from the underlying map object (at $I_1$). A "some" value containing an actual value $v$ is represented as $\lceil v \rceil$, and the "none" value is represented as $\perp$. If the key $k$ is already mapped to some value $v$ (at $I_3$), it should be updated to $v + 1$. In order to avoid racing updates to the underlying map, the inc method attempts to replace $v$ atomically with $v + 1$ (at $I_4$). If the value of $k$ is not changed between $I_1$ and $I_4$ by a concurrent update, the replace call succeeds and returns true. In this case, the increment is performed and the new value $v + 1$ is returned (at $I_5$). Otherwise, the inc method is repeated by a recursive call (at $I_6$). (We note that the inc method has a corecursive definition.) If the key $k$ is not in the underlying map (at $I_7$), a new mapping from $k$ to 1 needs to be added. Similarly to the previous case, in order to avoid racing updates to the underlying map, the inc method attempts to put the value 1 for $k$ atomically via a putIfAbsent call (at $I_8$). The putIfAbsent method always returns the previous value of $k$. Therefore, if it fails, it returns some value (at $I_{10}$), and the inc method is called again. If it succeeds, it returns none $\perp$ (at $I_{11}$), and the new value 1 is returned.

The vertical composition of the histogram implementation on top of the extended map is linearizable with respect to the sequential specification of the histogram.

THEOREM 4.5.4. $\mathsf{hist\text{-}imp} \triangleright_O \mathsf{emap\text{-}spec} \lesssim_{\mathbb{L}} \mathsf{hist\text{-}spec}$.

The proof of this theorem uses the proof technique that we saw in Section 4.5 and used in Section 4.5.1. The invariants for this proof are the following: (1) The *data relation* $\mathrm{R_D}$ says that the states of the underlying concrete map and the abstract map of the histogram are equal. (2) The *program relation* $\mathrm{R_P}$ captures a correspondence between the intermediate concrete and abstract programs according to the linearization points. Similar to the previous use cases, when the intermediate concrete program has not yet reached the linearization point, the corresponding intermediate abstract program is not executed yet. After the linearization point, the corresponding abstract program is already evaluated to a value. The linearization point of the get method is the get call on the underlying map (at $G_1$). The linearization point of the inc method is its last replace (at $I_4$) or putIfAbsent (at $I_{10}$) method call before its returns. (3) The *interprogram relation* $\mathrm{R_I}$ is trivially true.

## 4.6. Transactions

Linearizability offers simple guarantees at the level of individual method calls, but it is still up to the caller to compose these guarantees to reason about complex

$$\frac{(\mathsf{m}, s, r, s') \in \Delta}{(?\mathsf{m}, s, \lceil r \rceil, s') \in \Delta'} \qquad \frac{}{(?\mathsf{m}, s, \bot, s) \in \Delta'}$$

FIGURE 4.17. Abortable transition relation

programs that successively call multiple methods. A more convenient model for concurrent programming is offered by *transactions*: arbitrary programs, combining multiple calls to some interface, which are expected to run atomically as a whole. In this section, we first characterize the specification of transactions and then present objects that implement the specification.

Given an interface, a transaction is a program $p$ on this interface. A transaction is expected to execute atomically: either execute completely or abort without any effect. To embody this idea, we introduce a special interface whose one method $?\mathsf{exec}$ takes a whole program as an argument, represented as an interaction tree. This method is expected to run the transaction atomically. We equip $?\mathsf{exec}$ with a sequential specification by lifting the sequential specification of the underlying interface used by the transaction, as we describe next.

**Program specification.** The program specification corresponding to a given specification $\mathit{spec} = (S, s_0, \Delta)$ for the interface $M$, written $\mathsf{prog\text{-}spec}(\mathit{spec})$, is the specification $(S, s_0, \Delta')$, which describes an interface $\mathsf{Prog}\, M$ with just one method, $\mathsf{exec}(p)$, where $p$ is itself a program on $M$. The relation $\Delta'$ is defined as $(\mathsf{exec}(p), s, v, s') \in \Delta'$ if and only if $(s,\ p) \to^*_{\mathit{spec}} (s',\ v)$, which says that when $\mathsf{exec}(p)$ runs starting from the initial state $s$, it ends in a state $s'$ obtained by interpreting the sequence of method calls in $p$ based on *spec*.

**Abortable specifications.** Transactions execute concurrently and may conflict on their accesses to the shared state. Therefore, atomic execution of one transaction may prevent atomic execution of another. Thus, transactions may be aborted, and the client can either retry immediately or back off to reduce contention and retry. To capture this behavior, we define abortable specifications.

First, an interface $M$ can be translated into an *abortable interface* $\mathbf{?}M$. For any method $\mathsf{m} : M\, R$, there is a corresponding *abortable method* $?\mathsf{m} : \mathbf{?}M\ (\{\bot\} + R)$, where the result of the method is made optional: it is either “some” value $\lceil r \rceil$ containing an actual result $r$, or it is the “none” value $\bot$.

The abortable version of a given specification $(S, s_0, \Delta)$ over the interface $M = \overline{\mathsf{m}}$, written $\mathsf{ab\text{-}spec}(\mathit{spec})$, is then a specification $(S, s_0, \Delta')$ over the abortable interface $\mathbf{?}M = \overline{?\mathsf{m}}$. The transition relation $\Delta'$, shown in Figure 4.17, can nondeterministically either execute the method call and return its result $\lceil r \rceil$ or else abort without changing the state and return the “none” value $\bot$.

**Transactional specification.** Based on the above definitions of program and abortable specifications, we can now define a transactional specification, which characterizes the atomic execution of full programs, where aborting is always a possibility. The transactional specification of a given specification *spec*, written $\mathsf{trans\text{-}spec}(\mathit{spec})$, is simply: $\mathsf{ab\text{-}spec}(\mathsf{prog\text{-}spec}(\mathit{spec}))$.

$$\begin{aligned}\mathsf{prog\text{-}spec} &: \mathsf{Spec}\,M \to \mathsf{Spec}\,(\mathsf{Prog}\,M)\\ \mathsf{ab\text{-}spec} &: \mathsf{Spec}\,M \to \mathsf{Spec}\,(\mathbf{?}M)\\ \mathsf{trans\text{-}spec} &: \mathsf{Spec}\,M \to \mathsf{Spec}\,(\mathbf{?}\mathsf{Prog}\,M)\end{aligned}$$

FIGURE 4.18. Program, abortable, and transactional specification transformers

```
$\mathsf{instrument}(?\mathsf{exec}(p)) :=$
    $t \leftarrow \mathsf{init}\,();$
    $\mathsf{instrument}'(t,\ p)$

$\mathsf{instrument}'(t,\ p) :=$
    match $p$ with
    $|\ y \leftarrow \mathsf{m}(x); f(y) \Rightarrow r \leftarrow \mathsf{lift}(t,\ \mathsf{m}(x)); \mathsf{instRet}(f,\ r)$
    $|\ r \Rightarrow o \leftarrow \mathsf{commit}(t); \mathsf{instEnd}(r,\ o)$
    $|\ \tau; p \Rightarrow \tau; \mathsf{instrument}'(t, p)$

$\mathsf{instRet}(f,\ r) :=$
    let $\langle t, o\rangle := r$ in
    match $o$ with
    $|\ \lceil r \rceil \Rightarrow \tau; \mathsf{instrument}'(t,\ f(r))$
    $|\ \perp\ \Rightarrow \mathsf{abort}\ t; \perp$

$\mathsf{instEnd}(r,\ o) :=$
    match $o$ with
    $|\ \lceil t \rceil \Rightarrow \mathsf{abort}\ t; \perp$
    $|\ \perp \Rightarrow\ \lceil r \rceil$
```

FIGURE 4.19. Transaction instrumentation

**Strict serializability.** Based on the above definition of transactional specification, we now define the strict serializability of an object. An object *obj* that implements interface $M$ is strictly serializable with respect to a specification *spec* of $M$, written *obj* $\lesssim_{\mathbb{SS}}$ *spec*, if and only if *obj* is linearizable with respect to the transactional specification of *spec*, i.e., *obj* $\lesssim_{\mathbb{L}}$ trans-spec(*spec*).

There is a close relation between linearizability and strict serializability. Herlihy and Wing [1990] mention that "linearizability can be viewed a special case of strict serializability where transactions are restricted to consist of a single operation." Our modular framework captures strict serializability as an instance of linearizability with "operations" (*i.e.*, methods) as programs.

**Transactional objects.** Having defined strict serializability, we now turn our attention to constructing verified transactional objects—objects that provably meet the specification trans-spec(*spec*) and therefore provide ?exec methods, which take as input transactions (programs) and run them atomically. We define a two-piece template for doing so. First, we *instrument* the input program, inserting "life-cycle" methods to initiate, commit, and abort transactions. Second, we implement the methods of those instrumented transactions in an object, called a *transaction protocol object*. Below, we first consider the interface of a transaction protocol and the corresponding instrumentation. Then we construct the transactional object as a vertical composition of the instrumented program on top of a transaction protocol object.

Note that this construction is a reusable instance of the compositional definition and verification methodology that we saw in Section 4.4. The instrumentation function is independent of the transaction-protocol object. Therefore, the user programs can be instrumented independently and can be composed with different underlying protocols that have different performance characteristics. In addition, decoupling the protocols allows them to be used as the underlying objects of other transformations, as we will see for predicated objects in Section 4.7.

**Transaction protocol interface.** A transaction protocol interface $\mathsf{trans}(M)$ wraps another interface $M$, making it so that the interface $M$ can be used inside of a transaction. For instance, we can obtain the interface for transactional memory by wrapping the map interface into a transactional protocol interface. A transaction protocol interface $\mathsf{trans}(M)$ provides the following four methods: $\mathsf{init}$, $\mathsf{lift}$, $\mathsf{abort}$, and $\mathsf{commit}$, which define the life-cycle methods of a transaction.

The method $\mathsf{init}\colon \mathsf{TLocal}$ initializes the transaction and returns the transaction-local state (for example, an initial timestamp for the transaction). This state is passed between and updated by the other three methods during execution. The method $\mathsf{lift}\colon \mathsf{TLocal} \times M\,R \to (\mathsf{TLocal} \times \mathsf{Option}\,R)$ takes the current transaction-local state and a method call $\mathsf{m}$ and tries to execute $\mathsf{m}$. It returns a pair including the updated transaction-local state. The execution of $\mathsf{m}$ may not be successful, due to a conflict, so $\mathsf{lift}$ also returns an optional value, where $\bot$ indicates failure, and $\lceil r \rceil$ indicates that $\mathsf{m}$ successfully returned $r$. If a call is not successful, then the method $\mathsf{abort}\colon \mathsf{TLocal} \to \mathsf{unit}$ is subsequently called to clean up before the transaction is aborted. Finally, the method $\mathsf{commit}\colon \mathsf{TLocal} \to \mathsf{Option}\,\mathsf{TLocal}$ commits the transaction. Committing may itself fail due to a conflict, so it returns an option value. If it is *not* successful, $\mathsf{commit}$ returns the transaction-local state to be passed to the subsequent $\mathsf{abort}$ call; otherwise, when successful, it returns $\bot$.

**Transaction instrumentation.** Given a transaction protocol interface $\mathsf{trans}(M)$ and a program $p$ that uses $M$, the function $\mathsf{instrument}$ presented in Figure 4.19 transforms $p$ into a program over $\mathsf{trans}(M)$. (1) The method $\mathsf{init}$ is inserted at the beginning of the transaction. (2) Each method $\mathsf{m}$ in the program is executed by the $\mathsf{lift}$ method. The execution may succeed, in which case the continuation is instrumented, or it may fail, in which case $\mathsf{abort}$ is called. This logic is handled by $\mathsf{instRet}$. (3) Finally, if the program returns $r$, $\mathsf{commit}$ is called. Since that might fail, $\mathsf{instEnd}$ checks the outcome and triggers an $\mathsf{abort}$, returning $\bot$, upon failure; otherwise the value resulting from the execution of the program is returned.

**Transactional objects.** Instrumented transactions need to run on top of some other object that provides life-cycle methods. Given an implementation *pro* of the transaction protocol interface $\mathsf{trans}(M)$, we obtain a transactional object by instrumentation and vertical composition: $\mathsf{trans\text{-}obj}(pro) = \mathsf{instrument} \rhd_O pro$. The protocol object *pro* is correct if the transactional object it produces is strictly

```
     init () :=
$I_1$      $t \leftarrow$ reg.read ();
       if ($t$ mod $2 = 1$) init () else $t$

     lift ($t$, $m'$) :=
       match $m'$with
         | get ($k$) $\Rightarrow$
$G_1$          $v \leftarrow$ map.get ($k$);
$G_2$          $gt \leftarrow$ reg.read ();
           if ($gt = t$)$\langle t, \lceil v \rceil \rangle$ else $\langle t, \bot \rangle$
         | put ($k$, $v$) $\Rightarrow$
           if ($t$ mod $2 = 0$)
$P_1$            $b \leftarrow$ reg.cas ($t$, $t + 1$);
             if ($b$)
$P_2$              _ $\leftarrow$ map.put ($k$, $v$);
               $\langle t + 1, \lceil \mathsf{unit} \rceil \rangle$
             else $\langle t + 1, \bot \rangle$
           else
$P_3$            _ $\leftarrow$ map.put($k$, $v$);
             $\langle t, \lceil \mathsf{unit} \rceil \rangle$

     abort ($t$) :=  unit

     commit ($t$) :=
       if ($t$ mod $2 = 1$)
$C_1$        _ $\leftarrow$ reg.write($t + 1$);
         $\bot$
       else $\bot$
```

FIGURE 4.20. Implementation of the TML protocol tml-imp

serializable for any set of programs. That is, with respect to the specification *spec*, we have trans-obj(*pro*) $\lesssim_{\mathbb{SS}}$ *spec*. The following lemma formalizes a way of decomposing that obligation, for an arbitrary candidate object.

**Lemma 4.6.1.** *A candidate protocol object* $tm = (M_0, \textit{tm-imp}, \textit{spec}_0)$ *is strictly serializable with respect to* map-spec *iff* $(\mathsf{instrument} \triangleright \textit{tm-imp}) \triangleright_S \textit{spec}_0 \lesssim_{\mathbb{L}}$ trans-spec(map-spec).

**4.6.1. Transactional mutex locking.** It remains to see how to implement a transaction protocol object correctly. For the case of the map interface (which yields an implementation of transactional memory), we proved two implementations are strictly serializable: the single global lock [Menon et al., 2008] protocol and the Transactional Mutex Locking (TML) protocol [Dalessandro et al., 2010]. We illustrate the technique with TML.

The TML protocol object tml is a transaction protocol object for the map interface. It provides get and put methods on a map of keys to values. The protocol object uses a pair of a register and a map as its low-level interface. The register represents the global clock for the protocol, and the map stores values for the keys. We say that a transaction is a writer if it executes at least one put method and is a reader if it executes no put methods. Writer transactions acquire exclusive access to the map by compare-and-swapping the parity of the global clock from even to odd, and they later release it by changing the parity of the global clock back to even. Reader transactions execute optimistically: they do not acquire any locks and do not block writer transactions.

Figure 4.20 represents the implementation of the TML protocol object. In the recursive init method, the transaction reads the global clock ($I_1$) in a loop until it is *even* (i.e., there is no concurrent writer transaction). The transaction keeps the final time that it reads in init as its transaction-local time $t$.

To lift the get method, first the value of the input key is read from the map ($G_1$), and then the global clock is read ($G_2$). If the global time is still equal to the transaction-local time, no writer transaction has been active in the meantime, and the values that the current transaction has read are still valid. Hence, the transaction returns the read value. Otherwise, the values that the transaction has speculatively read may have changed, and it returns failure.

For the put method, it is first checked if the current transaction has already acquired exclusive access to the map, i.e., the transaction-local time is odd. If that is the case, the transaction can access the map directly ($P_3$). Otherwise, the transaction tries to acquire exclusive access by incrementing the global time using a compare-and-swap ($P_1$). If the compare-and-swap is not successful, the global clock has changed since it has been read in the init method. Another writer transaction has acquired exclusive access to the map and may have changed it. Therefore, the values previously read by the current transaction may no longer be valid, and the method fails by returning $\perp$. If the compare-and-swap is successful, then the map is updated with the input key and value ($P_2$). The transaction keeps the fact that it has acquired exclusive access by incrementing its transaction-local time to the next odd value.

TML performs updates only if the transaction already has exclusive access and it is safe to perform the updates. Therefore, the abort method simply returns. The commit method checks the parity of the transaction-local time. If the transaction-local time is odd, the current transaction is a writer and has acquired exclusive access. It releases the exclusive-access right by incrementing the global time to the next even value ($C_1$). Finally, the commit method returns successfully.

Theorem 4.6.2. *The TML protocol object* tml *is strictly serializable with respect to the map interface.*

By Lemma 4.6.1 and the definition of transactional specification, we should prove the following concurrent refinement,

$$(\mathsf{instrument} \rhd \mathsf{tml\text{-}imp}) \rhd_S (\mathsf{reg\text{-}spec} + \mathsf{map\text{-}spec}) \lesssim_{\mathbb{C}} \mathsf{atomic}(\mathsf{ab\text{-}spec}(\mathsf{prog\text{-}spec}(\mathsf{map\text{-}spec})))$$

Intuitively, any step of a user program that is instrumented and vertically composed on tml should be refined by the whole execution of the user program or by not running it at all. To prove this refinement, we use the same proof principle that we saw in

Figure 4.13. The unfixed linearization point of a reader transaction is the method call $G_2$ of its last successful get method call. Otherwise, it is $C_1$ (for a writer transaction).

## 4.7. Transactional Predication

We show an example of composing a linearizable object and a serializable object: transactional predication. Whereas, in Section 4.5.1, we implemented a linearizable map object, which guarantees that individual map methods are executed atomically, here we want to implement a serializable map object, which guarantees the atomicity of arbitrary map transactions, *i.e.*, programs that may involve many map method calls. Transactional predication is a technique to implement a serializable map object given a linearizable map (Section 4.5.1) and a TM, *i.e.*, a serializable object of mutable references (Section 4.6.1). The general idea is that the linearizable map quickly maps keys to references, so that calls on the high-level interface can be transformed into calls on references that are managed by the TM. The TM is used only for the final critical calls, which avoids expensive repeated TM calls but gains TM's composability properties.

Transactional predication may be a simple idea, but formalizing it is challenging because it implements a serializable object using linearizable and serializable objects. A naive and tedious verification approach would be to reason about how arbitrary high-level transactions are reduced to low-level transactions. Instead, we define transactional predication as a composition of implementations, and we reason about it compositionally and algebraically, by equational reasoning about vertical and horizontal object compositions ($\triangleright$, $+$).

**4.7.1. Transactional Map.** We saw the structure of the predicated set in Figure 4.4. The predicated map is similar except that instead of a Boolean, a location stores the value of the key or $\perp$ if the key is removed. The predicated map implements the high-level map interface (from Section 4.3) from keys Key to values Val using the sum $\mathsf{RefMap} + \mathsf{TM}$ of the two low-level interfaces RefMap and TM.

The interface TM is the map interface (that we saw in Section 4.3) instantiated with Ref as the keys and Val as values. The references are abstract locations (users may not inspect them, and they are typically meant to be references to mutable memory cells). This interface is implemented by a serializable object such as TML (Section 4.6.1), *i.e.*, programs consisting of get and put calls can be executed atomically.

**4.7.2. Locator.** The interface RefMap is a map of keys Key to references Ref, with only one method $\mathsf{lookup} : \mathsf{Key} \to \mathsf{Ref}$. It is implemented by the locator shown in Figure 4.21, atop a low-level map which provides methods $\mathsf{get}(k)$ and $\mathsf{putIfAbsent}(k, v)$. The locator object is a "lazy" map from keys to mutable references (locations) managed by the underlying transactional memory. When a key is looked up for the first time, a new reference will be allocated and returned; when the same key is looked up again, the same reference will be returned.

The locator object should appear to behave as a pure function $\phi : \mathsf{Key} \to \mathsf{Ref}$ associating map keys to references. A subtlety is that references will be allocated dynamically, so the function $\phi$ is only determined at runtime. In order to formally relate $\phi$ to the locator object, we convert $\phi$ into a sequential specification $\mathsf{ref\text{-}map\text{-}spec}_\phi$

```
locator (lookup($k$)) :=
  $o \leftarrow$ get ($k$);
  match $o$ with
  | $\lceil p \rceil \Rightarrow p$
  | $\bot \Rightarrow$
    $p \leftarrow$ newRef;
    $p \leftarrow$ putIfAbsent ($k, p$);
    $p$
```

FIGURE 4.21. Locator implementation

```
pred-map ($m$) :=
  match $m$ with
  | get ($k$) $\Rightarrow$
    $r \leftarrow$ RefMap.lookup ($k$);
    $v \leftarrow$ TM.get ($r$);
    $v$
  | put ($k, v$) $\Rightarrow$
    $r \leftarrow$ RefMap.lookup ($k$);
    TM.put ($r, v$);
    $\bot$
```

FIGURE 4.22. Core function of transactional predication

parametrized by the pure function $\phi : \mathsf{Key} \to \mathsf{Ref}$, whose transitions associate $\mathsf{lookup}(k)$ calls to return values $\phi(k)$. Since the return value is entirely determined by the method call, the state type is trivially a singleton. The correctness theorem for a locator does not exactly match the general definition of linearizability, because the abstract object $\mathsf{ref\text{-}map\text{-}spec}_\phi$ now depends on the trace of the concrete object $\mathsf{locator}$. For any history $h$ of the object $\mathsf{locator}$ ($h \in \mathsf{beh}(\mathsf{locator})$), there exists a function $\phi : \mathsf{Key} \to \mathsf{Ref}$ such that $h$ is a history of the specification $\mathsf{ref\text{-}map\text{-}spec}_\phi$, i.e., $\mathsf{beh}(\mathsf{locator}) \subseteq \cup_\phi\ \mathsf{beh}(\mathsf{ref\text{-}map\text{-}spec}_\phi)$.

We elide the details of the application of that theorem to simplify the presentation of this proof. The rest of the refinement proof in the remainder of this section is after refining the locator to its specification. Thus, that proof will be applied to the history $h$ from which we obtained the function $\phi$.

Given a function $\phi$, we can also define another implementation $\mathsf{pure\text{-}map}_\phi = (\mathsf{lookup}(k) \mapsto \phi(k))$ of the specification $\mathsf{ref\text{-}map\text{-}spec}_\phi$, by directly using $\phi$ to answer $\mathsf{lookup}$ calls. This alternative representation of $\mathsf{ref\text{-}map\text{-}spec}_\phi$ is useful in equational proofs since it abstracts away the "lazy" implementation. The low-level interface of $\mathsf{pure\text{-}map}_\phi$ is $\mathsf{Empty}$ as the object simply returns the value of $\phi$ for $k$ without making any low-level calls. The specification $\mathsf{ref\text{-}map\text{-}spec}_\phi$ and the implementation $\mathsf{pure\text{-}map}_\phi$ are related by the following equality (which we will use in later proofs). Let us write $obj_1 =_{\mathbb{C}} obj_2$ when $obj_1 \lesssim_{\mathbb{C}} obj_2 \wedge obj_2 \lesssim_{\mathbb{C}} obj_1$.

**Lemma 4.7.1.** *For all $\phi$ and $s$,*

$$\mathsf{id} \rhd_S (\mathsf{ref\text{-}map\text{-}spec}_\phi + s) \;=_{\mathbb{C}}\; (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \rhd \mathsf{empty_L} \rhd_S s$$

The key idea is that a program that makes calls to the interface $\mathsf{RefMap} + M$ intuitively has the same behavior as the program with interface $M$ obtained by interpreting away the $\mathsf{RefMap}$ calls using $\phi$. The $\mathsf{pure\text{-}map}_\phi$ object does not use a low-level interface, so $\mathsf{pure\text{-}map}_\phi + \mathsf{id}$ can be composed on top of $\mathsf{empty_L}$ to filter calls on its left low-level interface. The resulting programs call methods only on the interface $M$ (of $s$).

**4.7.3. Predication.** The implementation of predicated map $\mathsf{pred\text{-}map}$ is shown in Figure 4.22. The function $\mathsf{pred\text{-}map}$ interprets a map method $m$ that is parametrized by the key $k$. It first looks up the reference $r$ associated with the key $k$ by a $\mathsf{lookup}$ in the $\mathsf{RefMap}$ interface. Then, it accesses the value stored in the reference via $\mathsf{get}$ or $\mathsf{put}$ in the $\mathsf{TM}$ interface. The implementation $\mathsf{pred\text{-}map}$ is used later to instrument transactions over the exposed map interface.

The predicated map reduces calls on its map interface to calls on another map. Thus, it is straightforward that the object with the implementation $\mathsf{pred\text{-}map}$ on the low-level specification $\mathsf{ref\text{-}map\text{-}spec}_\phi + \mathsf{map\text{-}spec}$ *sequentially* simulates the high-level specification $\mathsf{map\text{-}spec}$.

**Lemma 4.7.2.**

$$\mathsf{interp\text{-}as\text{-}spec}\,(\mathsf{pred\text{-}map} \rhd_S (\mathsf{ref\text{-}map\text{-}spec}_\phi + \mathsf{map\text{-}spec})) \lesssim_{\mathbb{SP}} \mathsf{map\text{-}spec}$$

**4.7.4. Instrumentation.** To instrument a transaction $p$, we first compose it on top of the $\mathsf{pred\text{-}map}$ implementation to obtain a program over the sum interface $\mathsf{RefMap} + \mathsf{TM}$. Then, we instrument the program with the function $\mathsf{instrumentR}$, a variant of $\mathsf{instrument}$, defined in Figure 4.19, where the main change is that $\mathsf{instrumentR}$ only lifts method calls $m$ that belong to the $\mathsf{TM}$ interface, while forwarding $\mathsf{RefMap}$ calls without modification. We then compose it on top of $\mathsf{id} + \textit{tm-imp}$ that leaves the left calls unchanged but interprets the right $\mathsf{TM}$ calls by the TM implementation *tm-imp*.

$$\mathsf{p\text{-}map\text{-}imp} = \mathrm{transF}(\mathsf{pred\text{-}map}) \rhd \mathsf{instrumentR} \rhd (\mathsf{id} + \textit{tm-imp})$$

The operator transF, defined below, transforms an implementation $f$ of an interface $M$ into an implementation $\mathrm{transF}(f)$ of the interface $\mathsf{?Prog}(M)$. When called with a transaction $p$, $\mathrm{transF}(f)$ makes a single call with the instrumented transaction $p \rhd f$.

$$\mathrm{transF} : \mathsf{Impl}\, M\, N \to \mathsf{Impl}\,(\mathsf{?Prog}(M))\,(\mathsf{?Prog}(N))$$
$$\mathrm{transF}(f) = (?\mathsf{exec}(p) \mapsto (v \leftarrow ?\mathsf{exec}(p \rhd f); v))$$

The following lemma states the relation between $\mathsf{instrument}$ and $\mathsf{instrumentR}$.

**Lemma 4.7.3.** *For all $p$, $f$ and $g$,*

$$\mathrm{transF}(f) \rhd \mathsf{instrumentR} \rhd (g + \mathsf{id}) \rhd \mathsf{empty_L} = \mathrm{transF}(f \rhd (g + \mathsf{id}) \rhd \mathsf{empty_L}) \rhd \mathsf{instrument}.$$

The function $\mathsf{instrumentR}$ is applied to programs on a sum interface $M + \mathsf{TM}$, and it wraps calls on the right but does not alter calls on the left. Therefore, the methods on the left can be interpreted using an implementation $g$ before the instrumentation.

Further, since the composition on $\mathsf{empty}_\mathsf{L}$ removes the calls on the left, the function $\mathsf{instrument}$ can be applied instead of $\mathsf{instrumentR}$.

**4.7.5. Strict serializability.** The final transactional object is

$$\mathsf{p\text{-}map} = \mathsf{p\text{-}map\text{-}imp} \triangleright_S (\mathsf{ref\text{-}map\text{-}spec}_\phi + \mathit{spec}_0)$$

where $\mathit{spec}_0$ is the specification of the low-level interface of the TM protocol. The goal is to prove the serializability of this object with respect to the map specification $\mathsf{map\text{-}spec}$.

Before the proof, we give a helper lemma. It states that interpreting a program $p$ by an implementation $f$ and then interpreting it under the transactional specification of $s$ is the same as interpreting it directly as a transaction under the interpreted sequential specification of $f$.

**Lemma 4.7.4.** *For all $f$ and $s$,* $\mathrm{transF}(f) \triangleright_S \mathsf{trans\text{-}spec}(s) \;\lesssim_{\mathbb{SS}}\; \mathsf{interp\text{-}as\text{-}spec}(f \triangleright_S s)$.

The proof of serializability is equational, *i.e.*, by successive refinements. This proof style allows us to package the correctness conditions of individual components as (in)equations that are then chained together in the final proof in well-delimited rewriting steps interleaved with some administrative simplifications or factorizations provided by the equational theory of interaction trees (Proposition 4.4.12). The high-level idea is that the pure calls to $\mathsf{ref\text{-}map\text{-}spec}_\phi$ on the left can be filtered (step 2), and programs that are instrumented on the right by $\mathsf{instrumentR}$ can be transformed to flat programs that are instrumented by $\mathsf{instrument}$ (step 5). Having isolated the TM implementation *tm-imp*, its serializability guarantees can be applied (step 7). Finally, the correctness theorem of the core implementation exposes the map sequential specification (step 11). Those key steps rely respectively on properties of the low-level specification $\mathsf{ref\text{-}map\text{-}spec}$, the instrumentation functions $\mathsf{instrument}$ and $\mathsf{instrumentR}$, the serializable object *tm*, and the core implementation $\mathsf{pred\text{-}map}$ laid out above.

Theorem 4.7.5 (Correctness of transactional predication).

$$\mathsf{p\text{-}map} \lesssim_{\mathbb{SS}} \mathsf{map\text{-}spec}$$

*Proof.*

(1) $\mathsf{p\text{-}map} = \mathsf{p\text{-}map\text{-}imp} \triangleright_S (\mathsf{ref\text{-}map\text{-}spec}_\phi + \mathit{spec}_0)$
(2) By Lemma 4.7.1 (after expanding with $\mathsf{p\text{-}map\text{-}imp} = \mathsf{p\text{-}map\text{-}imp} \triangleright \mathsf{id}$):
$\lesssim_{\mathbb{C}} \mathsf{p\text{-}map\text{-}imp} \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright \mathsf{empty}_\mathsf{L} \triangleright_S \mathit{spec}_0$
(3) Unfold $\mathsf{p\text{-}map\text{-}imp}$:
$\mathrm{transF}(\mathsf{pred\text{-}map}) \triangleright \mathsf{instrumentR} \triangleright (\mathsf{id} + \mathit{tm\text{-}imp}) \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright \mathsf{empty}_\mathsf{L} \triangleright_S \mathit{spec}_0$
(4) Commute vertical compositions based on properties of $\triangleright$ and $+$ (Proposition 4.4.12):
$= \mathrm{transF}(\mathsf{pred\text{-}map}) \triangleright \mathsf{instrumentR} \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright (\mathsf{id} + \mathit{tm\text{-}imp}) \triangleright \mathsf{empty}_\mathsf{L} \triangleright_S \mathit{spec}_0$
$= \mathrm{transF}(\mathsf{pred\text{-}map}) \triangleright \mathsf{instrumentR} \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright \mathsf{empty}_\mathsf{L} \triangleright \mathit{tm\text{-}imp} \triangleright_S \mathit{spec}_0$
(5) By Lemma 4.7.3:
$= \mathrm{transF}(\mathsf{pred\text{-}map} \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright \mathsf{empty}_\mathsf{L}) \triangleright \mathsf{instrument} \triangleright \mathit{tm\text{-}imp} \triangleright_S \mathit{spec}_0$
(6) Abbreviate $m = \mathsf{pred\text{-}map} \triangleright (\mathsf{pure\text{-}map}_\phi + \mathsf{id}) \triangleright \mathsf{empty}_\mathsf{L}$:
$= \mathrm{transF}(m) \triangleright \mathsf{instrument} \triangleright \mathit{tm\text{-}imp} \triangleright_S \mathit{spec}_0$
(7) By serializability of *tm* (Lemma 4.6.1):
$\lesssim_{\mathbb{C}} \mathrm{transF}(m) \triangleright_O \mathsf{atomic}(\mathsf{trans\text{-}spec}(\mathsf{map\text{-}spec}))$

(8) By definition of $\mathsf{atomic}$ and simplification:
$= \text{transF}(m) \triangleright_O (\mathsf{id} \triangleright_S \mathsf{trans\text{-}spec}(\mathsf{map\text{-}spec}))$
$= \text{transF}(m) \triangleright_S \mathsf{trans\text{-}spec}(\mathsf{map\text{-}spec})$

(9) By Lemma 4.7.4 and the definitions of linearizability and strict serializability:
$\lesssim_{\mathbb{C}} \mathsf{atomic}(\mathsf{trans\text{-}spec}(\mathsf{interp\text{-}as\text{-}spec}(m \triangleright_S \mathsf{map\text{-}spec})))$

(10) By Lemma 4.7.1:
$\lesssim_{\mathbb{C}} \mathsf{atomic}(\mathsf{trans\text{-}spec}(\mathsf{interp\text{-}as\text{-}spec}(\mathsf{pred\text{-}map} \triangleright_S (\mathsf{ref\text{-}map\text{-}spec}_\phi + \mathsf{map\text{-}spec}))))$

(11) By Lemma 4.7.2 and Proposition 4.4.6:
$\lesssim_{\mathbb{C}} \mathsf{atomic}(\mathsf{trans\text{-}spec}(\mathsf{map\text{-}spec}))$

(12) By the definitions of linearizability and strict serializability:
$\lesssim_{\mathbb{SS}} \mathsf{map\text{-}spec}$ □

## 4.8. Opacity

Our framework models strict serializability in terms of linearizability. Opacity [Guerraoui and Kapalka, 2008], another correctness condition for transactional memories, requires active in addition to completed transactions to observe consistent state. Previous work [Lesani et al., 2012b] defined a transition system (or specification) for opacity that constrains the return values of all calls from active transactions. Here we give a sketch of how we can model opacity in terms of linearizability as well. A protocol object is opaque iff it is linearizable with respect to the opacity specification.

Given a interface $M$, recall that a transaction protocol object is an object implementing the protocol interface $\mathsf{trans}(M)$. Given also a specification *spec* with interface $M$, we previously defined strict serializability with respect to *spec* in terms of linearizability (of the instrumented protocol object) with respect to $\mathsf{trans\text{-}spec}(\mathit{spec})$. The specification $\mathsf{trans\text{-}spec}(\mathit{spec})$ specifies the observable behavior of completed transactions. Opacity will be defined similarly as linearizability (of the protocol object—without instrumentation) with respect to a specification $\mathsf{opacity\text{-}spec}(\mathit{spec})$. The specification $\mathsf{opacity\text{-}spec}(\mathit{spec})$ specifies the behavior of individual transactional methods in active transactions.

The opacity specification $\mathsf{opacity\text{-}spec}(\mathit{spec}) = (S_{\text{opacity}}, s_{0,\text{opacity}}, \Delta_{\text{opacity}})$ of type $\mathsf{Spec}\,\mathsf{trans}(M)$, from a specification $\mathit{spec} = (S, s_0, \Delta_{\mathit{spec}}) : \mathsf{Spec}\,M$ consists of the following components. First, a state in $S_{\text{opacity}}$ is a pair of a state in $S$, to be the result of committed operations, and a log of live transactions, which is a mapping $\ell$ from transaction IDs to histories $\ell(t) = \lceil(\mathsf{m}_1, r_1) \ldots (\mathsf{m}_i, r_i)\rceil$. Additionally, $\ell$ may map a transaction ID to $\bot$ to indicate a transaction which failed, but has yet to be aborted explicitly. Second, the initial state is $s_{0,\text{opacity}} = (s_0 \mid \emptyset)$, with an empty log in the second component. Third, the transition relation $\Delta_{\text{opacity}}$ is defined by the inference rules in Figure 4.23, to specify the behavior of the $\mathsf{trans}(M)$ methods $\mathsf{init}()$, $\mathsf{lift}(t, m)$, $\mathsf{commit}(t)$, and $\mathsf{abort}(t)$. These rules are adapted from the opacity automaton in Lesani et al. [2012b] to our present setting. To keep the presentation simple, this specification assumes that the transaction-local state $t$ used by transactional methods consists only of a transaction ID. The method $\mathsf{init}()$ returns a fresh transaction ID, initializing it with an empty history. For a method $\mathsf{lift}(t, m)$, which executes $m$ in transaction $t$, it is successful only if it can produce a consistent result $r$, *i.e.*, such that the history of that transaction is valid starting from the current committed state

$$\textsc{Opacity}\text{-init}\quad \frac{t \notin \ell}{(\mathsf{init}, (s, \ell), t, (s, \ell[t \mapsto \varepsilon])) \in \Delta_{\text{opacity}}}$$

$$\textsc{Opacity}\text{-lift-}\textsc{Ok}\quad \frac{\ell(t) = \lceil h \rceil \qquad s \xrightarrow{h \cdot (m,r)}_{\mathit{spec}} s'}{(\mathsf{lift}(t, m), (s, \ell), (t, \lceil r \rceil), (s', \ell[t \mapsto \lceil h \cdot (m, r) \rceil])) \in \Delta_{\text{opacity}}}$$

$$\textsc{Opacity}\text{-lift-}\textsc{Fail}\quad \frac{t \in \ell}{(\mathsf{lift}(t, m), (s, \ell), (t, \bot), (s, \ell[t \mapsto \bot])) \in \Delta_{\text{opacity}}}$$

$$\textsc{Opacity}\text{-commit-}\textsc{Ok}\quad \frac{\ell(t) = \lceil h \rceil \qquad s \xrightarrow{h}_{\mathit{spec}} s'}{(\mathsf{commit}(t), (s, \ell), \bot, (s', \ell \setminus t)) \in \Delta_{\text{opacity}}}$$

$$\textsc{Opacity}\text{-commit-}\textsc{Fail}\quad \frac{t \in \ell}{(\mathsf{commit}(t), (s, \ell), \lceil t \rceil, (s, \ell)) \in \Delta_{\text{opacity}}}$$

$$\textsc{Opacity}\text{-abort}\quad \frac{t \in \ell}{(\mathsf{abort}(t), (s, \ell), (), (s, \ell \setminus t)) \in \Delta_{\text{opacity}}}$$

$$\textsc{Aux}\text{-}\varepsilon\quad \frac{}{s \xrightarrow{\varepsilon}_{\mathit{spec}} s} \qquad\qquad \textsc{Aux}\text{-}\textsc{Cons}\quad \frac{s \xrightarrow{h}_{\mathit{spec}} s_h \qquad (m, s_h, r, s') \in \Delta_{\mathit{spec}}}{s \xrightarrow{h \cdot (m,r)}_{\mathit{spec}} s'}$$

FIGURE 4.23. Transitions of the opacity specification opacity-spec(*spec*), with auxiliary transition relation $\xrightarrow{h}_{\mathit{spec}}$

$s$. The method commit($t$) must also check the history for validity, as the commited state may have changed since the last method invocation from that transaction. If that succeeds, $t$ is removed from the log $\ell$, the resulting log is denoted as $\ell \setminus t$. The method abort($t$) is called when either lift($t, m$) or commit($t$) fails to remove $t$ from the log (see also Figure 4.19).

We can now define opacity. A transactional protocol object *obj* : Object (trans($M$)) *ensures opacity* with respect to a sequential specification *spec* : Spec $M$ if *obj* is linearizable with respect to opacity-spec(*spec*). We conjecture that this matches the definition in Lesani et al. [2012b], which has been related to the original formulation of opacity [Guerraoui and Kapalka, 2008]. Notably, opacity should subsume strict serializability: if *obj* ensures opacity with respect to *spec*, then *obj* is strictly serializable with respect to *spec*.

## 4.9. Conclusion

We have presented the first case study in formal verification that shows how to compose *verified transactional objects* whose implementations blend classic concurrent data structures with transactions. Moreover, while one might expect new complications in the reasoning framework to support this marriage, we demonstrated how it can be accomplished in a simple modular framework. Our notion of concurrent-object correctness is the classic one of *linearizability* with respect to sequential specifications, but applied in a higher-order logic with functional programs that manipulate higher-order structures. Our encoding of transactions relies on methods that take complete transactions as inputs and rewrite them syntactically to add synchronization, permitting us to state and prove strict serializability in terms of linearizability.

CHAPTER 5

# Related Work

## 5.1. Interaction Trees

There is a wealth of related work investigating structures similar to interaction trees, with numerous perspectives on interaction, effects, nontermination, compositionality, and formal reasoning.

The problem of accommodating effectful programming in purely functional settings is an old one, and a variety of approaches have been explored, monads and algebraic effects being two of the most prominent. We concentrate on these two techniques, beginning with general background and then focusing on the closest related work.

**5.1.1. Monads, Monad Transformers, and Free Monads.** Moggi's seminal paper [1989] introduced monads as one way to give meaning to imperative features in purely functional programs. Monads were subsequently popularized by Wadler [1992] and Peyton Jones and Wadler [1993] and have had huge impact, especially in Haskell. However, it was soon recognized that composing monads to combine multiple effects was not straightforward. Monad transformers [Moggi, 1990] are one way to obtain more compositionality; for example, Liang et al. [1995] showed how they can be used to build interpreters in a modular way. The `interp_state` function from Section 2.2 is an example of building an event interpreter using a monad transformer in this style. In our case, not all monads are suitable targets for interpretation: we require them to support recursion in the sense that their Kleisli categories are iterative [Adámek et al., 2010, Goncharov et al., 2016]. Correspondingly, not all monad transformers can therefore be used to build interpreters.

Datatypes à la Carte [Swierstra, 2008] showed how to use a *free monad* to define monad instances modularly. Transporting his definition to our setting, we would obtain the following:

```
CoInductive Free (E : Type → Type) (R:Type) :=
| Ret : R → Free E R
| Vis : E (Free E R) → Free E R.
```

This version of the `Vis` constructor directly applies the functor `E` to the coinductively defined type `Free E R` itself. However, this type violates the strict positivity condition enforced by Coq: certain choices of `E` would allow one to construct an infinite loop.

Subsequent work [Apfelmus, 2010, Kiselyov et al., 2013, Kiselyov and Ishii, 2015] showed how free monads can be made more liberal by exposing the continuation in the `Vis` constructor. The resulting "freer" monad (called `FFree` in their work) is essentially identical to our ITrees—the difference being that, because they work in Haskell, which admits nontermination by default, it needs no `Tau` constructor.

When considered up to strong bisimulation, ITrees form the free *completely iterative monad* [Aczel et al., 2003] with respect to a functor of the form `fun X ⇒ F X + X`, where the second component corresponds to `Tau` nodes. Quotiented by weak bisimulation, ITrees define a free *pointed monad* [Uustalu and Veltri, 2017]. There is a rich literature on the theory of iteration [Bloom and Ésik, 1993, Milius, 2005, Goncharov et al., 2017], studying the properties of operators such as `mrec` in yet more general category-theoretic settings. The ITrees library makes such results concretely applicable to formally verified systems.

ITrees are a form of resumptions, which originated from concurrency theory [Milner, 1975]. More precisely, ITrees can be obtained by applying a coinductive resumption monad transformer [Piròg and Gibbons, 2014, Cenciarelli and Moggi, 1993] to the delay monad of Capretta [2005]. Other variations of the resumption monad transformer have been used to model effectful and concurrent programs [Nakata and Uustalu, 2010, Goncharov and Schröder, 2011]. In particular, Nakata and Uustalu [2010] also used coinductive resumptions in Coq to define the semantics of Imp augmented with input-output operations. They also defined termination-sensitive weak bisimilarity ("equivalence up to taus") using mixed induction-coinduction. However, their semantics was defined as an explicitly coinductive *relation*, with judicious introductions of `Tau`. Their Coq development was specialized to Imp's global state and was not intended to be used as a general-purpose library. In contrast, our semantics are functional (denotational, definitional) *interpreters*, and we encapsulate nontermination (`Tau` is an internal implementation detail) using recursion operators that are compatible with Coq's extraction mechanisms.

**5.1.2. Algebraic Effects and Handlers.** Algebraic effects are a formalism for expressing the semantics of effectful computations based on the insight by Plotkin and Power that many computational effects are naturally described by algebraic theories [2001, 2002, 2003]. The idea is to define the semantics of effects equationally, with respect to the term model generated by operations $\mathtt{op} \in \Sigma$, the signature of an algebra. When combined with the notion of an *effect handler*, an idea originally introduced by Cartwright and Felleisen [1994] and later investigated by Plotkin and Pretnar [2013], algebraic effects generalize to more complex control effects yet still justify equational reasoning. The monoidal structure of algebraic effects is well known [Hyland et al., 2006]; more recent work has studied the relationship between monad transformers and modular algebraic effects [Schrijvers et al., 2016].

In our setting, an event interface such as `stateE` (Figure 2.12) defines an effect signature $\Sigma$, and its constructors `Get` and `Put s` define the operations. Plotkin and Pretnar used the notation $\mathtt{op}(x\!:\!X.\,M)$, corresponding to the ITrees `Vis op (fun x:X ⇒ M)` construct, and called it "operation application". They axiomatized the intended semantics of effects via equations on operation applications—for example, the fact that two `get` operations may be collapsed into one was expressed by the equation $\mathtt{get}(x : S.\,\mathtt{get}(y : S.\,kxy)) = \mathtt{get}(x : S.\,kxx)$. For ITrees, we prove such equations relative to an interpretation of the events, as in Section 2.2.1.

The handlers of algebraic effects specify the data needed to construct an interpretation of the effect; they have the form $\mathtt{handler}\{\mathtt{return}\ x \mapsto f(x), (\mathtt{op}(y;\kappa) \mapsto$

$h(y, \kappa))_{\mathsf{op} \in \Sigma}\}$. In terms of our notation, the `return` component of the handler specifies the `Ret` case of an interpreter, and the sum over operation interpretations is written using a dependent type. Here, $h$ corresponds to the most general elimination form for the ITree `Vis` constructor, which is a function of type `∀ X, E X → (X → itree E R) → M R` for `M` an iterative monad. However, Coq prevents us from creating a general-purpose interpeter parameterized by such a type—it needs to see the definition of the handler's body to verify the syntactic guardedness conditions.

In a language such as Eff [Bauer and Pretnar, 2015], which supports algebraic effects natively, the operational semantics plumbs together the continuations with the appropriate handlers, scoping them according to the dynamic semantics of the language. In our case, we must explicitly invoke functions like `interp_state` as needed, possibly after massaging the structure of events so that they have the right form.

Johann et al. [2010] studied the contextual equivalences induced by interpretations of standard effects. Most saliently, their paper developed its theory in terms of observations of "computation trees," which are "incompletely known" ITrees—they are inductively defined, and hence finite, but may also include $\bot$ leaves that denote (potential) divergence. Johann et al. showed how to endow the set of computation trees with a CPO structure based on approximation ($\bot \sqsubseteq t$ for any tree $t$) and use that notion to study contextual equivalences induced by various interpreters. The techniques proposed there should be adaptable to our setting: instead of working with observational partial orders, we might choose to work more directly with the ITree structures themselves.

**5.1.3. Effects in Type Theory.** Most of the work discussed above was done either in the context of programming languages with support for general recursion or in a theoretical "pen and paper" setting, rendering these approaches fundamentally different to the ITree library which is formalized in a total language. Work more closely related to ITrees is that undertaken in the context of dependent type theory.

The earliest work on mixing effects with type theory was done by Hancock and Setzer [2000], followed by Hancock's dissertation [Hancock, 2000]. This line of work, inspired by monads and especially Haskell's IO monad, showed how to encode such constructs in Martin-Löf type theory. Those theories, in contrast to ITrees, do not allow silent steps of computation, instead integrating guarded or sized coinductive types as part of a strong discipline of total functional programming. The benefit of this is that strong bisimilarity is the only meaningful notion of equivalence; the drawback is that they cannot handle general recursion. Later work on object encodings [Setzer, 2006] did consider recursive computations, though it did not study their equational theory or the general case of implementing interpreters within the type theory, as we have done. More recently, Abel et al. [2017] have demonstrated the applicability of these ideas in Agda. Although their paper includes a proof of the correctness of a stack object (among other examples), they do not focus on the general equational theory of such computations.

As mentioned previously, Capretta proposed using the "delay monad" to encode general recursion in a type theory, as we do here, though his paper used strong bisimulation as the notion of equivalence. The delay monad can be seen as either an

ITree without the `Vis` constructor or, isomorphically, an ITree of type `itree emptyE R`. The main theoretical contribution of that paper was showing that the monad laws hold and that the resulting system is expressive enough to be Turing complete. Subsequent work explored the use of the delay monad for defining operational semantics [Danielsson, 2012] and studied how to use quotient types [Chapman et al., 2015] or higher inductive types [Altenkirch et al., 2017] to define equivalence up to `Tau`, which we take as the basis for most of our equational theory. Because we are working in Coq, which does not have quotient or higher inductive types, we must explicitly use setoid rewriting, requiring us to prove that all morphisms respect the appropriate equivalences.

McBride [2015], building on Hancock's earlier work, used what he called the "general monad" to implement effects in Agda. His monad variant is defined inductively as shown below.

```
Inductive General (S:Set) (T : S → Set) (X : Set) : Set :=
| RetG (x : X)
| VisG (s:S) (k : T s → General S T X).
```

Its interface replaces our single `E : Type → Type` parameter with `S : Type` and a type family `S → Type` to calculate the result type of the event. McBride proposed encoding recursion as an (uninterpreted) effect, as we present in Section 2.3. In particular, he shows how to give a semantics to recursion using first a "fuel"-based (a.k.a. step-indexed) model and then by translation into Capretta's delay monad. The latter can be seen as a version of our `interp_mrec`, but one in which all of the effects must be handled. Our coinductively defined interaction trees also support a general fixpoint combinator directly, which is impossible for the `General` monad.

The FreeSpec Coq library, implemented by Letan et al. [2018], uses a "program monad" to model components of complex computing systems. The program monad is essentially an inductive version of `itree`[9] (without `Tau`). What we call "events," the FreeSpec project calls "interfaces." The FreeSpec project is primarily concerned with modeling first-order, low-level devices for which general recursion is probably not needed. Its library offers various composition operators, including a form of concurrent composition, and it includes a specification logic that helps prove (and automate proofs of) properties about the systems being modeled. However, due to FreeSpec's use of the inductive definition, such systems must be structured as acyclic graphs. Nevertheless, FreeSpec doesn't eschew coinduction altogether—as we explain below, it, like CompCert, defines the environment in which the program runs coinductively. FreeSpec's handlers are thus capable of expressing diverging computations, but it does not support the equational reasoning principles that we propose.

Arrows [Hughes, 2000] are another abstraction for effectful computations inspired directly by category theory, generalizing monads. Paterson [2001] extends that abstraction with a loop operator inspired by traced monoidal categories, a generalization of iterative categories.

**5.1.4. Composition with the Environment.** An idea that is found in several of the works discussed above is the need to characterize properties of the program's

[9]The original version of FreeSpec also included a `bind` constructor, but, following our ITrees development, it was removed in favor of defining bind.

environment. Recall the `kill9` program from earlier, which halts when the input is `9` but continues otherwise. One might wish to prove that, if the environment never supplies the input `9`, the program goes on forever. In a more realistic setting like CompCert, one might wish to make assertions about externally supplied functions, such as OS calls, `malloc` or `memcpy`, or to reason about the accumulated output on some channel such as the terminal.

The behavior of the environment is, in a sense, *dual* to the behavior of the program. CompCert, for example, formulates the environment as a coinductively defined "world," whose definition is (a richer version of) the following:

```
CoInductive world : Type :=
  World (io : string → list eventval → option (eventval * world))
```

Here the `string` and list of `eventval`s are the *outputs* of the event (they are provided by the program), and the result (if any) is a returned value and a new world. The environment's state is captured in the closure of the `io` function. Transliterating this type to our setting we arrive at:

```
CoInductive world E : Type := World (io : ∀ {A:Type}, E A → option (A *
    world E)).
```

Letan et al. [2018] use a definition very close to this (without the option) to define a notion of "semantics" for the program monad. Given such a definition, one can define a world that satisfies a certain property (for example, one that never produces `9` as an answer) and use it to constrain the inputs given to the program, by "running" the program under consideration in the given world. CompCert defines "running" via a predicate called `possible_trace` that matches the answers provided by the `io` function to the events of the program trace.

The CertiKOS project [Gu et al., 2015a, 2018a] takes the idea of composing a program with its environment even further. Their Concurrent Certified Abstraction Layers (CCAL) framework also uses a trace-based formulation of semantics. In their context, traces are called *logs* and (concurrent) components are given semantics in terms of sets of traces. Each component (*e.g.*, a thread) can be separately given a specification in terms of its interface to (valid) external environments, which encode information about the scheduler and assumptions about other components in the context. A layer interface can "focus" on subsets of its concurrently executing components; when it is focused on a single, sequential thread, the interface is a deterministic function from environment interactions (as represented by the log) to its next action. The parallel layer composition operation links two compatible layers by "running" them together (as above) according to the schedule (inputs to one component can be provided by outputs of the other). In this case, one thread's behaviors influence another thread's environment. They formulate such interactions in terms of concepts from game semantics, which gives rise to a notion of refinements between layer specifications. Layers have the symmetric monoidal structure familiar from algebraic effects.

We conjecture that the sequential behavior of the CCAL system could be expressed in terms of ITrees and that the concurrent composition operations of the framework could be defined on top of that. Our KTree combinators already offer a rich notion of composition, including general, mutually recursive linking, which is similar to that offered by CCAL. Moreover, we can define similar "running" operations directly on ITrees, rather than on traces, coordinating multiple ITrees via an executable scheduler. This means that, besides proving properties of the resulting system, we can extract executable test cases [Koh et al., 2019].

**5.1.5. Formal Semantics.** There are a plethora of techniques used to describe the semantics of programming languages within proof assistants. In evaluating these techniques, we need to consider both the simplicity of the definitions and their robustness to language extensions. The former is important because complex models are difficult to reason about, while the latter is important because seemingly small changes sometimes cascade through a language, invalidating previous work.

Denotational semantics translate the object language (*e.g.*, Imp or Asm) into the meta-language (*e.g.*, Gallina), seeking to leverage the existing power of the proof assistant. Chlipala [2007] also uses denotational semantics to verify a compiler, but in a simpler setting with a normalizing source language, and with models individually tailored to the intermediate languages. In contrast, ITrees serve as a common foundation for both semantics in our case-study compiler, and an equational theory enabling the verification of a termination-sensitive theorem. As we saw in Section 3.1, impure features such as nontermination can make this difficult, as proof assistants often include only a total function space. One way to circumvent this limitation is via a "fuel"-based semantics, where computations are approximated to some finite amount of unwinding. Owens et al. [2016] use this approach to develop functional big-step semantics. To reason about nonterminating executions, Owens et al. [2016] leverages the classical nature of the HOL logic to assert that, if no amount of fuel is sufficient for termination, then the computation diverges. They further show how oracle semantics [Hobor, 2008] can be used to enrich this language with both IO and nondeterminism. In practice, the approach is quite similar to ITrees, except that we can omit the fuel and instead directly construct the infinite computation tree. With ITrees, events encode oracle queries and `Tau`s represent internal steps, which may lead to divergence. However, as we showed in Chapter 3, users of ITrees are mostly insulated from `Tau`s when using the combinators from Section 2.3.

The approach of Owens et al. [2016] is reminiscent of traditional step indexing [Ahmed, 2004], in which the meaning of a program is described by a set of increasingly accurate approximations.

Formalizations of more classic domain-theoretic denotational models exist [Benton et al., 2009, 2010]. Unfortunately, the learning curve for this style of denotational semantics was widely considered to be quite steep. The complexity of domain-theoretic models prompted exploring more operational approaches to formalizing semantics [Plotkin, 2004b,a]. Big-step operational semantics share a similar flavor to denotational semantics as they both connect terms directly to their meaning. Unfortunately, interpreting big-step semantics inductively prevents them from representing divergent computations. Some works [Delaware et al., 2013, Chlipala, 2010] avoid the issue of nontermination entirely, ascribing semantics only to terminating executions. Charguéraud [2013] provides a technique for avoiding the problem by duplicating the semantics both inductively and coinductively, arguing that such duplication can be automated and therefore should not be overly burdensome. The functional style of this "pretty big step" semantics is quite similar to functional denotational semantics, and thus bears a resemblance to ITrees. ITrees avoid the need to duplicate the semantics by giving a data representation rather than a propositional representation.

Leroy and Grall [2009] give an in-depth discussion relating inductive and coinductive semantic styles, providing an inductive judgment for "terminates in a value (and a trace)" and a coinductive judgment for "diverges (with an infinite trace)." Relating these semantics can be difficult, and the proofs sometimes rely on classical logic.

**5.1.6. Alternative Interpretations.** Those same tree structures also serve as general representations of recursive data types, encoding the constructors as a suitable event type (`E` in `itree E`). Such types are also called W-types, M-types, or containers [Abbott et al., 2005, Altenkirch et al., 2015].

Game semantics [Abramsky and McCusker, 1999, Abramsky et al., 1997, Hyland, 1997] is another approach to modelling interactive systems. In interaction trees, the event type `E` associates a set of responses to each event. In game semantics, such a structure is called an arena, viewing events and responses as moves in a game. Michelbrink [2006] generalizes the trees of Hancock and Setzer [2000] to construct a simple game semantic model. The CertiKOS project [Gu et al., 2016] is also heavily influenced by game semantics [Gu et al., 2015a, 2018a, Vale et al., 2022, Koenig and Shao, 2020].

## 5.2. Verified Transactional Objects

**5.2.1. Composing Concurrent Operations.** Several previous works present techniques to compose multiple calls on concurrent data structures into atomic operations. Transactional boosting [Herlihy and Koskinen, 2008] benefits from commutativity specifications to allow concurrent execution of commutative calls and prevent concurrent execution of noncommuting calls by acquiring the same lock. Later works presented optimistic variants [Hassan et al., 2014, Dickerson et al., 2019].

Guerraoui [1995] introduced o-atomicity, a property of specifications of atomicity that allows multiple objects with different serialization orders to be composed in the same transaction, presenting sketches of both pessimistic and optimistic implementations that satisfy o-atomicity. Similarly, Reversible Atomic Objects (RAO) [Antonopoulos et al., 2016] are an implementation technique for such compositions. In contrast to the two works above, this thesis presents general and composable definitions to capture different specifications and implementation techniques modularly, plus proof principles to verify correctness. ROA can be captured in our framework as a composition instance, and the RAO refinement proof can be captured as linearizability of the resulting object with respect to the specification of the high-level interface. Finally, in contrast to RAO, our framework supports recursive method calls.

We saw transactional predication [Bronson et al., 2010] in this thesis. Similarly, transactional data structures [Spiegelman et al., 2016, Assa et al., 2020] and transactional software objects [Herman et al., 2016] use TM judiciously and further benefit from specific data-structure semantics and organization to improve efficiency. Follow-up work presents lock-free variants [Elizarov et al., 2019]. In Foresight [Golan-Gueta et al., 2013], the client declares an overapproximation of the set of methods that they foresee to be called. To maintains a partial order for the composed operations, the library may temporarily block a method call that does not commute with the possible future calls by the environment. A few other works [LaBorde et al., 2019, Zhang et al., 2018, Lamar et al., 2020] present custom synchronization mechanisms to provide transactional implementations of specific data structures such as linked lists and vectors. However, the above do not provide proof techniques and formal atomicity guarantees.

**5.2.2. Testing and Verification of Composed Operations.** Colt [Shacham et al., 2011] and ICFinder [Liu et al., 2013] test atomicity of, and Snowflake [Lesani et al., 2014] automatically verifies, composed methods that extend the interface of an already linearizable data structure [Lea, 2000]. Our framework includes tactics that can automatically verify a strict superset of the above use cases. Further, it supports the definition of a more diverse set of objects including composition of multiple objects and transaction protocols, and it provides general proof techniques to verify them. Flint [Liu et al., 2014] fixes nonatomic composed methods. It infers a specification from the method itself and applies heuristics to synthesize a concurrent implementation. In contrast to Flint's repair of composed methods, our framework supports their formal definition and mechanized verification.

TxC-ADT [Peterson and Dechev, 2017] generates happens-before graphs and applies model checking to check the consistency of transactional data structures. By contrast, our framework presents proof techniques to verify these data structures mechanically in a proof assistant.

**5.2.3. Testing and Verification of Atomicity.** Filipović et al. [2010] characterized linearizability as observational refinement. Attiya et al. [2017] characterized the TM correctness conditions TMS [Doherty et al., 2013] and opacity [Guerraoui and Kapalka, 2008] as observational refinement. Thus, the notions of simulation and refinement [Abadi and Lamport, 1991, Lynch and Vaandrager, 1995] have been applied to verify atomicity [Schellhorn et al., 2012, Bouajjani et al., 2017, Lesani et al., 2012a, Jagannathan et al., 2014, Hawblitzel et al., 2015b, Turon et al., 2013, Emmi and Enea, 2019, Kragl et al., 2020]. However, they do not consider verification of composed operations. A related project [Armstrong et al., 2017] first proves the atomicity of individual operations (read, write, commit) of the transaction protocol and then applies simulation to prove serializability on top of that. In contrast to our formalism, it does not modularly state serializability in terms of linearizability and uses a standalone definition of opacity.

A related project [Cerone et al., 2014, Murawski and Tzevelekos, 2019] presents multiple dedicated definitions of linearizability for objects (or libraries) that are implemented in terms of other objects. Our framework shows that a unified definition of linearizability, composition combinators, and proof technique can be the foundation of different instantiations including serializability. A few projects [Raad et al., 2019, Batty et al., 2013] consider modular verification of data structures on weak memory [Adve and Gharachorloo, 1996, Sewell et al., 2010]. However, they do not consider transactions or the relation of linearizability and serializability. Further, the above projects did not consider the composition of transactional memory and concurrent data structures.

**5.2.4. Modular Systems.** Objects and sequential specifications are similar to the modules and interfaces of certified abstraction layers [Gu et al., 2015b], which were introduced in a sequential and deterministic context. Determinism enables proofs by downward simulation, and the sequential nature of modules allows them to support a flexible form of horizontal composition. In contrast, we leverage linearizable objects [Herlihy and Wing, 1990, Filipović et al., 2010, Gotsman and Yang, 2012] to build hierarchies of concurrent objects, our simulation proofs are upwards, and horizontal composition of concurrent objects requires their interfaces to be fully disjoint. In subsequent work on certified concurrent abstraction layers [Gu et al., 2018b], interfaces specify behavior at the level of individual threads, whereas we focus on specifications using simple state machines that are agnostic to the number of threads interacting with objects.

**5.2.5. Program Logics.** To our knowledge, while Hoare logics have long been applied in concurrent program verification, they have not been used for modular proof of examples combining classic concurrent data structures with transactional memory. However, many different extensions have been influential, including to our work. Rely-guarantee reasoning [Jones, 1983] supports *temporal* decomposition of a workload across concurrent threads. The pioneering work on concurrent separation logic [O'Hearn, 2007, Hobor et al., 2008] and its descendants [Windsor et al., 2017] tackled *spatial* decomposition of memory across separately verified data structures. Fruitful combination of these two techniques was demonstrated in the logics RGSep [Vafeiadis and Parkinson, 2007], LRG [Feng, 2009, Liang and Feng, 2013] and TaDA [da Rocha Pinto et al., 2014]. This line of work relies on ghost state to formulate functional-correctness properties. A number of program logics rise to this challenge, by defining flexible higher-order ghost state connected to notions of state-transition systems. For instance, the logics FCSL [Nanevski et al., 2014] and Iris [Jung et al., 2016a, 2015] build in different notions of monoids for expressing protocols [Liang and Feng, 2013], and the GPS logic [Turon et al., 2014] applied similar ideas in the context of weak memory.

These logics support much more flexible state-sharing than in our framework. However, in return, we keep our framework much simpler and modularly build it from basic definitions, fixing little more than the classic notion of simulation and linearizability. Yet, we show that elaborate concurrent objects such as transaction protocols and predicated data structures can be implemented modularly. Further, in contrast to correctness conditions for transactions that were often written as standalone definitions [Papadimitriou, 1979, Guerraoui and Kapalka, 2008, Doherty et al., 2013, Scott, 2006, Jagannathan et al., 2005], we show that serializability can be defined modularly based on linearizability and composition operations.

CHAPTER 6

# Future Work and Conclusions

## 6.1. Existing Work using Interaction Trees

Interaction trees were originally used to verify a simple HTTP server [Koh et al., 2019, Zhang et al., 2021] using VST [Cao et al., 2018]. Interaction trees also enabled a connection between VST specifications of the socket primitives used in that server and CertiKOS specifications [Mansky and Honoré, 2020]. Li et al. [2021] leverage the executable nature of interaction trees for model-based testing of networked applications.

The theory of interaction trees relies heavily on the Paco library for parameterized coinduction [Hur et al., 2013]. Challenges in the development of the ITree library have motivated further generalizations in Paco [Zakowski et al., 2020]. Silver and Zdancewic [2021] use interaction trees to extend the framework of Dijkstra monads, for specifying effectful programs, to reason about nontermination and uninterpreted events.

Vellvm formalizes the semantics of LLVM using interaction trees [Zakowski et al., 2021]. Vellvm serves as the target of a verified compiler for Helix, a domain-specific language for numerical algorithms [Zaliva et al., 2020], applying and extending the proof methodology presented in Chapter 3. Heapster [He et al., 2021] automatically extracts functional specifications from imperative programs. The semantics of both the imperative source language and the functional target language are defined using interaction trees. Vellvm and the imperative fragment of Heapster are of course both richer languages than Imp, and they leverage the existing theory of events and handlers to modularly structure additional effects such as nondeterminism and errors—for representing undefined behavior. The functional fragment of Heapster is also effectful—using interaction trees—as memory-unsafe programs are extracted to functional specifications which may raise errors.

Foster et al. [2021] implement the ideas of interaction trees in Isabelle/HOL, adapting to the lack of higher-order types as we discussed in Section 2.1.1, and subsequently develop novel semantics for process languages such as CSP. Song et al. [2022] presents a logic combining contextual refinement and separation logic. Much like the objects of Chapter 4, the behavior of a module is represented by a handler, using a generalization of interaction trees featuring dual notions of nondeterminism.

## 6.2. Improving the Definition of Interaction Trees

It may be worth revisiting some arbitrary decisions in the formal definition of interaction trees, to simplify parts of the library or to better support new extensions.

The `Tau` constructor is the root of much of the complexity in coinductive proofs, that motivates using the Paco library. This is an issue that makes it tedious to define new variants of weak bisimulation—for example to equate trees with different types of events, or to make some events "invisible". A different definition of `itree` such as the following one, in terms of the delay monad [Capretta, 2005]. This construction can be viewed as the "free monad transformer" [Kmett] applied to the delay monad ("`FreeT f Delay r`"), which suggests that its equational theory is obtained similarly by transforming—thus reusing—the equational theory of the delay monad:

```
(* The delay monad *)
CoInductive delay A :=
| Tau (delay A)
| Now (a : A).

Variant itreeF (E : Type → Type) (R : Type) (X : Type) : Type :=
| Vis (A : Type) (e : E A) (k : A → X)
| Ret (r : R).

CoInductive itree E R := delay (itreeF E R (itree E R)).
```

In ITree, event types are type constructors (`E : Type → Type`). One consequence is that the "injectivity of the `Vis` constructor", a seemingly natural property:

```
∀ (E : Type) (A B : Type) (e1 e2 : E A) (k1 k2 : A → itree E B),
                Vis e1 k1 = Vis e2 k2 → e1 = e2 ∧ k1 = k2,
```

is not actually provable in Coq. The issue is that the type `A` which occurs in the type of `e1` and `e2 : E A` is also an argument of `Vis`, so that the equality `Vis e1 k1 = Vis e2 k2` induces a heterogeneous equality between `e1` and `e2`, which is strictly weaker than homogeneous equality. Thus, the "injectivity of `Vis`" above is equivalent to the uniqueness of identity proofs (UIP):

```
∀ (A : Type) (x y : A) (p q : x = y), p = q
```

which is notably incompatible with univalence, an active topic of study in type theory [Univalent Foundations Program, 2013]. In practice, event types `E : Type → Type` are not arbitrary functions on types, but user-defined sum types whose constructors determine the type index. Under a suitable encoding of that assumption, the "injectivity of the `Vis` constructor" may be proved.

There may be further advantages to internalizing the fact that event types are built out of sums in the representation of event types (`E : Type → Type`). Currently, we have a typeclass `E -< F` which carries a mapping from `E` events to `F` events, but in practice this mapping is also injective, a fact which may be leveraged to obtain a general form of case analysis on events. For example, Yoon et al. [2022] propose a variant of that typeclass, denoted `D +? E -< F`, to express that the sum `D +' E` is isomorphic to `F`, so that we not only know how to map `E` events to `F` events, but we also have a partial inverse from `F` back to `E`. A possible concern for this class is brittleness, in that the "difference" `D` between `F` and `E` is only unique up to isomorphism, but there are many possible syntactically different values for `D` (*e.g.*, by associating sums differently) that may arise from instance resolution in different contexts. Baking the sum structure of event types into their representation may be a less brittle alternative to accomplish similar goals.

Lastly, interaction trees as defined in this dissertation are a natural representation of programs with "external choices", made by the environment. In contrast, representing "internal nondeterminism" has been a recurrent problem in several applications of interaction trees. This problem has been addressed in various ad-hoc ways: by defining a variant of weak bisimulation which makes nondeterministic branches "invisible" [Koh et al., 2019], by interpreting choices to a "set of itrees" monad [Zakowski et al., 2021], or, more recently,[10] by redefining `itree` with an additional constructor for nondeterminism. A principled framework for better understanding this problem and comparing such solutions may be an interesting direction for future exploration.

## 6.3. Automating Proofs of Categorical Equations

The ITree library relies heavily on categorical abstractions, via the Kleisli category of the `itree` monad and the category of handlers. Both applications presented in previous chapters exemplify how this lets us reason about the behavior of nontrivial systems (nonterminating programs and complex concurrent and transactional objects) equationally, further boiling down the problem to string-diagramatic intuitions. However, in practice, the formal proofs can become quite tedious due to an overwhelming amount of administrative bookkeeping: long chains of compositions must be reassociated explicitly inbetween "meaningful" steps of rewriting, and associators and unitors of monoidal categories are sometimes challenging puzzles to cancel out. It would be worthwhile to develop automated solvers (or adapt any existing ones) for equations in arbitrary monoidal categories and related structures such as those found in ITree. To verify such provers, an obvious property is soundness. Its dual, completeness, would also be useful to characterize the class of equations that may be solved automatically, to provide insight into the kind of properties that may be proved equationally, and also to identify weaknesses to address in further work.

## 6.4. Higher-Order Semantics with Games

Interaction trees are essentially first-order structures. Although it is possible to an extent to pass interaction trees through events, as we do in the framework of verified transactional objects (Chapter 4), there remain significant obstacles to model more expressive systems. For instance, in many languages objects are first-class entities, which may be instantiated dynamically and treated as run-time values. It also seems challenging obtain a semantics for recursive higher-order languages such as PCF, in spite of the existing features to express recursion in ITree.

Game semantics [Hyland, 1997, Abramsky et al., 1997, Hyland and Ong, 2000] are a promising approach to represent higher-order structures, having notably provided the first fully abstract denotational semantics for PCF. There is a strong similarity between game semantics and interaction trees, as interaction trees can be seen as strategies playing events as moves. Interfaces in game semantics are stateful: the set of possible moves changes depending on previous moves, whereas the set of events in

[10]Ongoing work: https://github.com/vellvm/ctrees

an interaction tree is fixed uniformly. This aspect makes games highly expressive and at the same time quite challenging to formalize in a proof assistant.

## 6.5. Conclusion

This dissertation presented interaction trees, a structure for constructing executable denotational semantics of programming languages in a proof assistant. It is implemented as a library in Coq providing powerful combinators for composing and reasoning about interaction trees (Chapter 2).

Equipping Imp and Asm with denotational semantics enabled a fully equational proof of compiler correctness (Chapter 3). An equational theory of loop operators fully encapsulated tricky coinductive reasoning about nontermination, and a theory of handlers allowed us to decompose the semantics of a language into sums of simple effects [Zakowski et al., 2020, He et al., 2021, Yoon et al., 2022].

The denotational nature of interaction trees also let us build a compositional framework of verified transactional objects (Chapter 4). Viewing transactions as first-class programs, interaction trees that may be manipulated by objects, let us formalize serializability—multi-method atomicity—in terms of linearizability—single-method atomicity. These two styles of concurrency live alongside each other in the resulting framework, allowing us to reason about composition patterns combining the best of both worlds, the performance of classical concurrent objects and the programmability of transactions.

The structure of interaction trees has a long and rich history in representing effectful programs and their interactions with the rest of the world (Chapter 5): IO trees [Hancock and Setzer, 2000], free monads [Swierstra, 2008], algebraic effects [Plotkin and Power, 2001, 2003]... The primary contribution of this thesis is to codify a core theory of interaction trees as a practical and reusable library, enabling further applications to formal verification in a proof assistant. While some aspects of the datatype definition may be worth revisiting, as discussed in Section 6.2, I believe improvements on that front can be made without significantly affecting the user-facing abstractions around which the library is organized, and which are based on category theory: effects induce monads, loop combinators follow the laws of iteration, and handlers form a monoidal category. These robust abstractions guide us towards a denotational approach to represent the behavior of programs, inspiring us to embrace equational reasoning and compositionality.

# Bibliography

Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in agda–objects and graphical user interfaces. *Journal of Functional Programming*, 27, 2017.

Samson Abramsky and Guy McCusker. Game semantics. In *Computational logic*, pages 1–55. Springer, 1999.

Samson Abramsky et al. Semantics of interaction: an introduction to game semantics. *Semantics and Logics of Computation*, 14(1), 1997.

Peter Aczel, Jirí Adámek, Stefan Milius, and Jiri Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, 300(1-3):1–45, 2003. doi: 10.1016/S0304-3975(02)00728-4. URL https://doi.org/10.1016/S0304-3975(02)00728-4.

Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

Jiří Adámek, Stefan Milius, and Jiří Velebil. Equational properties of iterative monads. *Information and Computation*, 208(12):1306–1348, 2010. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2009.10.006. URL https://www.sciencedirect.com/science/article/pii/S0890540110000854. Special Issue: International Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. URL http://www.cs.indiana.edu/~amal/ahmedsthesis.pdf.

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015.

Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures*, pages 534–549, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54458-7.

Timos Antonopoulos, Paul Gazzillo, Eric Koskinen, and Zhong Shao. Vertical composition of reversible atomic objects. 2016.

Heinrich Apfelmus. The operational monad tutorial. *The Monad.Reader*, Issue 15, 2010.

Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European*

*Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987212.

Andrew W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014. ISBN 978-1-10-704801-0. URL http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB.

Alasdair Armstrong, Brijesh Dongol, and Simon Doherty. Proving opacity via linearizability: a sound and complete method. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 50–66. Springer, 2017.

Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman. Nesting and composition in transactional data structure libraries. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 405–406, 2020.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Characterizing transactional memory consistency conditions using observational refinement. *Journal of the ACM (JACM)*, 65(1):1–44, 2017.

Jeremy Avigad, Mario Carneiro, and Simon Hudon. Data Types as Quotients of Polynomial Functors. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-122-1. doi: 10.4230/LIPIcs.ITP.2019.6. URL http://drops.dagstuhl.de/opus/volltexte/2019/11061.

Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for c/c++ concurrency. *ACM SIGPLAN Notices*, 48(1):235–248, 2013.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, Jan 2015.

Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 115–130, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9.

Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages. July 2010. URL https://www.microsoft.com/en-us/research/publication/formalizing-domains-ultrametric-spaces-and-semantics-of-programming-languages/.

Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co) datatypes for isabelle/hol. In *International Conference on Interactive Theorem Proving*, pages 93–110. Springer, 2014.

Stephen L. Bloom and Zoltán Ésik. *Iteration Theories - The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer, 1993. ISBN 978-3-642-78036-3. doi: 10.1007/978-3-642-78034-9. URL https://doi.org/10.1007/978-3-642-78034-9.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *International Conference on Computer Aided Verification*, pages 542–563. Springer, 2017.

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835703. URL http://doi.acm.org/10.1145/1835698.1835703.

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. *SIGPLAN Not.*, 45(6):330–340, jun 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806634. URL https://doi-org.proxy.library.upenn.edu/10.1145/1809028.1806634.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. Vst-floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reasoning*, 61(1-4):367–422, 2018. doi: 10.1007/s10817-018-9457-5. URL https://doi.org/10.1007/s10817-018-9457-5.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005. ISSN 1860-5974. doi: 10.2168/LMCS-1(2:1)2005. URL http://www.lmcs-online.org/ojs/viewarticle.php?id=55.

Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Symposium on Theoretical Aspects of Computer Software*, volume LNCS, pages 244–272. Springer-Verlag, 1994.

Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. Technical report, In Proceedings of the Conference on Category Theory and Computer Science, 1993.

Pavol Černỳ, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list implementations. In *International Conference on Computer Aided Verification*, pages 465–479. Springer, 2010.

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability. In *International Colloquium on Automata, Languages, and Programming*, pages 98–109. Springer, 2014.

James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015*, pages 110–125, Cham, 2015. Springer International Publishing. ISBN 978-3-319-25150-9.

Arthur Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 41–60, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_3. URL http://dx.doi.org/10.1007/978-3-642-37036-6_3.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250742. URL http://doi.acm.org/10.1145/1250734.1250742.

Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 93–106, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706312. URL http://doi.acm.org/10.1145/1706299.1706312.

Adam Chlipala. Infinite data and proofs. In *Certified Programming with Dependent Types*. MIT Press, 2017. URL http://adam.chlipala.net/cpdt/html/Cpdt.Coinductive.html.

Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16 (5):1512–1542, 1994.

Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2014.

Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. Transactional mutex locks. In *European Conference on Parallel Processing*, pages 2–13. Springer, 2010.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *In: International Conference on Functional Programming 2012, ACM Press*. Citeseer, 2012.

Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. 2009.

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 207–218, 2013. doi: 10.1145/2429069.2429094. URL https://doi.org/10.1145/2429069.2429094.

Thomas Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. Conflict abstractions and shadow speculation for optimistic transactional objects. In *Asian Symposium on Programming Languages and Systems*, pages 313–331. Springer, 2019.

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25 (5):769–799, 2013.

Avner Elizarov, Guy Golan-Gueta, and Erez Petrank. Loft: lock-free transactional data structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 425–426, 2019.

Michael Emmi and Constantin Enea. Violat: generating tests of observational refinement for concurrent objects. In *International Conference on Computer Aided Verification*, pages 534–546. Springer, 2019.

Xinyu Feng. Local rely-guarantee reasoning. In *POPL '09*, 2009.

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. In *Programming Languages and Systems, 18th European*

*Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 252–266, 2009. doi: 10.1007/978-3-642-00590-9_19. URL https://doi.org/10.1007/978-3-642-00590-9_19.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.

Simon Foster, Chung-Kil Hur, and Jim Woodock. Formally verified simulations of state-rich processes using interaction trees in isabelle/hol. In *32nd International Conference on Concurrency Theory*, 2021.

Carlos Eduardo Giménez. *Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants*. PhD thesis, École Normale Supérieure de Lyon, 1996.

Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-47770-9.

Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Concurrent libraries with foresight. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 263–274, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462172. URL http://doi.acm.org/10.1145/2491956.2462172.

Sergey Goncharov and Lutz Schröder. A coinductive calculus for asynchronous side-effecting processes. *CoRR*, abs/1104.2936, 2011. URL http://arxiv.org/abs/1104.2936.

Sergey Goncharov, Stefan Milius, and Christoph Rauch. Complete elgot monads and coalgebraic resumptions. *Electronic Notes in Theoretical Computer Science*, 325: 147–168, 2016.

Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg. Unifying guarded and unguarded iteration. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 517–533, 2017. doi: 10.1007/978-3-662-54458-7_30. URL https://doi.org/10.1007/978-3-662-54458-7_30.

Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In *International Conference on Concurrency Theory*, pages 256–271. Springer, 2012.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676975. URL http://doi.acm.org/10.1145/2676726.2676975.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM*

*SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 595–608, New York, NY, USA, 2015b. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676975. URL https://doi.org/10.1145/2676726.2676975.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661, 2018a. doi: 10.1145/3192366.3192381. URL http://doi.acm.org/10.1145/3192366.3192381.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 646–661, New York, NY, USA, 2018b. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192381. URL https://doi.org/10.1145/3192366.3192381.

Rachid Guerraoui. Modular atomic objects. *Theory and Practice of Object Systems*, 1 (2):89–99, 1995.

Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, 2008.

Dwight Guth. A formal semantics of python 3.3. 2013.

Tatsuya Hagino. Codatatypes in ml. *Journal of Symbolic Computation*, 8(6):629 – 650, 1989. ISSN 0747-7171. doi: https://doi.org/10.1016/S0747-7171(89)80065-3. URL http://www.sciencedirect.com/science/article/pii/S0747717189800653.

Peter Hancock. *Ordinals and interactive programs*. PhD thesis, University of Edinburgh, UK, 2000. URL http://hdl.handle.net/1842/376.

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 317–331, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44622-4.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 48–60, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930809. doi: 10.1145/1065944.1065952. URL https://doi.org/10.1145/1065944.1065952.

Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 196–213, Berlin, Heidelberg, 1997.

Springer Berlin Heidelberg. ISBN 978-3-540-68438-1.

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 387–388, 2014.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015a. doi: 10.1145/2815400.2815428. URL http://doi.acm.org/10.1145/2815400.2815428.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*, pages 449–465. Springer, 2015b.

Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, et al. A type system for extracting functional specifications from memory-safe imperative programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021.

Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345237. URL http://doi.acm.org/10.1145/1345206.1345237.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0818638109. doi: 10.1145/165123.165164. URL https://doi.org/10.1145/165123.165164.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton, NJ, USA, 2008. AAI3333851.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37 (1):67 – 111, 2000. ISSN 0167-6423. doi: https://doi.org/10.1016/S0167-6423(99)00023-4. URL http://www.sciencedirect.com/science/article/pii/S0167642399000234.

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 193–206, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429093. URL http://doi.acm.org/10.1145/2429069.2429093.

J Martin E Hyland and C-HL Ong. On full abstraction for pcf: I, ii, and iii. *Information and computation*, 163(2):285–408, 2000.

Martin Hyland. Game semantics. *Semantics and logics of computation*, 14:131, 1997.

Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70 – 99, 2006. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2006.03.013. URL http://www.sciencedirect.com/science/article/pii/S0304397506002659. Clifford Lectures and the Mathematical Foundations of Programming Semantics.

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, 2005.

Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, and Jan Vitek. Atomicity refinement for verified compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–30, 2014.

P. Johann, A. Simpson, and J. Voigtländer. A generic operational metatheory for algebraic effects. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 209–218, July 2010. doi: 10.1109/LICS.2010.29.

Cliff B. Jones. Specification and design of (parallel) programs. In *Information Processing 83*, volume 9, pages 321–332, 1983.

Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

Simon Peyton Jones. Tackling the awkward squad: monadic i/o, concurrency, exception and foreign-language calls in haskell. *Engineering theories of software construction*, pages 47–96, 2005.

André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996. doi: 10.1017/S0305004100074338.

Ralf Jung. Understanding and evolving the rust programming language. 2020.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL '15*, pages 637–650, 2015. ISBN 978-1-4503-3300-9.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 256–269, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951943. URL http://doi.acm.org/10.1145/2951913.2951943.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on*

*Functional Programming*, pages 256–269, 2016b.

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL): 1–32, 2019.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. doi: 10.1145/2804302.2804319. URL http://doi.acm.org/10.1145/2804302.2804319.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, volume 48, pages 59–70. ACM, 2013.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL http://doi.acm.org/10.1145/1629575.1629596.

Edward A. Kmett. free library. https://hackage.haskell.org/package/free.

Jérémie Koenig and Zhong Shao. Refinement-based game semantics for certified abstraction layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 633–647, 2020.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From c to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 234–248, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6222-1. doi: 10.1145/3293880.3294106. URL http://doi.acm.org/10.1145/3293880.3294106.

Bernhard Kragl, Shaz Qadeer, and Thomas A Henzinger. Refinement for structured concurrent programs. In *International Conference on Computer Aided Verification*, pages 275–298. Springer, 2020.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014. doi: 10.1145/2535838.2535841. URL http://doi.acm.org/10.1145/2535838.2535841.

Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Wait-free dynamic transactions for linked data structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'19, page 41–50, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362900. doi: 10.1145/3303084.3309491. URL https://doi.org/10.1145/3303084.3309491.

Kenneth Lamar, Christina Peterson, and Damian Dechev. Lock-free transactional vector. In *Proceedings of the Eleventh International Workshop on Programming*

*Models and Applications for Multicores and Manycores*, pages 1–10, 2020.

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, 2018. URL https://softwarefoundations.cis.upenn.edu/qc-current/index.html.

Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7): 107–115, 2009. doi: 10.1145/1538788.1538814. URL http://doi.acm.org/10.1145/1538788.1538814.

Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41 (1):1–31, 2008.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284 – 304, 2009. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2007.12.004. URL http://www.sciencedirect.com/science/article/pii/S0890540108001296. Special issue on Structural Operational Semantics (SOS).

Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *International Conference on Concurrency Theory*, pages 516–530. Springer, 2012a.

Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place. In *Workshop on the theory of transactional memory*, pages 137–151, 2012b.

Mohsen Lesani, Todd Millstein, and Jens Palsberg. Automatic atomicity verification for clients of concurrent data structures. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 550–567, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_37. URL http://dx.doi.org/10.1007/978-3-319-08867-9_37.

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J Bell, Adam Chlipala, Benjamin C Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–31, 2022.

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 338–354, 2018. doi: 10.1007/978-3-319-95582-7_20. URL https://doi.org/10.1007/978-3-319-95582-7_20.

Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 529–539, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3464798. URL https://doi.org/10.1145/3460319.3464798.

Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on*

*Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462189. URL http://doi.acm.org/10.1145/2491956.2462189.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199528. URL http://doi.acm.org/10.1145/199448.199528.

Peng Liu, Julian Dolby, and Charles Zhang. Finding incorrect compositions of atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 158–168, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491435. URL http://doi.acm.org/10.1145/2491411.2491435.

Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: Fixing linearizability violations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 543–560, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660217. URL http://doi.acm.org/10.1145/2660193.2660217.

N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214 – 233, 1995. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1995.1134. URL http://www.sciencedirect.com/science/article/pii/S0890540185711340.

Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706329. URL http://doi.acm.org/10.1145/1706299.1706329.

William Mansky and Wolf Honoré. Connecting higher-order separation logic to a first-order outside world. *Programming Languages and Systems. ESOP 2020*, 12075, 2020.

Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2018. URL http://coq.inria.fr. Version 8.8.1.

Coq development team. *The Coq proof assistant reference manual. The Gallina specification language. Co-inductive types, Caveat.* LogiCal Project, 2019. URL https://coq.inria.fr/distrib/V8.9.0/refman/language/gallina-specification-language.html#caveat. Version 8.9.0.

Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 257–275, 2015. doi: 10.1007/978-3-319-19797-5_13. URL https://doi.org/10.1007/978-3-319-19797-5_13.

J McCarthy and J Painter. Correctness of a compiler for arithmetic expressions. symposium in applied mathematics, vol. 19, mathematical aspects of computer science. *American Mathematical Society*, 1967.

Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. *ACM Sigplan Notices*, 43(5):15–26, 2008.

Markus Michelbrink. Interfaces as games, programs as strategies. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, pages 215–231, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-31429-5.

Stefan Milius. Completely iterative algebras and completely iterative monads. *Inf. Comput.*, 196(1):1–41, 2005. doi: 10.1016/j.ic.2004.05.003. URL https://doi.org/10.1016/j.ic.2004.05.003.

Robin Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157 – 173. Elsevier, 1975. doi: https://doi.org/10.1016/S0049-237X(08)71948-7. URL http://www.sciencedirect.com/science/article/pii/S0049237X08719487.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised.* MIT press, 1997.

Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23, June 1989. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55–92, 1991.

Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1990.

Andrzej S. Murawski and Nikos Tzevelekos. Higher-order linearisability. *Journal of Logical and Algebraic Methods in Programming*, 104:86–116, 2019. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2019.01.002. URL https://www.sciencedirect.com/science/article/pii/S2352220817302250.

Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: an exercise in mixed induction-coinduction. In *Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010.*, pages 57–75, 2010. doi: 10.4204/EPTCS.32.5. URL https://doi.org/10.4204/EPTCS.32.5.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 290–310, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-642-54832-1. doi: 10.1007/978-3-642-54833-8_16. URL http://dx.doi.org/10.1007/978-3-642-54833-8_16.

Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49498-1.

Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

Ross Paterson. A new notation for arrows. *ACM SIGPLAN Notices*, 36(10):229–240, 2001.

Christina Peterson and Damian Dechev. A transactional correctness tool for abstract data types. *ACM Trans. Archit. Code Optim.*, 14(4), November 2017. ISSN 1544-3566. doi: 10.1145/3148964. URL https://doi.org/10.1145/3148964.

Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Papers Presented at the Symposium*. ACM Press, Jan 1993.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

Maciej Piròg and Jeremy Gibbons. The coinductive resumption monad. *Electronic notes in theoretical computer science.*, 308:273–288, 2014. ISSN 15710661.

Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45315-4.

Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45931-6.

Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3 – 15, 2004a. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2004.03.009. URL http://www.sciencedirect.com/science/article/pii/S1567832604000268. Structural Operational Semantics.

Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.

Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), December 2013. doi: 10.2168/LMCS-9(4:23)2013. URL https://lmcs.episciences.org/705.

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS) 2002, 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2014.

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 243–259, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_21. URL http://dx.doi.org/10.1007/978-3-642-31424-7_21.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and algebraic effects: What binds them together. Technical Report CW699, Department of Computer Science, KU Leuven, 2016.

Dana Scott. Data types as lattices. *SIAM Journal on computing*, 5(3):522–587, 1976.

Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

Michael Scott. Sequential specification of transactional memory semantics. 2006.

Anton Setzer. Object-oriented programming in dependent type theory. *Trends in functional programming.*, 7, 2006.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 51–64, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048073. URL http://doi.acm.org/10.1145/2048066.2048073.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 204–213, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917103. doi: 10.1145/224964.224987. URL https://doi.org/10.1145/224964.224987.

Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

Youngju Song, Minki Cho, Dongjae Lee, and Chung-Kil Hur. Conditional contextual refinement (ccr), 2022. URL https://arxiv.org/abs/2203.07431.

Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 682–696, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908112. URL http://doi.acm.org/10.1145/2908080.2908112.

Robert C Steinke and Gary J Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, 2004.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4): 423–436, 2008.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 691–707, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660243. URL http://doi.acm.org/10.1145/2660193.2660243.

Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–356, 2013.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Tarmo Uustalu and Niccolò Veltri. The delay monad and restriction categories. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing – ICTAC 2017*, pages 32–50, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67729-3.

Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*. 2007.

Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. Layered and object-based game semantics. *Proceedings of the ACM on Programming Languages*, 2022.

Philip Wadler. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992.*, pages 233–264, 1992. doi: 10.1007/978-3-662-02880-3_8. URL https://doi.org/10.1007/978-3-662-02880-3_8.

Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51. IEEE, 2018.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368, 2015. doi: 10.1145/2737924.2737958. URL http://doi.acm.org/10.1145/2737924.2737958.

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. Starling: Lightweight concurrency verification with views. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 544–569, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63387-9.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):

1–32, 2019.

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages*, (ICFP), 2022.

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 71–84, 2020.

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.

Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. Verified translation between purely functional and imperative domain specific languages in helix. In Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel, editors, *Software Verification*, pages 33–49, Cham, 2020. Springer International Publishing. ISBN 978-3-030-63618-0.

Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC)*, 5(1):1–37, 2018.

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. Verifying an http key-value server with interaction trees and vst. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.