Building Systems with Guarantees

Anders Alnor Mathiasen

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

Building Systems with Guarantees

Verification of Transactional Databases and of a New Language with Effect Handlers and Region-based Memory Management

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Anders Alnor Mathiasen
November 15, 2026

Abstract

In this Ph.D. dissertation, we present two new scientific contributions to help programmers build systems with guarantees all the way down to the fine-grained semantics of program executions. At the basis of our contributions, we use higher-order concurrent separation logic; this allows us to make modular proof of distributed systems and make claims about programming languages precise.

In the first half of the dissertation (Chapter 2 and Chapter 3), we focus on the first contribution which is a method for formally verifying transactional databases. Until the contribution of our method, it is not possible to verify that database implementations correctly implement the transactional guarantees that they are supposed to give programmers that use them in their systems. We give separation logic specifications for a number of isolation levels, and show that any database that is verified against the separation logic specifications implements its isolation levels as formalized in the transactional consistency models used in the database community. The result is a free theorem proven directly from the per-operation separation logic specifications for the database operations without any additional assumptions, and it adds no runtime overhead to database implementations.

In the second half of the dissertation (Chapter 4), we present the second contribution which is a new programming language called *Yarrow*, an ML-like language with (one-shot and multi-shot) effect handlers and region-based memory management. This is the first time this combination of language features has been combined in one language; the control flow of effect handlers break the stack discipline of functions used by region-based memory management strategies, and multi-shot effect handlers means that regions can be exited more than once. We create an operational semantics that explains the details of the intricate behaviors of this combination of programming language features. Using the operational semantics of Yarrow, we define the *Yarrow Logic*, a program logic for making modular proofs of correctness for programs written in Yarrow. We verify a number of case studies showing that programs with effect handlers can avoid the less efficient garbage collection, which so far has been the norm for programs with effect handlers, by using regions based memory management.

All the results in the dissertation are mechanized in the Rocq proof assistant giving us high guarantees of correctness for the contributions.

Resumé

I denne ph.d.-afhandling præsenterer vi to nye videnskabelige bidrag, der skal hjælpe programmører med at bygge systemer med garantier hele vejen ned til den finkornede semantik for programeksekveringer. Bidragene bygger på højere ordens samtidig separationslogik; dette gør det muligt for os at lave modulære beviser for distribuerede systemer og gøre påstande om programmeringssprog præcise.

I afhandlingens første halvdel (Kapitel 2 og Kapitel 3), fokuserer vi på det første bidrag, som er en metode til formelt at verificere transaktionelle databaser. Indtil bidraget af vores metode, er det ikke muligt at verificere, at implementering af databaser korrekt implementerer de transaktionelle garantier, som de skal give programmører, der bruger dem i deres systemer. Vi opstiller separationslogik-specifikationer for en række isolationsniveauer, og viser at enhver database, der er verificeret mod separationslogik-specifikationerne, implementerer sine isolationsniveauer som formaliseret i de transaktionelle konsistensmodeller, der anvendes i database-forskningsmiljøet. Resultatet er et free theorem bevist direkte ud fra de operationsspecifikke separationslogik-specifikationer for databaseoperationerne uden yderligere antagelser, og der tilføjes ingen eksekveringsomkostninger til databaseimplementeringerne.

I afhandlingens anden halvdel (Kapitel 4) præsenterer vi det andet bidrag, som er et nyt programmeringssprog kaldet *Yarrow*, et ML-lignende sprog med (one-shot og multi-shot) effect handlers og regionsbaseret hukommelsesstyring. Dette er første gang, denne kombination af sprogkonstruktioner er blevet samlet i ét sprog; kontrolflowet med effect handlers bryder den stack-disciplin for funktioner, som anvendes af regionsbaserede hukommelsesstyringsstrategier, og multi-shot effect handlers betyder, at regioner kan forlades mere end én gang. Vi udvikler en operationel semantik, der forklarer detaljerne i den indviklede opførsel, som denne kombination af sprogkonstruktioner medfører. Ved brug af Yarrows operationelle semantik definerer vi *Yarrow Logic*, en programlogik til at lave modulære korrekthedsbeviser for programmer skrevet i Yarrow. Vi verificerer en række casestudier, der viser, at programmer med effect handlers kan undgå den mindre effektive garbage collection, som hidtil har været normen for programmer med effect handlers, ved at bruge regionsbaseret hukommelsesstyring.

Alle resultater i afhandlingen er mekaniserede i bevisassistenten Rocq,

hvilket giver os høje garantier for korrektheden af bidragene.

Acknowledgments

Thank you to my advisors Lars Birkedal and Amin Timany. I have appreciated all of our unfiltered discussions, and I admire your ability to ask sharp questions that push the research forward. You gave me the freedom and encouragement to pursue my own research ideas — that meant a lot to me.

I have enjoyed being a part the Logic & Semantics group at Aarhus University that Lars has been building over many years, thank you to all the people that made it a great environment during my stint.

To the REMS group at Cambridge University, at which I was on a research stay for the first half of 2025, thank you for welcoming me into your group and showing me around Cambridge and London.

To friends and family, thank you for being patient with me.

Anders Alnor Mathiasen,
Aarhus, November 15, 2026.

Contents

Abstract **i**

Resumé **iii**

Acknowledgments **v**

Contents **vii**

I Overview **1**

1 Introduction **3**
1.1 Program Logics 4
1.2 Verifying Transactional Databases 7
1.3 Programming Language Design: Combining Effect Handlers and Region-Based Memory Management 14
1.4 Academic Contributions 21

II Publications **23**

2 Reasoning about Weak Isolation Levels in Separation Logic **25**
2.1 Introduction 26
2.2 Specifying Read Uncommitted 29
2.3 Specifying Read Committed 38
2.4 Read Committed Implies Read Uncommitted 41
2.5 Specifying Snapshot Isolation 44
2.6 Snapshot Isolation Implies Read Committed 51
2.7 Verifying that a Multi-Version Concurrency Control Key-Value Store Implements Snapshot Isolation 51
2.8 Related Work 58
2.9 Conclusion and Future Work 60
2.10 Appendix: Utility Code 61
2.11 Appendix: Example Portfolio 66

2.12 Appendix: Serializability Does not Imply Snapshot Isolation in Separation Logic 68

3 Verifying Isolation Levels of Database Implementations for Free Using Separation Logic 69
3.1 Introduction 69
3.2 Technical Overview and Prior Work 74
3.3 A Realistic State-based Model 78
3.4 Specifying Transactional Consistency in Separation Logic . . . 83
3.5 Connecting Transactional Models to Semantic Models of Program Execution 88
3.6 Proof Structure: Transactional Library Implements Transactional Model 97
3.7 Proof Sketch: Transactional Library Implements Transactional Model 100
3.8 Related Work 103
3.9 Conclusion and Future Work 103
3.10 Appendix: Proving safety of examples 104

4 Yarrow: Reconciling Effect Handlers and Region-Based Memory Management 107
4.1 Introduction 107
4.2 Background 111
4.3 Operational Semantics 115
4.4 Program Logic 126
4.5 Case Study: LIFO data structure 137
4.6 Case study: Checkpointing 138
4.7 Case Study: Asynchronous Computation 139
4.8 Conclusion, Related Work and Future Work 142
4.9 Appendix: Condition for effect handlers with locally allocated continuation closures 144

Bibliography 145

Part I

Overview

Chapter 1

Introduction

Building reliable software systems without bugs is notoriously difficult, especially with the ever-increasing demand for more performant systems with higher througput. This Ph.D. dissertation answers two main research questions with the goal of aiding programmers in building software systems with correctness guarantees.

The first research question we answer is: *how to build verifiable correct transactional databases?* Modern databases are highly concurrent and provide transactions as a mechanism for programmers to group database operations into atomically applied units. Transactions have intricate semantics, with a long tradition of study in the database community, but until the contributions in this dissertation, there exists no method for verifying that a database implementation actually provides the guarantees that database vendors claim it does.

The second research questions we answer is: *how to reconcile effect handlers and region-based memory management in a programming language?* Effect handlers are a generalization of exception handlers and a feature of programming languages that allows programmers to suspend computation to an effect handler by performing effects, and to resume computation again using continuations. This powerful feature gives programmers a lot of flexibility of the control flow of programs, but it complicates the use of fine-grained memory management strategies such as region-based memory management that are more efficient than automatic garbage collection. We contribute a new programming language *Yarrow* that reconciles effect handlers with region-based memory management, and Yarrow Logic, a logic in which we can verify programs written in Yarrow.

To give thorough answers to these research questions, we must reason about the execution of the code that programmers write. Program logics allow us to specify and verify software with respect to detailed models of program execution, thus, program logics form the basis of the work in this dissertation. As we also formalize all of our contributions in a proof assistant, we obtain high assurance that our results are correct.

We use the remainder of this chapter to motivate the research questions, and provide the reader with background information to understand the contributions that we present throughout the papers in Chapter 2, Chapter 3 and Chapter 4.

1.1 Program Logics

As a starting point of our study, we use an ML-like language with mutable references (this is the sequential part of what is referred to as HeapLang [Iris-Team, 2026]). Below is a subset of the expressions and values of this language:

$$
\begin{aligned}
& z \in \mathbb{Z} \qquad x \in \mathit{Var} \qquad \ell \in \mathit{Loc} \\
v \in \mathit{Val} ::= {} & () \mid z \mid x \mid \ell \mid \texttt{true} \mid \texttt{false} \mid \texttt{rec}\, f\, x = e \\
& \mid (v_1, v_2) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \ldots \\
e \in \mathit{Expr} ::= {} & v \mid x \mid \circledcirc_1\, e \mid e_1 \circledcirc_2 e_2 \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
& \mid \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid e_1\ e_2 \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{ref}\ e \mid !e \mid e_1 \leftarrow e_2 \\
& \mid (\texttt{match}\ e_1\ \texttt{with}\ \texttt{inl}\ x_1 => e_1 \mid \texttt{inj}\ x_2 => e_2) \mid \ldots
\end{aligned}
$$

To precisely define how programs written in this language execute, we use an operational semantic with reduction rules for language primitives like the one below for storing a value to a location on the heap (the heap h is a finite map from locations to values):

$$\big(\ell \leftarrow v_1, h[\ell := v_2]\big) \rightsquigarrow \big((), h[\ell := v_1]\big)$$

To create a program logic for reasoning about the language above, and the extensions we apply to the language in order to answer the research questions of this dissertation, we use Iris [Jung et al., 2015, 2016, 2018; Krebbers et al., 2017], a higher-order concurrent separation logic framework; Iris is an answer to Matthew Parkinson's request [Parkinson, 2010] for having a unified framework for creating sequential and concurrent separation logics [Brookes, 2007; OHearn, 2007; O'Hearn et al., 2001; Reynolds, 2002b] that can reason about many different program constructs. In the remainder of this section, we cover the very basics of Iris and emphasize the parts that are most important to the chapters of this dissertation. For a more complete explanation of Iris, we refer to Birkedal and Bizjak [2017] for a gentle introduction and Jung et al. [2018] for more in depth explanations.

Reasoning about program executions We can instantiate Iris with an operational semantic of our choice and reason about the execution of programs using *Hoare triples* [Hoare, 1969] and *weakest preconditions*.

$$\{P\}\, e\, \{v.\, Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\, Q\})$$

A Hoare triple $\{P\}\, e\, \{v.\, Q\}$ asserts that if the precondition P holds, then the expression e is safe to execute, and if e terminates with a return value v then

Q holds. By executing safely, we mean that the expression can always reduce w.r.t to the operational semantic if the expression has not already reduced to a value (*i.e.,* the execution does not get stuck). Hoare triples are defined in terms of $\mathsf{wp}\, e \{v.\, Q\}$, which is the weakest precondition that must hold for e to execute safely and its return value v satisfy Q. Sometimes, we express reasoning rules directly in terms of the weakest precondition instead of using Hoare triples. Iris is a resource aware separation logic; the persistently modality $\Box$ in the definition of Hoare triples makes sure no exclusive owned resources are assumed prior to proving the Hoare triple. Persistent propositions, such at those surrounded by the persistently modality, does not rely on exclusive ownership and can be duplicated freely, thus a proof of a Hoare triple can be used many times. An example of a resource requiring exclusive ownership is the points-to resource $\ell \mapsto v$ which asserts that the value v is stored at location ℓ on the heap. When used with the separating conjunction $*$ (the separating conjunction is a conjunction requiring disjoint resources between its clauses, thus only duplicable resources can be used in both branches), we can assert that more than one points-to resource for the same location can not exist, *i.e.,* $\ell \mapsto v * \ell' \mapsto v \vdash \ell \neq \ell'$. The *magic wand* $P \mathrel{-\!\!*} Q$ in the definition of Hoare triples is an implication with exclusive ownership of the resources in P.

Throughout this dissertation, we rely on the creation of different Iris resources, sometimes referred to as *ghost state*, in our proofs. But, most user-facing resources, *i.e.,* resources that are not hidden behind abstraction layers, are based on points-to resources. For instance, we can use points-to resources and Hoare triples to reason about the execution of heap manipulating programs according to the operational semantics as seen in the Ht-store and Ht-load rules below:

Ht-store
$\{\ell \mapsto -\}\, \ell \leftarrow w\, \{v.\, v = () \land \ell \mapsto w\}$

Ht-load
$\{\ell \mapsto w\}\, !\ell\, \{v.\, v = w \land \ell \mapsto w\}$

Evaluation contexts To reason about programs made of up several expressions, we use *evaluation contexts* to define the order of executing expressions in the programming language. An evaluation context is essentially a program with a hole in it, *e.g.,* $\bullet$ is the empty evaluation context, and $\texttt{if}\, \bullet\, \texttt{then}\, e_1\, \texttt{else}\, e_2$ is an evaluation context for the `if`-expression stating that we must first evaluate the condition-expression before we can proceed with evaluating the rest of the `if`-expression — this is enforced by the reduction rule for evaluation contexts (all reductions for expressions, defined with the $\rightsquigarrow$ arrow, go through this rule):

$$\frac{(e, h) \rightsquigarrow (e', h')}{(K[e], h) \to (K[e'], h')}$$

Evaluation contexts naturally come up when we want to break proofs of programs down into smaller pieces as seen in the HT-BIND rule.

$$\text{HT-BIND}\quad \frac{S \vdash \{P\}\, e\, \{v.\, Q\} \qquad S \vdash \forall v. \{Q\}\, K[v]\, \{w.\, R\}}{S \vdash \{P\}\, K[e]\, \{w.\, R\}}$$

Modularity with the frame rule At the heart of separation logic and Iris's modularity lies the HT-FRAME rule; with the HT-FRAME rule, we can "frame away" resources P that are not needed for proving the safety of an expression e.

$$\text{HT-FRAME}\quad \frac{S \vdash \{P\}\, e\, \{v.\, Q\}}{S \vdash \{P * R\}\, e\, \{v.\, Q * R\}}$$

Thus, any specification for an expression e only needs to mention the resources that are used locally to prove safety of e, and a proof of such can still be used in any larger proof effort that uses arbitrary other resources.

Later in Chapter 1.3.1, we see that when we add effect handlers to the programming language, the HT-FRAME is no longer sound in the form stated above.

Reasoning about concurrency with invariants In Chapter 3 and Chapter 4 when we reason about databases, we add concurrency primitives (a new thread executing e can be created using **fork** $\{e\}$) and encode sequentially consistent heap semantics (*i.e.,* heap manipulating expressions are atomic) in our reduction rules where threads are nondeterministicly picked to take a step. See the specification for **fork** $\{e\}$ below:

$$\text{HT-FORK}\quad \frac{S \vdash \{P\}\, e\, \{w.\, \mathsf{True}\}}{S \vdash \{P\}\, \mathbf{fork}\ \{e\}\ \{v.\, v = ()\}}$$

Further, we add primitives for network communication (more on this later in Chapter 1.2.3).

When reasoning about concurrent programs that share mutable state, we can not use points-to resource directly as they require exclusive ownership which we can not give to multiple threads. Iris supports reasoning with invariants in which we can take an Iris propositon P, possibly containing exclusive resources, and create a persistent invariant $\boxed{P}$ that we can duplicate such that all threads have a copy (we omit invariant names here that prevent taking ownership of resources in an invariant more than once). Using the HT-INV-TIMELESS rule, threads can use the contents of an invariant around *atomic* expressions (generally, first order propositions that do no contain invariants are timeless,

i.e., satisfy timeless P, see [Jung et al., 2018] for details).

$$\text{HT-INV-TIMELESS}\quad \frac{e \text{ is atomic} \qquad \mathsf{timeless}\ P \qquad S \wedge \boxed{P} \vdash \{P * Q\}\, e\, \{v.\, P * R\}}{S \wedge \boxed{P} \vdash \{Q\}\, e\, \{v.\, R\}}$$

Atomic expression are expressions that evaluate to a value in a single step as defined in the reduction rules of the operational semantics, *i.e.,* they evaluate in an atomic step of the machine on which the program runs — thus it is safe to assume exclusive ownership of shared resources as no other thread can manipulate the resources at the same time.

Not all expression for which we want to use invariants are atomic expressions. For instance, database operations are not atomic expression of our programming language, they are operations made up of several expressions and calls over the network. To allow for the use of invariants for these expressions, we make use of *atomic Hoare triples* [da Rocha Pinto et al., 2014] (we comment on this in Chapter 2.2.1 and Chapter 3.4.2 also). Informally, an operation that can be specified with an atomic Hoare triple (the syntax of atomic Hoare triples is $\langle P \rangle\, e\, \langle v.\, Q \rangle$) exposes a *linearization point* [Herlihy and Wing, 1990]; a linearization point is a point in time when the effects of the operation takes place instantaneously. In the context of program logics, this means that the operation being specified with the atomic Hoare triple consists of an atomic expression around which the operation takes effect (by assuming the precondition P and asserting the postcondition Q) and invariants can be opened. The HT-ATOMIC rule below show the property, similar to HT-INV-TIMELESS, that atomic Hoare triples enjoy:

$$\text{HT-ATOMIC}\quad \frac{S \vdash \langle R * P \rangle\, e\, \langle v.\, Q * R \rangle}{S \wedge \boxed{R} \vdash \langle P \rangle\, e\, \langle v.\, Q \rangle}$$

1.2 Verifying Transactional Databases

Databases are one of the most important pieces of infrastructure backing most stateful applications from banking applications to social media platforms. These applications have large numbers of concurrent users that databases must support with very little latency. Modern databases support transactions to group operations together into atomic units; either all the updates of a transaction are applied to the database or no updates are applied at all. Yet, there is no way of verifying that a database implementation actually implements its transactional guarantees. The kind of database operations we consider are: a `start` operation for starting transactions, a `read` operation for reading a value from a key (*e.g.,* `read` x), a `write` operation for writing a value v to a key x (*e.g.,* `write` x v) and a `commit` operation to end transactions. The `commit` operation returns a boolean to indicate whether the transaction successfully committed, or it aborted to not violate any transactional guarantees.

Complexity of modern databases To give some intuition as to why it is highly non-trivial to implement consistency guarantees for transactions in databases, we unfold some of the complexity of modern databases. Below, we list a number of categories that each contribute with some amount of complexity:

- **Network:** Messages sent between machines do not necessarily arrive, and if they arrive, we can not assume it is in the order in which they were sent. Such guarantees are properties of the communication protocol used between two endpoints, *e.g.,* the text transfer protocol (HTTP), or a remote procedure call library, may use the TCP (transport layer protocol) to ensure this.

- **Concurrency:** Many concurrent users create concurrent access. To control the concurrent access this without blocking the database entirely by handling one transaction at a time, concurrency control algorithms such as Two Phase Locking [Eswaran et al., 1976] or Multi-Version Concurrency Control [Bernstein and Goodman, 1983] are used. In Two Phase Locking, transactions grab shared locks and exclusive locks per key as a transaction is processed. Multi-Version Concurrency Control is an approach where multiple versions of the data is saved to enable reading from consistent snapshots of the database while not blocking for new updates. Now, the measures we just described are about concurrency at the database level on the granularity of transactions (we can process more than one transaction at a time), there is also per operation concurrency (the ability to process more than one operation at a time). For the latter, we utilize multiple cores on the machine level to obtain parallelism and make use of *e.g.,* Mutexes and Compare-And-Swap operations to safely handle access to shared mutable memory.

- **Sharding:** Memory or durable storage on a single machine is finite, therefore databases create so-called shards to distribute data over multiple machines; each shard is responsible for storing some fraction of the data that the database holds. Clients of the database can make operations on data that spans multiple shards, thus measures must be taken to ensure that updates are made correctly across shards. The two-phase commit protocol [Gray, 1978] is an example of a protocol for committing distributed atomic transactions (transactions across multiple shards).

- **Replication:** It is beneficial to replicate the same data onto more than one physical machine such that the database can remain available even if some machines become unavailable, for instance due to a network failure. Distributing data across a number of servers implies that some coordination must take place between the replicas to serve clients with consistent data. For this purpose, consensus algorithms such as Paxos

[Lamport, 2001] or Raft [Ongaro and Ousterhout, 2014] can be used for achieving *state machine replication*, a technique in which a deterministic state machine is spread across multiple physical machines. With Raft and Paxos, a database can tolerate failures as long as a majority of the servers are responsive.

- **Reconfiguration:** The servers on which a database runs are unlikely to stay the same throughout the lifetime of the database, for instance due to updates or replacement of hardware. Raft and Paxos has reconfiguration procedures for changing the underlying servers of the state machine replication while continuing to serve clients so no downtime occurs.

- **Durability:** For data to survive when machines crash, *e.g.,* due to power outages or natural catastrophes, not all data can stay in memory (RAM) but has to be written to durable storage in an atomic manner. ARIES [Mohan et al., 1992] is a crash recovery algorithm that guarantees every committed transactions updates are permanent — even if the machine crashes and memory is lost; it achieves this by retaining a log with enough information to redo or undo operations as needed.

- **Database model:** A key-value store model is one model for storing data with a simple API. The relational database model is an alternative where data is organized into tables, with rows and columns, and predominantly queried with SQL. Some applications prefer to use the graph database model where data is stored as structured graphs.

A database that efficiently handles all of these aspects is out of scope of what we can verify in this dissertation. But, we emphasize that the goal, which we have achieved, has been to come up with a method that can be used to verify databases with all these aspects given enough time and resources: our method does not make any assumptions about implementation details. In Chapter 2, we verify a database implementation using a multi-version concurrency control algorithm. We consider it future work to verify a database implementation with all the bells and whistles listed above.

1.2.1 Isolation Levels

To describe the transactional guarantees that database implementations give, we use *isolation levels*. Isolation levels describe how isolated concurrent transactions are from each other. The textbook isolation level is *serializability*, this gives strong guarantees by saying that concurrent transactions must execute as if they are executing one at a time. Modern databases often do not use serializability for two reasons: (1) serializability is an overly strong guarantee not needed by applications, and (2) serializabilty decrease throughput of the database (*i.e.,* the database's ability to process many concurrent transactions in a short

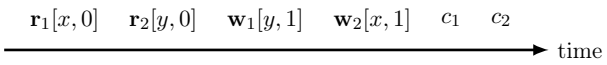
amount of time) compared to weaker isolation levels. A popular so-called weak isolation level is *snapshot isolation*; this, together with *read uncommitted* and *read committed*, are the levels we focus on in this dissertation. Snapshot isolation permits more concurrency than serializability by famously allowing the *write skew example* below (we verify this example in Chapter 2.5):

$$
\begin{array}{ll}
T_1 : \texttt{start} & T_2 : \texttt{start} \\
\quad \texttt{read}\ x & \quad \texttt{read}\ y \\
\quad \texttt{write}\ y\ 1 & \quad \texttt{write}\ x\ 1 \\
\quad \texttt{commit} & \quad \texttt{commit}
\end{array}
$$

This example consists of two concurrent transactions T_1 and T_2 (we can think of the transactions as running on the machines of two separate clients that interacts with the database): the first transaction reads from the key x and writes to the key y, whereas the second transaction reads from y and writes to x. Snapshot isolation was first presented with an algorithmic description by Berenson et al. [1995] that states transactions can read from a snapshot of the database created when the transaction starts, and that a transaction can commit if no other concurrent transactions commit updates to the same keys (no write conflicts) in the time between the starting snapshot of the transaction was taken, and the transaction wants to commit. Under snapshot isolation, we can always assert that both transactions above successfully commit (this example can not break the guarantees snapshot isolation gives), but because serializability imposes a strict order on the execution of transactions, it is not the case for serializability; the executions in which both transactions read from the same snapshot of the database, and don't see each other's updates (the effect of the `write` operations), does not correspond to an execution in which the two transactions execute one at a time (we consider these particular executions in more detail below and use it as a motivational example throughout Chapter 3).

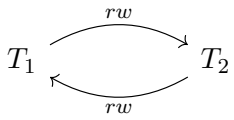
1.2.2 Transactional Consistency Models

Originally, isolation levels were defined and distinguished from each other using informal natural language descriptions and small examples of executions called phenomena; the ANSI SQL standard of 1992 defines a single phenomenon for each of the isolation levels read uncommitted, read committed, repeatable read and serializability. Each isolation level must prohibit the phenomena of all the previous levels in the aforementioned order. In Berenson et al. [1995], additional phenomena are added to make the definitions of isolation levels more precise (see Chapter 2.1 or Chapter 3.1 for more background on isolation levels); one of these is the write skew phenomenon that sets serializability and snapshot isolation a part:



This exactly shows a particular execution in the example from the previous subsection that snapshot isolation admits but serializability prohibits: the `read` operations of the two transactions are executed first, and they return the initial state of the database, in this case the value 1. Thereafter, transaction T_1 performs its `write` operation followed by T_2; this makes it impossible to order the transactions, as required by serializability, as neither transaction observers the `write` operation of the other transaction.

The use of phenomena to define isolation levels was criticized for being imprecise as it does not generalize; it by no means talk about all the possible behavior between a database and arbitrary clients using it. This led to a splurge of research in transactional consistency models for precisely defining which executions are allowed under different isolation levels. The works of Adya [1999] and Adya et al. [2000] starts this line of research by defining isolation levels in terms of conditions on graphs. Graphs are made from executions (more precisely, they use a custom notion of *histories*) where the nodes in the graphs are transactions, and the edges represent different relationships between transactions. For instance, consider the graph for the write skew phenomenon below:



Here, the two transactions T_1 and T_2 form the nodes, and between nodes we have so-called *anti-dependency* edges that form a cycle. There is an anti-dependency edge going from T_1 to T_2 because T_1 read the value of x that immediately preceded the value of x written by T_2, likewise for the edge going from T_2 to T_1 but for the value of y. The serializability isolation level is defined by, among other things, not allowing cycles of anti-dependency edges. Contrary to the phenomena used prior to Adya [1999] and Adya et al. [2000], the definitions in Adya's model generalizes over all possible executions of transactions: isolation levels are general statements precisely defining which relationship are allowed in any workload of transactions.

Many different types of models have been created to formalize transactional guarantees, *i.e.,* isolation levels; in Chapter 3.1 we list many of these works. But, all these models have in common that none of them present a method for formally verifying an actual database implementation; that is, none of these models let us reason at the level of a detailed semantics of code execution — they work at a much higher level of abstraction. For instance, the work of Cerone et al. [2015b] create operational specifications *"that are closer to implementations"* [Cerone et al., 2015b] in form of an operational semantics. But, the operational semantics are at the granularity of database operations, that is reduction relations are giving for the `start`, `write`, `read` and `commit` operations. What we are interested in, is reasoning and giving guarantees all

the way down to the implementation details using detailed models of program execution.

1.2.3 Separation Logic Specifications

As emphasized earlier, using the Iris separation logic, we can reason at the level of program executions by instantiating Iris with an operational semantics of our choice.

Reasoning at the implementation level of databases For verfying isolation levels of databases, we use the Aneris [Aneris-Team, 2023; Krogh-Jespersen et al., 2020] program logic, an instantiation of Iris with an OCaml like language AnerisLang. AnerisLang is a subset of OCaml (we have a transpiler from OCaml to AnerisLang), with concurrency, mutable references and primitives for communication over UDP sockets. In essence, AnerisLang is an extension of HeapLang with the following expressions:

$$\begin{aligned} e \in \mathit{Expr} ::= \ldots \mid & \texttt{makeaddress}\ e_1\ e_2 \mid \texttt{getaddress}\ e \mid \texttt{socket} \\ \mid & \texttt{socketbind}\ e_1\ e_2 \mid \texttt{sendto}\ e_1\ e_2\ e_3 \mid \texttt{receivefrom}\ e \\ \mid & \texttt{settimeout}\ e_1\ e_2\ e_3 \mid \texttt{start}\ e_1\ e_2 \end{aligned}$$

These operations are modelled after the BSD-socket API (see Aneris-Team [2023] for details), except for the **`start`** operation, which is a special operation used for spawning new nodes on the network similar to how the **`fork`** operation creates new threads (in **`start`** $e_1\ e_2$ the expression e_1 should evaluate to an IP address of a machine on which the program e_2 starts executing). The operational semantics of the expressions relies on a model of the network (the network is a part of the operational semantics that Iris is instantiated with in Aneris). Below, we see just how granular the operational semantics that we build upon is w.r.t to the network:

$$\text{MESSAGE-DUBLICATE}\ \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M}) \to (\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M} \uplus \{m\})} \qquad \text{MESSAGE-DROP}\ \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M}) \to (\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M} \setminus \{m\})}$$

Besides the heaps $\vec{\mathcal{H}}$ for each machine of a network node, the program state contains $\vec{\mathcal{S}}$, which are the active sockets created by nodes in the network that are available for message delivery, and a message soup containing all messages that have been sent. The message soup is used to model a UDP-like network in which messages can be duplicated and dropped as captured by the reduction rules above. We have omitted the rule for message delivery: this reduction rule captures the last characteristic of the UDP protocol, namely that the delivery order of messages can be arbitrarily reordered. Due to prior work by Gondelman et al. [2023], we do not have to reason at the level of UDP when working with database implementations; we can make use of a reliable

communication library for communicating over the network. But, this does not change the fact that we reason about database implementation all the way down to the fine granularity of the memory and network semantics.

Specifying database operations In Chapter 2, we give separation logic specifications for the isolation levels snapshot isolation, read committed and read uncommitted (a speciation for an isolation level consists of specifications for each of the database operations) and prove that the specification for snapshot isolation implies the one for read committed, and that the specification for read committed implies the one for read uncommitted. Lastly in Chapter 2, we prove a database implementation correct w.r.t the snapshot isolation specification (by the implications, the implementation also implements the other levels).

Below, we show the specification of the `read` and `write` database operations for snapshot isolation (SI-READ-SPEC and SI-WRITE-SPEC). Observe how these operations are almost identical to the load and store operations for a sequentially consistent heap that we saw in Chapter 1.1: we use a points-to resource $k \mapsto_c ov$ tied to the connection c to the database. The only difference is that the specification for the `write` operation uses a resource $\mathsf{KeyUpdStatus}(c, k, b)$; this keeps track of a boolean indicating whether the current transaction has written to the key k. We use this information for checking for write conflicts at commit time.

SI-READ-SPEC
$$\langle k \mapsto_c ov \rangle \ \texttt{read}\ c\ k\ \langle ov.\, k \mapsto_c ov \rangle$$

SI-WRITE-SPEC
$$\left\langle \begin{array}{l} k \mapsto_c ov * \\ \mathsf{KeyUpdStatus}(c, k, b) \end{array} \right\rangle \ \texttt{write}\ c\ k\ v\ \left\langle \begin{array}{l} ().\, k \mapsto_c \mathsf{Some}\ v * \\ \mathsf{KeyUpdStatus}(c, k, \mathsf{True}) \end{array} \right\rangle$$

In Chapter 2.5, we give the full specification for snapshot isolation with the `start` and `commit` operations. It is in the specifications for the `start` and `commit` operations that the real complexity arises, as in snapshot isolation transactions read from a consistent snapshot of the database created when starting the transaction (it is based on this snapshot we get the points to resources used in the SI-READ-SPEC and SI-WRITE-SPEC rules), and at commit time, we must check if there has been any write conflicts with concurrent transactions.

Now, how we do we know that whichever specification we write to capture snapshot isolation actually captures the intended property? We can verify examples based on phenomea like the write skew example (this is what we do throughout Chapter 2), but then we suffer from the same problem that lead to the invention of transactional consistency models such as the work on dependency graphs by Adya [1999]; Adya et al. [2000]: it does not generalize to all the possible interactions between clients and a database. Thus, or goal is to create a connection from databases verified against the separation logic specifications to the transactional consistency models.

1.2.4 Free Theorems

The work of Birkedal et al. [2021] shows how we can establish trace properties about the interactions between clients and a library as free theorems from the structure of separation logic specifications alone. This is similar to, but not the same as, how Reynolds [1983]; Wadler [1989] showed that we can derive equality properties from the types of polymorphic functions — these are another kind of free theorems.

Structure of separation logic specifications To show what we mean by the structure of separation logic specifications for a library, consider the motivational example used in Birkedal et al. [2021] which is a file system with the following specification:

$$\{\mathsf{closed}\}\ \texttt{open()}\ \{\mathsf{open}\} \quad \{\mathsf{open}\}\ \texttt{close()}\ \{\mathsf{closed}\} \quad \{\mathsf{open}\}\ \texttt{read()}\ \{\mathsf{open}\}$$

The predicates closed and open used in the preconditions and postconditions of the Hoare triples represent the state of the file system. From the structure of the separation logic specifications, we can prove trace properties about any client using the library such as *"a file can only be accessed from the file system if the file system has previously been opened and not subsequently closed"* [Birkedal et al., 2021]. A key property of this technique (which we go into more detail with in Chapter 3) is that it lets us state a property, in form of a trace property, about all the possible executions that come from the interactions between clients and any implementation of library.

A connection to transactional consistency models The observation we make in Chapter 3, is that the transactional consistency model of Crooks et al. [2017], which is proven equivalent to the dependency graphs models of Adya [1999]; Adya et al. [2000], can be expressed as a trace property. We then prove this trace property from the structure of separation logic specifcations for the database operations by using the linearization points that we get from the atomic Hoare triples. The result we obtain in Chapter 3 is then that any database implementation verified against the separation logic speciations for the snapshot isolation, read committed or read uncommitted isolation levels from Chapter 2, always produces executions, no matter how clients interact with the database, that are correct in Adya's model. This result comes for free for database implementation that are verified against the separation logic specification with no additional proof effort and no runtime overhead.

1.3 Programming Language Design: Combining Effect Handlers and Region-Based Memory Management

Effect handlers is a generalization of exception handlers in which the programmer can install handlers that are triggered by performing effects, and resume computation, from where the effect was performed, using continuations that are available to effect handlers. Effect handlers come in two versions: One-shot effect handlers where the continuation is used at most once, and multi-shot effect handlers where there is no restriction on how many times the continuation is used to resume computation.

Region-based memory management is a way to allocate memory in syntactically scoped regions that are freed when the program exists the scope of the region; regions are often associated with the scope of a function. It is a safer way of avoiding automatic memory management with garbage collection, which can be expensive for runtime performance, than using manual memory management in which the programmer must explicitly free memory when it is no longer used; using manual memory management, the programmer is responsible for avoiding user-after-free bugs and leaking memory.

Combining effect handlers and regions-based memory management in one programming language is a question of programming language design: we have to figure out their combined semantics, before we can build reasoning principles with the semantics as foundation in form of proof and type systems. We proceed by introducing the two programming language features in more detail separately.

1.3.1 Effect Handlers

We study effect handlers [Plotkin and Pretnar, 2009] in the context of an ML-like language; to model the operational semantics, we start out with sequential part of HeapLang that we saw in Chapter 1.1. To use effect handlers, we give programmers access to two additional expressions, for handling and for performing effects. The expression **try** e_1 **with** *op* x k. e_2 | **ret** x. e_3 is used to install an effect handler with the name *op* for the expression e_1. Inside e_1, the effect can be performed using **do** *op* v; when performing the effect, control flow is transferred to the handler branch e_2 in the nearest enclosing effect handler installed for *op*, where v is substituted for x in e_2. The continuation k in e_2 is used to resume the computation from where the effect was performed in e_1. When e_1 evaluates to a value v, v is bound to x in the return branch e_3 of the effect handler. In this dissertation, we use *deep* effect handlers where the effect handler is reinstalled around the continuation k, thus the same effect handler can be triggered multiple times, contrary to *shallow* effect handlers that are not reinstalled.

Example: the state effect Let us consider one use case of effect handlers, namely the implementation of global state using the `State` effect [Kammar et al., 2013; Plotkin and Pretnar, 2009]. In Figure 1.1 on line 1, we have a functional implementation of the `State` effect. The `State` effect exposes two

```
1  let hdl_state init f =
2    let comp =
3      try f () with
4      | effect State arg k ->
5        match arg with
6        | Read -> (fun x => (k x) x)
7        | Write x -> (fun () => (k ()) x)
8      | ret x -> (fun () => x)
9    in
10   comp init
11
12 let increment () = do State (Write ((do State Read) + 1))
13
14 let hdl_increment () = hdl_state 0 increment
```

Figure 1.1: Implementation of the state effect.

operations, `Write` for updating the state and `Read` for reading from the state. The user of the effect determines which operation to use through the arguments of the effect which the implementation branches on, see line 5 (operations are notation for sum types, specially `Write x` is notation for `InjR x` and `Read` is notation for `InjL ()`). This particular implementation of the `State` effect is purely functional in that it works by creating a function `comp` that takes as argument the initial value of the global state. Every time the computation `f` for which the `State` effect is installed uses the `Write` or `Read` operation, a new function is created that takes as argument the global state; for the `Read` operation the state is used to resume computation inside `f` with the continuation k, but for the `Write` operation the state is discarded and updated with a new state determined with an argument from `f`. When `f` returns through the return branch of the effect handler, a function which simply returns the return value of `f` is made to end the computation.

Effect handlers are a powerful measure for abstracting away from implementation details of stateful computation; in Figure 1.2, we have an alternative implementation of the `State` effect implemented using a mutable reference. A

```
1 let hdl_state init f =
2   let r = ref init in
3   try f () with
4   | effect State arg k ->
5     match arg with Read -> k !r | Write x -> (r <- x; k ())
6   | ret x -> x
```

Figure 1.2: Implementation of the state effect using a mutable reference.

program such as `increment` on line 12 in Figure 1.1 that uses the `State` effect

can not distinguish between which of the two implementation it uses and will return the same result regardless.

We will return to more applications of effect handlers shortly in Chapter 1.3.3, but first we consider some of the consequences that comes with having effect handles as a feature in programming languages.

Exiting functions more than once The only effect that we have seen so far is a one-shot effect, *i.e.,* the `State` effect does not use the continuation k more than once in the effect branch. Effects that use their continuations more than once, of which we will see a number of in Chapter 4, are referred to as multi-shot effects. Having multi-shot effects as a feature in a programming language makes reasoning especially complicated, for instance, the HT-FRAME rule, that we rely upon in program logics for modular reasoning, becomes unsound. To see why that is, consider reasoning about the function `g` from De Vilhena and Pottier [2021] below:

```
let g f = let r = ref 0 in f (); r := !r + 1; assert (!r = 1)
```

Before calling the function `f`, we have the points-to resource $\mathtt{r} \mapsto 0$ for tracking the state of the newly created reference `r`. Suppose the specification for calling the function `f` is $\{P\}\,\mathtt{f}\ ()\,\{v.\,Q\}$, then under normal circumstances without effect handlers, we use the HT-FRAME rule to frame away the points-to resource $\mathtt{r} \mapsto 0$. Specifically, for reasoning about `f`, we can, by using the specification for `f` and the HT-FRAME rule, prove the subgoal $\{P * \mathtt{r} \mapsto 0\}\,\mathtt{f}\ ()\,\{v.\,Q * \mathtt{r} \mapsto 0\}$ which leaves us with the resources Q and $\mathtt{r} \mapsto 0$ for reasoning about the remaining increment and assertion inside `g`. But this is unsound, if `f` performs a multi-shot effect it can exit more than once making the assertion invalid for all but the first execution that exits `f` due to the increment of `r`. As a concrete example of this behavior, consider the closed example below that uses the `g` function:

```
let hdl_twice f =
  try f () with
  | effect Twice arg k -> k (); k ()
  | ret x -> x

hdl_twice (fun () => g (fun () => do Twice ()))
```

Thus, when reasoning about programs that use effect handlers, the HT-FRAME rule only holds in the presence of one-shot effects, whereas it can only be used in the presence of multi-shot effects if the resources being framed are persistent (unlike points-to resource).

As exemplified above, we can not reuse all the existing reasoning rules when adding effect handlers to a programming language. For reasoning about programs with effect handlers, we need a way to capture what happens to resources when control flow changes, by performing effects and resuming computation with continuations, in a way that lets us abstract over the implementation

details of effect handlers — we want a modular logic where the proof of an effect handler, and the program for which the handler is installed, can be carried out separately.

1.3.2 Modular Reasoning About Effect Handlers

The works by De Vilhena and Pottier [2021] show how we can build a modular program logic for reasoning about one-shot effect handlers (the restriction to one-shot effect handlers was later lifted to allow multi-shot effect handlers by de Vilhena [2022]). This is achieved using the so called *effectfull weakest precondition*:

$$\text{ewp } e \ \langle \Psi \rangle \{\Phi\}$$

The effectful weakest precondition is an extension of the weakest precondition with a protocol Ψ that describes the contract between the effect handler and the program performing the effect (De Vilhena and Pottier [2021] did not yet have named effects thus the protocol in the effectful weakest precondition is related to the nearest enclosing effect handler). The most common type of protocol is the *send-receive protocol*, which in essence works similar to how a Hoare triple can be used to describe the contract between the caller and callee of a function:

$$!\vec{x}(v_1)\{P\}.?\vec{y}(v_2)\{Q\}$$

Above, the *send-receive protocol* states that the argument when performing the effect v must be equal to v_1, and the precondition P must hold ($\vec{x}$ can be a number of existentially quantified variables used in other entities of the protocol), the implementation of the effect handler then gets to assume the resources in P. When the effect handler implementation wants to resume computation using the continuation, it must be proven that the postcondition Q holds and that the continuation is called with an argument equal to v_2 ($\vec{y}$ are universally quantified variables that can be used in v_2 and Q).

Using another type of protocol, namely the *composition* protocol, which composes two existing protocols with the + symbol, we can define a protocol that captures the contract between an effect handler that implements the `State` effect and a program performing the effect:

$$\begin{aligned} \mathit{STATE}\ \mathit{St} \triangleq \quad & !x(\texttt{Read})\{\mathit{St}\ x\}.?(x)\{\mathit{St}\ x\} \\ & + !x\ x'(\texttt{Write}\ x')\{\mathit{St}\ x\}.?(())\{\mathit{St}\ x'\} \end{aligned}$$

The predicate St tracks the contents of the global state; regardless of whether the `State` effect is implemented using a functional approach as in Figure 1.1, or using a mutable reference as in Figure 1.2, then the protocol captures the read and write semantics of the `State` effect: `Read` returns the contents of the state and leaves it unchanged, and `Write` updates the state and returns the unit value.

1.3.3 Application of Effect Handlers

In the paper titled *"Concurrent System Programming with Effect Handlers"* by Dolan et al. [2017] a number of applications of effect handlers was shown, among those, how to effectively implement cooperative concurrency (with no parallelism). Cooperative concurrency implemented using effect handlers, which we will refer to as "asynchronous computation" going forward because a `yield`-like operation is often omitted as part of the implementation, is a recurring example that has been studied in a number of papers on reasoning about effect handlers [De Vilhena and Pottier, 2021; van Rooij and Krebbers, 2025].

In Figure 1.3, we have included an implementation of asynchronous computation matching the one found in the aforementioned papers. The implementation is centered around the `fulfill` function and a queue `q` of threads that are ready to be scheduled. The `fulfill` function takes as argument a task `e` and

```
1  let new_promise () = ref (Waiting (list_nil ()))
2
3  let next q = if queue_empty q then () else (queue_pop q) ()
4
5  let run main = region (
6    let q = queue_create () in
7    let fulfill = (fun p e =>
8      try e () with
9      | effect AC request k ->
10       (match request with
11       | Async e ->
12         let p = new_promise () in queue_push q (fun () => k p);
13         fullfill p e
14       | Await p ->
15         (match !p with
16         | Done v -> k v
17         | Waiting ks -> p <- (Waiting (list_cons k ks)); next q))
18     | ret x ->
19       (match !p with
20       | Done v -> assert false
21       | Waiting ks ->
22         list_iter (fun k => queue_push q (fun () => k v)) ks;
23         p <- (Done v);
24         next q)
25   ) in
26   let p = new_promise () in
27   fullfill p main)
```

Figure 1.3: Implementation of asynchronous computation.

a promise `p` — promises are implemented as a reference to a list of threads that are waiting on the result of the task (this list, like the queue `q`, holds the continuations of threads). When the task `e` is being run as a thread (line 8), `fulfill` installs the asynchronous computation effect `AC` which exposes two operations to the thread. The first operation is the `Async e` operation (line 11) for asynchronously computing a task `e`, this operation returns a promise.

The second operation that the `AC` effect exposes is the `Await p` operation (line 14), for awaiting on the result of a promise; the thread that uses the `Await p` operation is added to the list of waiting threads that the promise points to. When a computation finishes (line 18), all the threads waiting on the promise is added to the queue of threads that are ready to run (they are given the result of the task that the promise corresponds to as argument), where after the head of the queue is run next.

In Chapter 4, we revisit asynchronous computation where we show how all the global allocations, *e.g.,* the references used to represent promises, can be allocated in memory regions which is more efficient than the current strategy of relying on garbage collection for memory management.

1.3.4 Using Region-based Memory Management with Effect Handlers

Region-based memory management is offered as an alternative to garbage collection in a few selected programming languages; starting with the ML kit [Birkedal et al., 1996; Tofte and Talpin, 1997; Tofte et al., 2001, 2004], where it was shown how bounded regions can use the call stack memory of a function, and regions that are unbounded should resort linked-list structures on the heap, which influenced languages such as Cyclone [Grossman et al., 2002; Jim et al., 2002] and Rust. OxCaml [Georges et al., 2025; Lorenzen et al., 2024] is an extension of OCaml with, among others features, region based memory management. OCaml does support one-shot effect handlers, but the combined semantics with regions in OxCaml is not formalized. The programming language Flix [Lutze et al., 2023; Madsen, 2022] has a region construct and supports multi-shot effect handlers but their combined use is undefined [Madsen, 2026].

Semantics To use effect handlers with region-based memory management, we must create a mathematical definition of their combined semantics: in Chapter 4, we create an operational semantics for a new language Yarrow, an ML-like language with one-shot and multi-shot effect handlers and region-based memory management. Making a precise mathematical definition of the combined semantics in a new language is non-trivial for some of the same reasons that make effect handlers complex to reason about: because functions using multi-shot effect handlers can exit more than once, regions can also be exited more than once. This breaks the underlying assumptions of previous region-based memory management systems that free memory when the scope of a region (often tied to the scope of function) ends. There are a number of execution models for effect handlers (see Chapter 4.8) — we build on the one of OCaml by Sivaramakrishnan et al. [2021] that explicitly manipulates the stack which makes it amenable to reasoning about regions as well.

Modular reasoning Previous work on modular reasoning about regions in program logics [Georges et al., 2025] rely on well-bracketed control flow Timany et al. [2024b] (a function always returns before its caller can resume execution) for reclaiming resources used for memory in regions. Due to effect handlers, this is no longer an assumption we can rely on. In Chapter 4, we create the Yarrow Logic (YL), a modular Iris based program logic for creating proofs of correctness, w.r.t to the operational semantics of Yarrow, for programs using effect handlers with region-based memory management. The logic uses points-to resource for memory in regions (with carefully inserted reclamation points) and uses the idea of protocols from De Vilhena and Pottier [2021] (but redesigned to take into account regions).

Semantic typing and a runtime implementation Creating a type system for Yarrow is future work. Semantic typing Timany et al. [2024c] is a technique in which type systems can be proven sound by giving types and the typing judgement semantic meaning, *e.g.,* by making an interpretation in a program logic. This technique is a popular application of Iris based program logics, and the program logic by de Vilhena [2022]; De Vilhena and Pottier [2021] has already been used to create type systems for programming languages with effect handlers [de Vilhena and Pottier, 2023; van Rooij and Krebbers, 2025]. The YL could thus be used as a foundation for a type system for Yarrow. One upside of semantic typing versus the classic syntactic progress and preservation proof, is that semantic typing allows us to prove by hand that safe libraries using unsafe features (notice these are syntactically ill-typed and cannot necessarily be algorithmically checked as part of the type system) can be hidden behind type abstractions and composed with syntactically well-typed programs.

Naturally, a runtime implementation of Yarrow is an interesting line of future work. As the operational semantics of Yarrow is based on the design of effect handlers in OCaml Sivaramakrishnan et al. [2021], OCaml could serve as a starting point for such an implementation.

1.4 Academic Contributions

In this section, we list the academic contributions that make up the remaining chapters of the dissertation and explicitly state the personal contributions. We conclude the section by providing links to the Rocq formalizations.

1.4.1 Publications and Manuscripts

This dissertation is based upon, *i.e.,* it reuses text from, the following list of publications and manuscripts:

Publications

- [Alnor Mathiasen et al., 2025] Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. Reasoning about weak isolation levels in separation logic. *Proc. ACM Program. Lang.*, 9(ICFP), August 2025. doi: 10.1145/3747515. URL https://doi.org/10.1145/3747515

Manuscripts

- [Mathiasen et al., 2026a] Anders Alnor Mathiasen, Amin Timany, and Lars Birkedal. Verifying isolation levels of database implementations for free using separation logic, 2026a. URL https://arxiv.org/abs/2607.15877

- [Mathiasen et al., 2026b] Anders Alnor Mathiasen, Amin Timany, and Lars Birkedal. Yarrow: Reconciling effect handlers and region-based memory management, 2026b. URL https://arxiv.org/abs/2607.15876

During my studies, I (the author of this dissertation) also co-authored the publication below, but this is not included as part of this dissertation:

- [Mathiasen and Pavlogiannis, 2021] Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of Andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434315. URL https://doi.org/10.1145/3434315

Below, we list the origin of the text in each of the chapters of this dissertation:

- Chapter 2: This chapter is a verbatim reproduction of Alnor Mathiasen et al. [2025].

- Chapter 3: This chapter is a verbatim reproduction of Mathiasen et al. [2026a].

- Chapter 4: This chapter is a verbatim reproduction of Mathiasen et al. [2026b], except appendix A is included as part of Chapter 4.7.

1.4.2 Personal Contributions

For the paper in Chapter 2, I am the main author. The parts about snapshot isolation were developed and formalized in collaboration with Leon Göndelmann and with contributions from Leon Ducruet. The rest of the paper is based on my ideas and I carried out the formalization for these parts. I wrote the whole paper with comments from the co-authors Amin Timany and Lars Birkedal who were in advisory roles.

For the paper in Chapter 3, I am the main author, co-authors Amin Timany and Lars Birkedal were in advisory roles; the paper is based on my idea. Specifically, I carried out the formalization and wrote the paper with comments from my co-authors.

Likewise, for the paper in Chapter 4, I am the main author, co-authors Amin Timany and Lars Birkedal were in advisory roles; the paper is based on my idea. Specifically, I carried out the formalization and wrote the paper with comments from my co-authors.

1.4.3 Formalization

The formalization for each of the chapters is carried out in the Rocq proof assistant and can be accessed using the links below. The formalization include everything from the operational semantics', example code and proofs of all the mathematical statements in the chapters.

- Chapter 2: `https://zenodo.org/records/15626657`
- Chapter 3: `https://zenodo.org/records/21497185`
- Chapter 4: `https://zenodo.org/records/21497480`

As the trusted computing base of the Rocq proofs assistant is very small, we gain high assurance that the results in this dissertation are correct.

Part II

Publications

Chapter 2

Reasoning about Weak Isolation Levels in Separation Logic

Abstract

Consistency guarantees among concurrently executing transactions in local- and distributed systems, commonly referred to as isolation levels, have been formalized in a number of models. Thus far, no model can reason about executable implementations of databases or local transaction libraries providing weak isolation levels. Weak isolation levels are characterized by being highly concurrent and, unlike their stronger counterpart serializability, they are not equivalent to the consistency guarantees provided by a transaction library implemented using a global lock. Industrial-strength databases almost exclusively implement weak isolation levels as their default level. This calls for formalism as numerous bugs violating isolation have been detected in these databases.

In this paper, we formalize three weak isolation levels in separation logic, namely *read uncommitted*, *read committed*, and *snapshot isolation*. We define modular separation logic specifications that are independent of the underlying transaction library implementation. Historically, isolation levels have been specified using examples of executions between concurrent transactions that are not allowed to occur, and we demonstrate that our specifications correctly prohibit such examples. To show that our specifications are realizable, we formally verify that an executable implementation of a key-value database running the multi-version concurrency control algorithm from the original snapshot isolation paper satisfies our specification of snapshot isolation. Moreover, we prove implications between the specifications—snapshot isolation implies read committed and read committed implies read uncommitted—and thus the verification effort of the database serves as proof that all of our specifications are realizable. All results are mechanized in the Rocq proof assistant on top of the Iris separation logic framework.

2.1 Introduction

Transactions, defined as a set of operations considered as a single unit, is a common concept implemented by most modern databases and can also be encountered in local transactional memory systems. There exists multiple isolation levels for describing the allowed interaction between concurrently executing transactions. The strong isolation level of serializability [Papadimitriou, 1979] often serves as an introduction to isolation levels. This level has consistency guarantees equivalent to what one would achieve in a transactional library protected by a global lock acquired at start time and released at commit time by the transactions. While serializability can be a useful property, it is often too strong for the desired use case and will negatively impact the throughput of the system. In contrast to serializability, weaker isolation levels allow for more concurrency and higher throughput. In fact, most modern commercial database systems, e.g., MySQL, Microsoft SQL, Oracle and PostgreSQL, have a weaker isolation level than serializability as default (Microsoft SQL, Oracle and PostgreSQL uses read committed [MicrosoftSQL, 2025; Oracle, 2025; PostgeSQL, 2025] which we specify in this paper). With increased concurrency comes weaker consistency guarantees that by nature are harder to capture and reason about. As numerous bugs have been found in commercial database implementations violating isolation [Jepsen, 2025], it is highly important to formally verify database implementations and their clients.

Existing formalizations of isolation levels A number of different formalizations have been developed to capture and reason about isolation levels. Historically, in the ANSI SQL standard of 1992, isolation levels were defined using examples of executions between concurrently executing transactions, so-called phenomena, which must be prohibited by the particular isolation level. The isolation levels of read uncommitted, read committed, repeatable read and serializability were defined using a phenomenon for each level except read uncommitted. This was later critiqued for being imprecise, which led to the invention of snapshot isolation and improved phenomena by Berenson et al. [1995]. With the phenomena as a basis, a number of different models have been developed to reason about isolation levels. Predominantly, models have been based on dependency graphs [Adya, 1999; Adya et al., 2000], operational semantics [Cerone et al., 2015b; Crooks et al., 2017; Kaki et al., 2017; Xiong, 2020] or abstract executions [Burckhardt et al., 2012; Ketsman et al., 2023]. While the mentioned models can be used to define the desired consistency guarantees for transactions, in the form of isolation levels, they can not be applied to verify executable implementations of databases, transactional libraries and clients thereof. Recently, the separation logic framework Iris [Jung et al., 2018] has been used in Chang et al. [2023] to give a modular specification for serializability in the context of a key-value store. Separation logic is well known for its ability to specify various properties, using Hoare triples, and being able

to reason about executable implementations [Gondelman et al., 2021; Nieto et al., 2022, 2023; Sharma et al., 2023]. The specification for serializability in Chang et al. [2023] leverages that serializability is a strong property: A concise specification is given for a run operation that bundles together the start operation, commit operation and intermediate operations of a transaction into a single specification — something which is unfeasible for weaker isolation levels (this is discussed in Chapter 2.8). The argument of using separation logic to capture isolation levels and reason about executable code has yet to be completed by showing that separation logic can capture weak isolation levels.

Specifying weak isolation levels in separation Logic The main contributions of this paper are formal separation logic specifications of weak isolation levels: read uncommitted, read committed and snapshot isolation. Together with serializability and repeatable read, these levels constitute the common levels found in commercial systems (repeatable read is identical to serializability except for its behavior regarding the predicate-read operation [Adya et al., 2000] which we do not model). To demonstrate that our specifications capture the desired isolation levels, we have, for the phenomena in the literature, created and proven examples of client transactions running against libraries adhering to our specifications which show that the phenomena are prohibited. We have chosen to verify the phenomena from the literature as clients because (1) They are concise examples that exercises edge cases, which a larger client implementation may not necessarily cover; and (2) these client-examples are used as the only specification in the SQL standard, and this is what industry databases use as specification. In addition to the phenomena, we have also verified a bank transfer example that represented a common use case for transactions (Chapter 2.5). Furthermore, we prove implications between our specifications, snapshot isolation implies read committed and read committed implies read uncommitted, which is consistent with the literature [Adya, 1999]. Implications between separation logic specifications are unusual, as they establish that any implementation of the assumed specification also implements the implied specification.

Realizing specifications To show that our specifications are realizable by concrete implementations, we implement a database and verify that it meets our snapshot isolation specification. Together with the implication proofs between the specifications, this serve as a proof that all of our specifications are realizable. In more detail, we have used the distributed separation logic of Aneris [Krogh-Jespersen et al., 2020] to formally verify an in-memory single-node key-value store with support for transactions. In fact, we have implemented and verified the algorithm from the original snapshot isolation paper by Berenson et al. [1995]. The implementation uses multi-version concurrency control where for each key all previous updates are stored and ordered by timestamps. In

addition to the operations in the API of the database, we have implemented utility functionalities on top of these operations in the form of wait and run operations. The wait operation returns once it has observed the key-value pair it has been given as argument, while the run operation wraps start and commit operations around a transaction body as in Chang et al. [2023]. We verify specifications for the wait operation for all the isolation levels we are considering, while we argue that only snapshot isolation is strong enough to have a concise specification for the run operation. Our implementations are written in OCaml and transpiled to AnerisLang for mechanized verification in the Rocq proof assistant.[1] In fact, all the results in this paper have been mechanized in Rocq. Aneris is an instantiation of the Iris separation logic framework with an OCaml like language (AnerisLang) and semantics for unreliable network communication. We emphasize that while we have used Aneris for the verification and implementation of a database, our specifications are independent of Aneris and can be used with other instantiations of Iris/separation logics.

Challenges One of the key challenges of this work is to capture the isolation levels by modular separation logic specifications, with formal specifications for each of the individual operations on the key-value store — traditionally, isolation levels have not been formulated at this level of granularity with separate specifications for each of the operations: Existing transactional models either have a global view of all transactions, or are formulated with a high-level idealized operational semantics far from the semantics used to model the programming language on which we build in this paper. We emphasize that one of the advantages of our approach, using separation logic specifications, is modularity in because it enables one to reason formally about client programs that (possibly among other libraries) make use of the key-value store as a library. Another key challenge is to prove the implications between the specifications of the three isolation levels. The specifications are highly modular as they keep the logical resources abstract by hiding the definitions. In the implication proofs, we must construct separation logic resources used in the implied specification, given the abstract separation logic resources of the assumed specification, in a way that lets us show the formal specifications for each of the individual operations of the implied specification. That is while hiding the fact that the newly constructed logical resources are defined in terms of the assumed logical resources. Lastly, to verify that the multi-version concurrency control algorithm of Berenson et al. [1995] implements our separation logic specifications, in the context of a distributed system, requires non-trivial and substantial proof effort (more than 5000 lines of Rocq proof code), especially since our specification for snapshot isolation includes evidence for unsuccessful commits in the form of a conflict between concurrent transactions expressed as a resource in the postcondition of the commit specification.

[1]The Coq proof assistant has recently been renamed to Rocq.

Contributions In summary, our paper makes the following contributions:

1. The first modular separation logic specifications for weak isolation levels, in particular read uncommitted, read committed and snapshot isolation. The specifications are independent of the underlying algorithm and whether the system is a local or distributed system, and we provide examples that show the specifications prohibit the phenomena of the literature.

2. Formal proofs showing that the specification of snapshot isolation implies the specification of read committed, and that the specification of read committed implies the specification of read uncommitted.

3. Implementation and verification of an in-memory single-node key-value database with snapshot isolation. The implementation is based on the original multi-version concurrency control snapshot isolation algorithm, and is the first formally verified executable database supporting transactions.

This gives us, for the first time, one unified logic in which we can verify clients and database implementations for transactions with weak isolation levels. That is all while enjoying the modularity of higher-order separation logic such as the ability to combine code using transactions with other verified libraries (something which existing work do not address). All results in this paper are mechanized in the Rocq proof assistant using the Iris separation logic.

Structure of the paper The rest of this paper is structured as follows: In Chapter 2.2, we present our specification for read uncommitted using examples we have verified. Chapter 2.2.2 considers the proof of a particular example in detail using the read uncommitted specification. In Chapter 2.3, we present our specification for read committed and the examples we have proven using the read committed specification. In Chapter 2.4, we prove the specification for read committed implies the specification for read uncommitted. In Chapter 2.5, we present our specification for snapshot isolation. Here, we also present the examples we have proven using the snapshot isolation specification. In Chapter 2.6, we prove the specification for snapshot isolation implies the specification for read committed. In Chapter 2.7, we explain the verification of our database implementation with respect to our specification for snapshot isolation. In Chapter 2.8 we discuss related work, and in Chapter 2.9 we talk about future work before concluding. We will refer to Appendix 2.10 for presentation of a utility wait operation for all the isolation levels and a utility run operation for snapshot isolation together with a discussion of why a run operation is impractical for read uncommitted and read committed.

2.2 Specifying Read Uncommitted

In this section, we first recall the informal description of read uncommitted, and then present our modular separation logic specification in Chapter 2.2.1. This is done using the so-called *read uncommitted data* and *read own data* examples which we have verified using our specifications. Afterwards, in Chapter 2.2.2, we show in some detail how we can use the specification to prove safety and correctness of the concurrently executing transactions in the *read uncommitted data* example. We present the proof to help the reader get acquainted with using the style of specifications presented throughout the paper.

Read uncommitted Read uncommitted is the weakest isolation level. The level has mostly theoretical value, and it is often used in transactional models to establish a baseline, of definitions and structure, which we can use to define stronger levels. We follow the same approach. The only guarantees one gets from using read uncommitted is that data being read has been written by some transaction (captured by the *read uncommitted data* example in Figure 2.1) , and if a read comes after a write from the same transaction, then the read will be reading from this write (captured by the *read own data* example in Figure 2.2).

The *read uncommitted data* example (we ignore the invariant in blue for now) consists of two transactions (separated by the vertical bars), which are executed concurrently on two different nodes in a distributed system, alternatively threads in a local system, in which there is also a server with the key-value store. Notice that the example includes an assert statement, which crashes if the expression within it does not hold. We use assert because it makes it easy to state (in a logic-independent way) that an intended property holds. In this example, the assert statement expresses that the reading transaction will read nothing or the value written by the writing transaction, even though the latter does not commit. This is possible in read uncommitted as there is no guarantee that transactions will read committed data; it is only guaranteed that the data has been written by another transaction — independently of whether this transaction aborted or committed.

The *read own data* example follows a similar structure with two transactions. Here the assert statement expressed that no matter when the write of the first transactions is scheduled, the second transactions will always read the value previously written by itself.

Some formalizations of read uncommitted, e.g., Adya [1999], imposes restrictions on the order in which versions are installed inside the key-value store. As separation logic abstracts away from implementation details, this is not a requirement we can or want to impose. This choice is in line with the work of Crooks et al. [2017] which takes a client centric approach.

$$
\begin{array}{l||l}
\texttt{start} & \texttt{start} \\
\texttt{write}\ x\ 1 & \textbf{let}\ v_x = \texttt{read}\ x\ \ \textbf{in} \\
\texttt{loop} & \texttt{assert}(v_x = \mathsf{None} \lor v_x = \mathsf{Some}\ 1) \\
 & \texttt{commit}
\end{array}
$$

$$Inv \triangleq \exists V,\ {}_{\text{RU}}.x \mapsto V\ \ * (V = \emptyset \lor V = \{1\})$$

Figure 2.1: Read uncommitted data.

$$
\begin{array}{l||l}
\texttt{start} & \texttt{start} \\
\texttt{write}\ x\ 1 & \texttt{write}\ x\ 2 \\
\texttt{commit} & \textbf{let}\ v_x = \texttt{read}\ x\ \ \textbf{in} \\
 & \texttt{assert}(v_x = \mathsf{Some}\ 2) \\
 & \texttt{commit}
\end{array}
$$

Figure 2.2: Read uncommitted data.

2.2.1 Separation Logic Specifications

Our formal specifications of the read uncommitted operations are defined using the Aneris program logic [Krogh-Jespersen et al., 2020]. Aneris is a higher-order distributed separation logic, based on the Iris framework [Jung et al., 2018], fully mechanized in the Rocq proof assistant.[2] Our specifications are written using Hoare triples $\{P\}\, e\, \{v.\, Q\}^{ip}$. A Hoare triple asserts that given a machine state satisfying the precondition P, then the execution of the expression e is safe and, moreover, if the execution terminates then Q holds for the resulting machine state when the return value is bound to v. In Aneris, Hoare triples are annotated with the IP-address of the node on which the expression is executing, but we often omit this annotation when the IP-address is not of importance or for presentation purposes. In fact, for most parts of this paper (except for the proof of the snapshot isolation implementation in Chapter 2.7), one can just pretend that the logic is the standard Iris logic; no special knowledge of Aneris is required to understand the paper. The propositions in the precondition and postcondition of the Hoare triples are often referred to as resources, since Aneris is a separation logic with a concept of ownership. We recall that the separating conjunction $P * Q$ only can be proven from a resource that can be split into two resources satisfying P and Q respectively. The magic wand $P \mathrel{-\!\!*} Q$ is closely related to the separating conjunction: in combination with exclusive ownership

[2]While we happened to use Aneris, we could have used another distributed separation logic based on the Iris framework such as Grove [Sharma et al., 2023], see the discussion in Chapter 2.8.

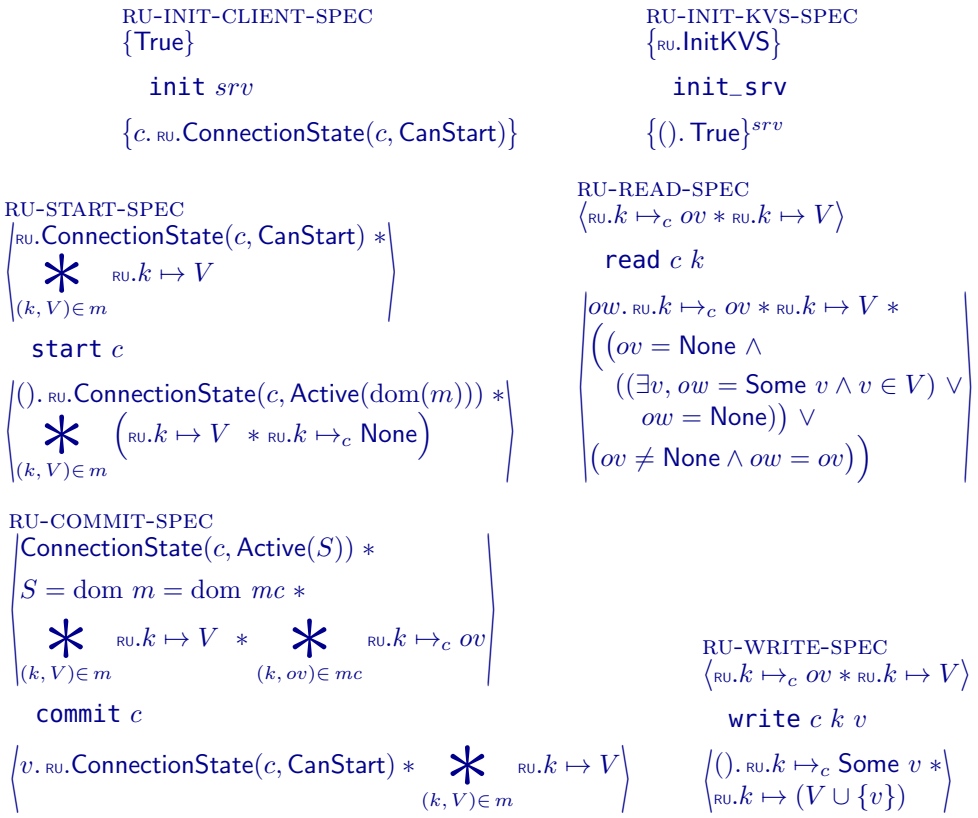


Figure 2.3: Specification for read uncommitted (RUSpecs).

of P, then $P \twoheadrightarrow Q$ entails Q. Some resources denote pure knowledge and do not assert exclusive ownership; this holds in particular for all persistent predicates, which are those satisfying that $\square\, P \vdash P$, where $\square$ is the so-called persistence modality which, intuitively, takes a predicate and forgets about the resources exclusively owned by the predicate. Persistent predicates can thus always be freely duplicated. For instance, Hoare triples are defined using the persistent modality, to allow us to use our specifications an unbounded number of times.

We now begin to describe the specifications of the operations of read uncommitted, which can all be found in Figure 2.3. We motivate and introduce the concepts used in the specifications along the way. For notation purposes, as we will be presenting specifications for other isolation levels with similar notation, we use the color blue and the prefix RU. when presenting the read uncommitted specification. Later, we will use similar notation for the other isolation levels.

Init client operation The specifications for all our isolation levels are centered around exclusive points-to resources. This is a popular choice in separation logic as it allows specifications only to require knowledge of a single

key/location as opposed to a larger state. We will be working with two types of exclusive points-to resources: A *local points-to resource* per active transaction, denoted $\text{RU}.k \mapsto_c ov$ for read uncommitted, which asserts that the key k points to an optional value ov for the transaction active on connection c, and a *global points-to resource*, denoted $\text{RU}.k \mapsto V$ which asserts that the key k points to a state consisting of a set of values V, which captures the (global) state of the key-value store. A client only gets to make requests to the key-value store once a connection has been established with the provider of the key-value store, whether it is a local connection or happens over the network. A connection can be established by the client calling the `init` method, whose specification RU-INIT-CLIENT-SPEC is in Figure 2.3. A client can always try to establish a connection with the key-value store (the precondition is simply True), and in the postcondition, the client gains proof that it has a connection in the form of the resource ConnectionState(c, CanStart). The `init` method keeps trying to establish a connection to the server with IP-address *srv* when called, thus if no key-value store is available on the IP-address, the call will not terminate. If a client has multiple connections to different key-value stores, the points to resources will also take as argument the IP-address of the key-value store to which they belong. As long as there is only one active key-value store, the IP-address argument is implicit.

Start operation As potentially many clients concurrently want to start their transactions, the resources capturing the key-value store state will often reside in an invariant. Invariants are propositions that hold at every step of execution once they have been established. An invariant holding a resource P is denoted $\boxed{P}$. [3] In Iris, and hereby Aneris, one can only open invariants across atomic expressions (expressions which in the operational semantics can step to a value using a single step). For this reason, we use *logical atomic triples* instead of standard Hoare triples to specify the transaction operations (for notation, logical atomic triples uses angle brackets as opposed to the curly brackets of ordinary Hoare triples). We refer to Jung et al. [2015] for a detailed description of logical atomic triples; in this paper, we simply treat them as special Hoare triples that allow us to open invariants around them. In Figure 2.4, we display the rule used to open an invariant around atomic Hoare triples.

$$\frac{\text{INV-ATOMIC}\quad \vdash \langle R * P\rangle\, e\, \langle v.\, Q * R\rangle}{\boxed{R} \vdash \langle P\rangle\, e\, \langle v.\, Q\rangle}$$

Figure 2.4: Invariant rule for atomic Hoare Triples.

[3] A reader familiar with Iris will notice that we have omitted invariant names, and in later parts of the paper masks, to make the presentation less technical.

With the rule Inv-atomic, we can use the resources in an invariant to prove the precondition of an atomic Hoare Triple, and we can use the resources in the postcondition of the atomic Hoare Triple to reestablish the invariant.

Now, the ru-start-spec rule takes as precondition the connection state together with global points-to resources for all the keys on which the transaction wants to perform reads and writes – these are the keys in the map m in the notation. As the key-value store state, in the form of global points-to resources, can be shared among many clients in an invariant, the start operation is specified using a logical atomic triple. The postcondition of the start operation states that the connection is active for the keys in the domain of m (this information will be used later at commit time) and gives back the global points-to resources together with fresh local points-to resources.[4]

Now, we could have elided an explicit `init` operation and made it part of the `start` operation to simplify the specification. This would, however, imply initialization overhead (cache and network-connection creation) on a per-transaction basis which is suboptimal.

Read and write operation Having started a transaction with the start operation, an active transaction can perform read and write operations using its newly gained local points-to resources. The read operation requires the global and local points-to resources for the key in its precondition. If the active transaction has already written to the key, this value will be read straight from the local points-to resource. Otherwise, the value being read can be None or come from the global state, i.e., the value will be from the set of values V pointed to by the global points-to resource. As for the read operation, the precondition of the specification for the write operation requires global and local points-to resources for the key that is being written to. As we are specifying read uncommitted where transactions are allowed to read uncommitted data, both the global and local points-to resource is updated. Note, in particular, that the global points-to resource gets updated immediately for other transactions to read, even if the writing transaction does not commit.

At this point, it can be tempting to create a stronger specification. For instance, a meaningful optimization would be to cache a local read and update ov to ow when ov is none for local points-to resource in the read specification. But if we enforce such an optimization in the specification, then we are specifying something stronger than read uncommitted (not all implementations of read uncommitted have this optimization). Do notice that any implementation doing the optimization can implement our specification, it will simply pick ow from the set V on a subsequent read if ov remains None.

[4]Separation-logic aficionados who are familiar with the frame rule may wonder why the global points-to resources are included unchanged in both the precondition and the postcondition; we have decided to include them (both for start and commit) in order to be able to prove formally that read committed implies read uncommitted, see Chapter 2.4.

Commit operation At commit time, the connection state, local and global points to resources are collected in the precondition of the RU-COMMIT-SPEC rule. The domain conditions ensure that local and global points-to resources are available for all the keys on which the transaction was started. No matter if the commit is successful or not (the return value of commit is a boolean), the state of the key-value store remains the same in the postcondition of the RU-COMMIT-SPEC, as all the updates made by the transaction have already been propagated at the time of the write operation, cf. RU-WRITE-SPEC, to the global points-to resources. Thus, the job of the commit specification in read uncommitted is simply to collect the local points-to resources and change the connection state.

Init KVS operation and the complete specification Initializing the key-value store with RU-INIT-KVS-SPEC requires the $\text{RU}.\mathsf{InitKVS}$ resources which can not be obtained from using the per operation specifications in Figure 2.3. Likewise, initial global points to resources must be obtained before one can start using the specifications in Figure 2.3. Thus, a complete specification for an isolation level must provide all the resources we need in order to get started with using the per operation specifications. Below, in (2.1), we have defined exactly what the complete read uncommitted specification RUSpec is. The complete specification RUSpec is stated using the Iris update modality. The Iris update modality $\Rrightarrow P$ states that initialization of resources or updates to currently owned resources can be made to satisfy P. In the context of RUSpec, this means we can initialize the resources that constitute the specification. We will continue with explaining each of the components RUSpec consists of.

$$\mathsf{RUSpec} \triangleq \Rrightarrow \text{RU}.\mathsf{InitKVS} * \mathop{\Large *}_{k \in \mathsf{Keys}} \text{RU}.k \mapsto \emptyset * \mathsf{RUSpecs} * \text{RU}.\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} \tag{2.1}$$

First, we have the resource $\text{RU}.\mathsf{InitKVS}$ to initialize the key-value store as mentioned. Next, for every key we have $\text{RU}.k \mapsto \emptyset$ with each key pointing to the empty set as the initial state. RUSpecs is defined as all the specifications from Figure 2.3, concretely it is the separated conjunction between each of the Hoare Triples in Figure 2.3. Lastly, we have a global invariant $\text{RU}.\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$, which we have not yet seen a use case for. The global invariant is needed to be able to prove two properties which we have named the *exclusion* and *creation* properties. We will first motivate and then explain these properties next.

The exclusion and creation properties As already mentioned, the state of key-value store, in the form of global points-to resources, will be shared using an invariant when concurrent access is needed (see the following subsection for an example). The invariant has to be of a form that captures the key-value store states that are created by the transactions. Not all states will be valid at any

given time, for instance, if a value has been observed it can not disappear later. To capture such observations, our specifications (for all our isolation levels) come with a *seen* resource, which works together with the global points-to resource. The $\mathsf{Seen}(k, V)$ resource for read uncommitted intuitively captures the observation of having seen the key-value store in the state where k was pointing to the set of values V. Clients can use this knowledge to exclude absurd states of an invariant. The exclusion of key-value store states based on observations happens through the exclusion property (2.2).

$$\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \text{RU.}\mathsf{Seen}(k, V') * \text{RU.}k \mapsto V \vdash \Rrightarrow \text{RU.}k \mapsto V * V' \subseteq V \tag{2.2}$$

$$\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \text{RU.}k \mapsto V \vdash \Rrightarrow \text{RU.}k \mapsto V * \text{RU.}\mathsf{Seen}(k, V) \tag{2.3}$$

The essence of the exclusion property is that given an observation $\text{RU.}\mathsf{Seen}(k, V')$ and the current state of a key $\text{RU.}k \mapsto V$, it must be the case that the current state is the same as the observation or the result of having applied updates to the observation, i.e., $V' \subseteq V$ holds. The seen resources are created by the creation property (2.3). To be able to use the creation and the exclusion properties we must have access to the global invariant of the read uncommitted key-value library which is therefore a part of the complete specification RUSpec. [5]

We remark that in the exclusion property (2.2) and the creation property (2.3), the $\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ and $\text{RU.}\mathsf{Seen}(k, V')$ resources are not included on the right-hand sides but can be found on the left-hand sides of the entailment; it is not necessary to include them on the right since both of these predicates are persistent and hence can be freely duplicated prior to applying the exclusion property rule.

2.2.2 Proof of Read Uncommitted Data Example

In this section, we show how to use the specifications from Figure 2.3 to prove the *read uncommitted data* example shown in Figure 2.1. What we mean by "proving the read uncommitted data example" is that we prove a specification (a Hoare triple) for this example. By adequacy of the Aneris logic, this then means that the assert shown in Figure 2.1 will not fail. In other words, by proving a specification for the example, we show that the property expressed in the assert statement holds for any interleaving.

We begin by using the setup rule (2.1). As the key-space for the example consists only of x, the setup rule gives us $\text{RU.}x \mapsto [\,]$. This resource is then used to create the client invariant $\boxed{\exists V,\ x \mapsto V\ * (V = \emptyset \vee V = \{1\})}$, also shown in Figure 2.1 (all the examples in this paper uses invariants but we only state them when it is relevant for the presentation). The $\text{RU.}\mathsf{InitKVS}$ resource from the

[5] The Seen resources can also be defined as a standalone library that will assume the remaining read uncommitted specification.

setup is used to initialize the key-value store using RU-INIT-KVS-SPEC. We then prove each of the two client transactions separately — this is possible in Aneris as long as one distributes resources appropriately among the clients prior to doing the individual proofs. As invariants are persistent and thus duplicable resources, we can create two copies of $\boxed{\exists V,\ x \mapsto V\ * (V = \emptyset \lor V = \{1\})}$ and use one for each of the individual clients proofs. The sharing of invariants is only safe as long as each client owning an invariant uses it safely (this is enforced in Iris). Namely, the resources in an invariant can be accessed around operations specified with atomic Hoare triples or atmoic expressions where we use the former. For both transactions it is the case that their proofs are bootstrapped using RU-INIT-CLIENT-SPEC, which means that in each case, we have a $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$ resource for their connection c (each client have a unique connection). We will now proceed by proving the clients transactions one at a time.

Write-transaction The transaction is started using the start operation. To satisfy the precondition of the RU-START-SPEC rule, we provide $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$ and the global points-to resources from the client invariant, i.e., $_{\text{RU}}.x \mapsto V$ for a fresh set of values V for which $V = \emptyset \lor V = \{1\}$ holds. By the RU-START-SPEC rule we then obtain the resources $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\})) * {}_{\text{RU}}.x \mapsto V * {}_{\text{RU}}.x \mapsto_c \mathsf{None}$. The global points-to resource $_{\text{RU}}.x \mapsto V$ is used to close the client invariant while we hold onto the rest of the resources. We then use the RU-WRITE-SPEC rule for the `write` x 1 operation. Again, we must open the invariant to gain $_{\text{RU}}.x \mapsto V'$, for a fresh set of values V' for which $V' = \emptyset \lor V' = \{1\}$ holds. Now, $_{\text{RU}}.x \mapsto V'$, together with the local points-to resource $_{\text{RU}}.x \mapsto_c \mathsf{None}$ we obtained using the RU-START-SPEC rule, is provided to the RU-WRITE-SPEC rule to obtain $_{\text{RU}}.x \mapsto V' \cup \{1\} * {}_{\text{RU}}.x \mapsto_c \mathsf{Some}\ 1$. Notice that no matter which of the cases is true in the disjunction $V' = \emptyset \lor V' = \{1\}$, we have that $V' \cup \{1\}$ equals $\{1\}$ which is sufficient for restoring the client invariant again. We can hold on to $_{\text{RU}}.x \mapsto_c \mathsf{None}$, but we will no longer need it at this point. The last operation is `loop`, which is a non-terminating operation and which therefore trivially satisfies any specification (as we are using a partial-correctness logic; technically, in Iris this is proved using so-called Löb induction).

Read-transaction As for the write-transaction, we initialize using the RU-INIT-CLIENT-SPEC rule, and thus we obtain ownership of the resource $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$. Also, as for the write-transaction, reasoning about the start operation goes the same way by opening and closing the client invariant leaving us to continue with the resources $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\})) * {}_{\text{RU}}.x \mapsto_c \mathsf{None}$. For the read operation `read` x , we must open the client invariant to gain $_{\text{RU}}.x \mapsto V$ for a fresh set of values V for which $V = \emptyset \lor V = \{1\}$ holds. According to the postcondi-

tion of RU-READ-SPEC, as our local points-to resource points to None, we get the following information about the return value ow of the write operation: $(\exists v, ow = \mathsf{Some}\ v \land v \in V) \lor ow = \mathsf{None}$. Having obtained this information we can close the invariant with $_{\text{RU}}.x \mapsto V$ and proceed to the assertion $\texttt{assert}(v_x = \mathsf{None} \lor v_x = \mathsf{Some}\ 1)$. As v_x is equal to the return value of the write operation ow, we use the information $(\exists v, ow = \mathsf{Some}\ v \land v \in V) \lor ow = \mathsf{None}$ together with our knowledge about the set V, i.e., $V = \emptyset \lor V = \{1\}$, to conclude that the assertion does indeed hold. Having concluded that the assertion holds, we are only left with reasoning about `commit`. In the precondition of RU-COMMIT-SPEC we collect the connection state, $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\}))$, the local points-to resource, $_{\text{RU}}.x \mapsto_c \mathsf{None}$, and the global points-to resource from the invariant for a fresh V', $_{\text{RU}}.x \mapsto V'$. The postcondition gives us back $_{\text{RU}}.x \mapsto V'$ to close the invariant and the ability to start a new transaction in the form of the connection state $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$.

Remarks Notice how the proof of the read uncommitted data example is modular: We use the node-local reasoning of Aneris, together with the client invariant, to reason about each transaction in isolation. The proof is also modular in the sense that it only relies on the specification of read uncommitted, not the implementation thereof. We further remark that it follows formally by the implication proofs in Chapter 2.4 and Chapter 2.6 that the example is also provable using read committed and snapshot isolation.

2.3 Specifying Read Committed

In this section, we present our modular separation logic specification for read committed. We also present the *dirty read* example and the *commit order* example which we have proven using the specification. Contrary to read uncommitted, read committed is of high practical value as it is the default isolation level of leading database systems such as Microsoft SQL, Oracle or PostgreSQL. A reader unfamiliar with transactional guarantees may be surprised by the wide spread of read committed as, albeit being stronger than read uncommitted, read committed does not impose any order on the data being read. Instead, read committed builds upon read uncommitted by adding the requirement that transactions are only allowed to read committed data. Many formalizations specifying read committed, and other isolation levels for that sake, are not concerned with guarantees about aborting transaction. For example, Adya [1999]; Crooks et al. [2017] do not require that uncommitted transactions must read committed data. Allowing aborting transactions not to read committed data would significantly increase the complexity of our separation logic specification, and hence we have made the design choice that aborting transactions should also read committed data. In practice, imposing the requirement that uncommitted transactions may read *any* uncommitted

data is a weak requirement that will not exclude many, if any, implementations from realizing the specification. We remark that our specification still specifies read committed, it is just not the weakest specification there exists (an algorithm producing a subset of the valid executions under an isolation level does naturally still adhere to that isolation level). We remark that an equivalent choice is made in Chang et al. [2023] when specifying serializability.

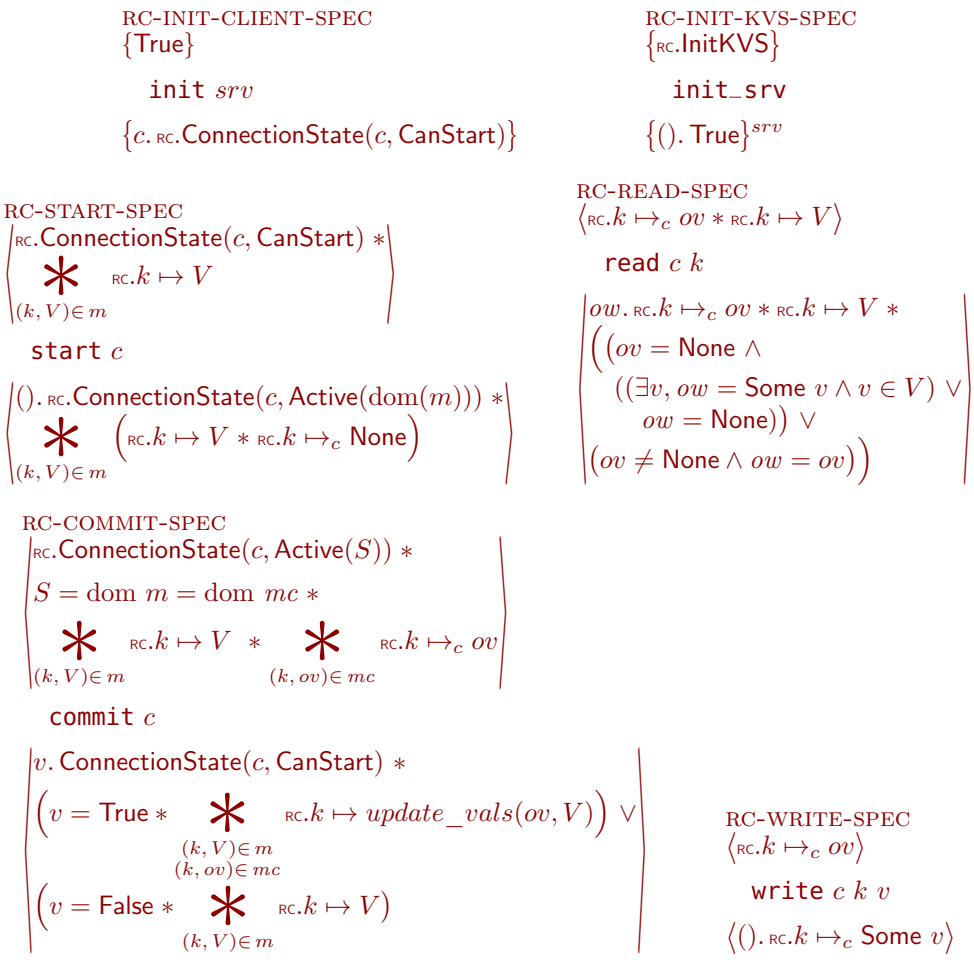


Figure 2.5: Specification for read committed (RCSpecs).

Specification The specification (Figure 2.5) is based upon the same structure as the specification for read committed: local and global points-to resources for sets of values with the only differences occurring in the RC-WRITE-SPEC and the RC-COMMIT-SPEC rules. Hence, we present the read committed specification by highlighting the difference from the read uncommitted specification in the RC-WRITE-SPEC and the RC-COMMIT-SPEC rules.

In comparison with RU-WRITE-SPEC, RC-WRITE-SPEC does not update the global points-to resource, and the update is simply recorded in the local points-to resource. This captures that a concurrently active transaction can not read the change until it is committed. In the RC-COMMIT-SPEC, if the commit is successful, the updates recorded in the local points-to resources are added to the global points-to resources using the *update_vals* function, see (2.4). In case the commit is unsuccessful, the key-value store state, i.e., the global points-to resources, remain unchanged and the changes of the transaction are lost.

$$update\_vals(ov, V) \triangleq \mathsf{match}\ ov\ \mathsf{with}\ \mathsf{Some}\ v => V \cup v \mid \mathsf{None} => V \tag{2.4}$$

As for read uncommitted, observations can be made using Seen resources. In fact, the exclusion property (2.2) and the creation property (2.3) holds for read committed too (all resources should be prefixed with RC. instead of RU.). To conclude the presentation of the read committed specification, we state the complete specification, similar to (2.1), which formally defines the read committed specification (RCSpec), see (2.5).

$$\mathsf{RCSpec} \triangleq \Rrightarrow \mathrm{RC}.\mathsf{InitKVS} * \underset{k \in \mathsf{Keys}}{\ast} \mathrm{RC}.k \mapsto \emptyset * \mathsf{RCSpecs} * \mathrm{RC}.\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} \tag{2.5}$$

Read Committed Example. We have used the read committed specification to formally prove the *dirty read* example Figure 2.6 and the *commit order* example Figure 2.7.

`start`	`start`
`write` x 1	**let** v_x = `read` x **in**
`loop`	`assert`(v_x = None)
	`commit`

$$Inv \triangleq \exists V,\ \mathrm{RC}.x \mapsto V\ * V = \emptyset$$

Figure 2.6: Dirty read example.

The dirty read example builds upon the read uncommitted data example from Figure 2.1, and is based upon the dirty read phenomenon in the literature (Adya [1999]; Berenson et al. [1995] and 1992 ANSI SQL standard), by removing the clause in the conjunction of the assertion that corresponds to the reading transaction seeing the uncommitted data from the writing transaction. This is as expected for read committed, since read committed disallows transactions to read uncommitted data. Recall that a phenomenon is an example of an execution between concurrently executing transactions that should be prohibited; in our example this is captured by the assertion showing that it is impossible.

`start`	`start`	`start`
`write` x 1	`write` y 1	**let** v_a = `read` a **in**
let v_y = `read` y **in**	**let** v_x = `read` x **in**	**let** v_b = `read` b **in**
If (v_y = Some 1) then	If (v_x = Some 1) then	`assert`(!(v_a = Some 1
{`write` a 1 }	{`write` b 1 }	$\wedge$ v_b = Some 1))
`commit`	`commit`	`commit`

Figure 2.7: Commit order example.

The proof of the example is simple and uses the invariant shown in Figure 2.6 which state that no value will be written to the x variable.

The commit order example asserts the absence of a situation in which two transactions both are seeing data committed by the other transaction. This should not be allowed to occur as there must be an order in which transactions commit. The first two transactions in the dirty read example revolve around the same structure: First the value 1 is written into a key that the other transaction reads from and checks whether it was written to using an if-statement. If this was the case, the transactions write to another key (respectively a and b) to signify that they saw the write of the other transaction. The last transaction reads from a and b and asserts that the transactions could not possible both have seen each others writes (the case where both a and b has been written to).

We remark that it follows from the implication proof in Chapter 2.6 that the two examples in this section are also provable for snapshot isolation.

2.4 Read Committed Implies Read Uncommitted

Given our formal separation logic specifications of read uncommitted and read committed, we can now formally prove that read committed implies read uncommitted:

Theorem 2.4.1 (Read committed implies read uncommitted)**.** *The specification for read committed* (2.5) *implies the specification for read uncommitted* (2.1)*:*

$$\mathsf{RCSpec} \rightarrow \mathsf{RUSpec}.$$

Proof structure The proof, consisting of approximately 800 lines of Rocq proof code, centers around closing the gap between read uncommitted and read committed that comes from the fact that in read uncommitted the global state is updated immediately at the time of writing values, whereas for read committed, the global state is not updated until the transaction commits successfully. The proof proceeds naturally by first assuming the read committed specification RCSpec (2.5), which consists of the global invariant $_{\mathsf{RC}}.\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$, fresh global

points-to resources $\ast_{(k,V)\in \mathsf{Keys}}\, {}_{\text{RC}}.k \mapsto \emptyset$, an initialization resource ${}_{\text{RC}}.\mathsf{InitKVS}$ for the key-value store and the read committed per operation specifications RCSpecs (Figure 2.5). To prove RUSpec, we must naturally show the equivalent resources and specifications for read uncommitted (including the exclusion (2.2) and creation (2.3) properties). Now, we need to define all the resources used in RUSpec. In doing this, we will have all the resources found in RCSpec at our disposal. We define the key-value store resource for initialization to be the same as the corresponding read committed resource, but for the global invariant and the global and local points-to resources we use a more involved definition with *resource algebras*. Resource algebras are the building blocks of Iris, see Jung et al. [2016] for details. All we need to know here is that using resource algebras, we can create custom predicates and rules to relate these predicates (usually we will refer to these custom predicates as resources algebras). Having defined all resources, the largest part of the proof revolves around showing the per operation specifications of read uncommitted, i.e., RUSpecs. When proving each specification, we get to use the corresponding specification for read committed from RCSpecs as an assumption. This is helpful, because we first assume the read uncommitted resources in the precondition of the specification and these we have defined in terms of the read committed resources and resource algebras. We thus use the read committed specification for the particular operation to transform the read committed resources, while the resource algebras are transformed by hand, to meet the post condition of the read uncommitted specification.

We remark that while the proof follows a quite natural path, it is non-trivial to come up with a definition of the read uncommitted resources in terms of the read committed resources that makes the proof work. We refer to the Rocq formalization for the details of the whole proof and continue here with an excerpt regarding the write specification and insights about how the proof goes at commit time for the commit specification.

Proof excerpt Our goal is to prove the RU-WRITE-SPEC rule which uses the resources ${}_{\text{RU}}.x \mapsto V$ and ${}_{\text{RU}}.k \mapsto_c ov$. We get to assume the RC-WRITE-SPEC rule and can define the read uncommitted resources, which we will refer to as the *wrapped* resources, using the equivalent resources for read committed, i.e., ${}_{\text{RC}}.x \mapsto V$ and ${}_{\text{RC}}.k \mapsto_c ov$.

To define the wrapped resources, we use a *ghost theory*, which is a set of separation logic predicates defined using resource algebras, together with rules that relate the different predicates. Our ghost theory consists of two predicates, $\mathsf{AuthSet}(V)$ and $\mathsf{FragSet}(V)$, on sets of values V. Intuitively, the *authorative* part $\mathsf{AuthSet}(V)$ can be initialized with the empty set, i.e., $V = \emptyset$, and will at all times hold all the values that have been added since V was the empty set. This is seen in (2.6) where, given the global invariant and $\mathsf{AuthSet}(V)$, one can make an update (signified by $\Rightarrow$) by adding an element to V resulting in

$\mathsf{AuthSet}(V \cup v)$. We remark that unless explicitly stated otherwise, resources from an invariant in Iris can be used to prove a proposition below the update modality as long as the invariant still holds after the update.

$$\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \mathsf{AuthSet}(V) \vdash \Rrightarrow \text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v) \quad (2.6)$$

At the same time, one gains the *fragmental* part $\mathsf{FragSet}(V \cup v)$. As such, $\mathsf{FragSet}(V)$, for any V, represents the values of $\mathsf{AuthSet}(V)$ at some point in the past (note that (2.6) does not update old fragmental parts when new values are added, and values can not be removed). Naturally, this means that at any given point, we can conclude that the set of values in a fragmental resource is a subset of the values in an authoritative resource, see (2.7).

$$\begin{aligned} &\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \mathsf{AuthSet}(V) * \mathsf{FragSet}(V') \\ &\vdash \text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \mathsf{AuthSet}(V) * \mathsf{FragSet}(V') * V' \subseteq V \end{aligned} \quad (2.7)$$

Having defined our ghost theory, we can define the wrapped resources used in the write specification Figure 2.8. Recall, $\text{RU}.k \mapsto_c ov$ and $\text{RC}.k \mapsto_c ov$ are used

$$\begin{aligned} \text{RU}.x \mapsto V &\triangleq \exists V', V' \subseteq V * \text{RC}.x \mapsto V' * \mathsf{AuthSet}(V) \\ \text{RU}.k \mapsto_c ov &\triangleq \exists V, ((\exists v, ov = \mathsf{Some}\ v \wedge v \in V) \vee ov = \mathsf{None}) * \text{RC}.k \mapsto_c ov * \mathsf{FragSet}(V) \end{aligned}$$

Figure 2.8: Wrapped resources.

in the same way in read uncommitted and read committed: ov/ov is the latest value written (if a write has been made) by the current active transaction, and it is updated using the write specification. In contrast, $\text{RU}.x \mapsto V$ and $\text{RC}.x \mapsto V$, holding the global state, are not used in the same way: For read uncommitted the global state is updated immediately at the time of writing the value, and for read committed the global state is not updated until the transaction commits successfully. Therefore, the wrapped resources for read uncommitted in Figure 2.8 are constructed to hide the fact that values are not propagated to the global state until commit time. To see this, observe that inside the definition of $\text{RU}.x \mapsto V$ lies $\text{RC}.x \mapsto V'$ with V' being a subset of V. The set V contains the committed and uncommitted values whereas V' only contains the committed values. $\mathsf{AuthSet}(V)$ is then used to keep track of all the uncommitted values. Further, $\text{RU}.k \mapsto_c ov$ is defined using the equivalent resource $\text{RC}.k \mapsto_c ov$ where it is also stated that if $ov = \mathsf{Some}\ v$ for some value v, then v must be in the set of all values (captured using $\mathsf{FragSet}(V)$ and the requirement $v \in V$ for some V). In the proof of the write specification, which we will sketch next, it is not evident that we the need our ghost theory since we only update it, but we do not use it to draw any conclusions. The necessity of the ghost theory appears at commit time: As V' in $\text{RC}.x \mapsto V'$ is updated with v from $\text{RC}.k \mapsto_c \mathsf{Some}\ v$ at commit time for read committed, and $\text{RU}.x \mapsto V$ remains

the same at commit time for read uncommitted, we use the fact that there exists some V'' with $v \in V''$, for which $\mathsf{FragSet}(V'')$ holds, per the definition of our wrapped resource, which implies v is also in V using rule (2.7) of our ghost theory (the resource $\mathsf{AuthSet}(V)$ comes from inside the definition of $_{\text{RU}}.x \mapsto V$). Thus, at commit time, even though $_{\text{RC}}.x \mapsto V'$ changes inside the wrapped resource $_{\text{RU}}.x \mapsto V$, we can conclude that the added value is already in V, and V can remain unchanged, due to the ghost theory construction, which is needed to prove RU-COMMIT-SPEC. In this sense, the wrapped resource are constructed to hide the fact that values are not propagated until commit time.

The write specification we end up having to prove (when unwrapping the wrapped resources) is shown in Figure 2.9.

$$\left\langle \begin{array}{l}(\exists V, ((\exists v', ov = \mathsf{Some}\ v' \land v' \in V) \lor ov = \mathsf{None}) * {}_{\text{RC}}.k \mapsto_c ov * \mathsf{FragSet}(V)) * \\ (\exists V', V' \subseteq V * {}_{\text{RC}}.x \mapsto V' * \mathsf{AuthSet}(V))\end{array} \right\rangle$$

$$\texttt{write}\ c\ k\ v$$

$$\left\langle \begin{array}{l}(). (\exists V, ((\exists v', \mathsf{Some}\ v = \mathsf{Some}\ v' \land v' \in V) \lor \mathsf{Some}\ v = \mathsf{None}) * {}_{\text{RC}}.k \mapsto_c \mathsf{Some}\ v * \\ \mathsf{FragSet}(V)) * (\exists V', V' \subseteq V \cup v * {}_{\text{RC}}.x \mapsto V' * \mathsf{AuthSet}(V \cup v))\end{array} \right\rangle$$

Figure 2.9: Unwrapped write specification.

In the precondition, when instantiating the existential quantifiers, the relevant resources we get to assume are $_{\text{RC}}.k \mapsto_c ov * {}_{\text{RC}}.x \mapsto V' * \mathsf{AuthSet}(V)$ with $V' \subseteq V$. Using the read committed write specification RC-WRITE-SPEC, we can transform the resource $_{\text{RC}}.k \mapsto_c ov$ into $_{\text{RC}}.k \mapsto_c \mathsf{Some}\ v$. Using (2.6), we can transform $\mathsf{AuthSet}(V)$ into $\mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v)$. Hence, we can now satisfy the postcondition of Figure 2.9, using $_{\text{RC}}.k \mapsto_c \mathsf{Some}\ v * {}_{\text{RC}}.x \mapsto V' * \mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v)$, which concludes the proof sketch of the write specification.

2.5 Specifying Snapshot Isolation

In this section, we first present our modular separation logic specification for snapshot isolation, and then we present three examples based on phenomena from the literature, *write skew*, *read skew* and *non-repeatable read*, in addition to a bank transfer example which we have all proven using the specification. Snapshot isolation, like read committed, is an isolation level with practical value, but it comes with much stronger guarantees about what transactions are allowed to read.

Snapshot isolation, as presented in Berenson et al. [1995], makes use of snapshots. A snapshot is defined as the state of the database at a given time. Snapshot isolation makes a distinction between the start snapshot, on which a transaction reads, and the commit snapshot, on which the transaction attempts

to commit its updates. All commits happen in a globally defined linear order. The start and commit snapshots are taken at the time of the start operation and the commit operation respectively. In this paper, the snapshots will be the most recent snapshots at the time they are a captured. This can be referred to as "strong snapshot isolation" [Daudjee and Salem, 2006], as one can relax the criteria on the start snapshot to be any valid snapshot at the time of the start operation — not necessarily the most recent. [6] Naturally, the start snapshot and the commit snapshot can be different as other concurrent transactions can commit in the time between the snapshots. This leads us to the crucial commit-check of snapshot isolation, which expresses when a transaction can commit:

> *(Snapshot Isolation Commit-check)* A transaction is allowed to commit only if it has no write conflicts with any transaction committed between its own start snapshot and its commit snapshot.

As for the previous isolation levels, our separation logic specification will be based around local and global points-to resources, but this time the global points-to resources do not point to a set of values but *histories*. Histories are lists of values with the last element representing the latest committed value to the key. Histories, as opposed to sets, are needed to express the snapshot isolation commit check: we can use the histories from a start snapshot and a commit snapshot to conclude whether writes have happened between the two snapshots.

The initialization specifications, SI-INIT-CLIENT-SPEC and SI-INIT-KVS-SPEC, are identical to the ones of the previous isolation levels. We now describe the specifications of the start, read, write and commit operations for snapshot isolation which can all be found in Figure 2.10.

Start In snapshot isolation, when starting a transaction, we take a snapshot of the database. It is on the basis of this snapshot that the sequential reasoning inside the transactions happens: If the transaction does not include any writes, then all reads happen from this snapshot. In the precondition of SI-START-SPEC, the client will provide its connection state together with the state of the key-value store in form of global points-to resources, which is typically obtained from an invariant. The start operation then changes the connection state to active such that it holds information about its start snapshot ($\text{si.ConnectionState}(c, \text{Active}(m))$). As the sequential reasoning inside transactions happens based on the start snapshot, all the local points-to resources are created with values equal to the current state of the key-value store.

[6] In Berenson et al. [1995] starting snapshots are allowed to be any previous snapshot and not necessarily the most recent. Specifications for the weaker variant would be much harder to work with, as you would get a disjunction of all possible snapshots to reason about when starting a transaction.

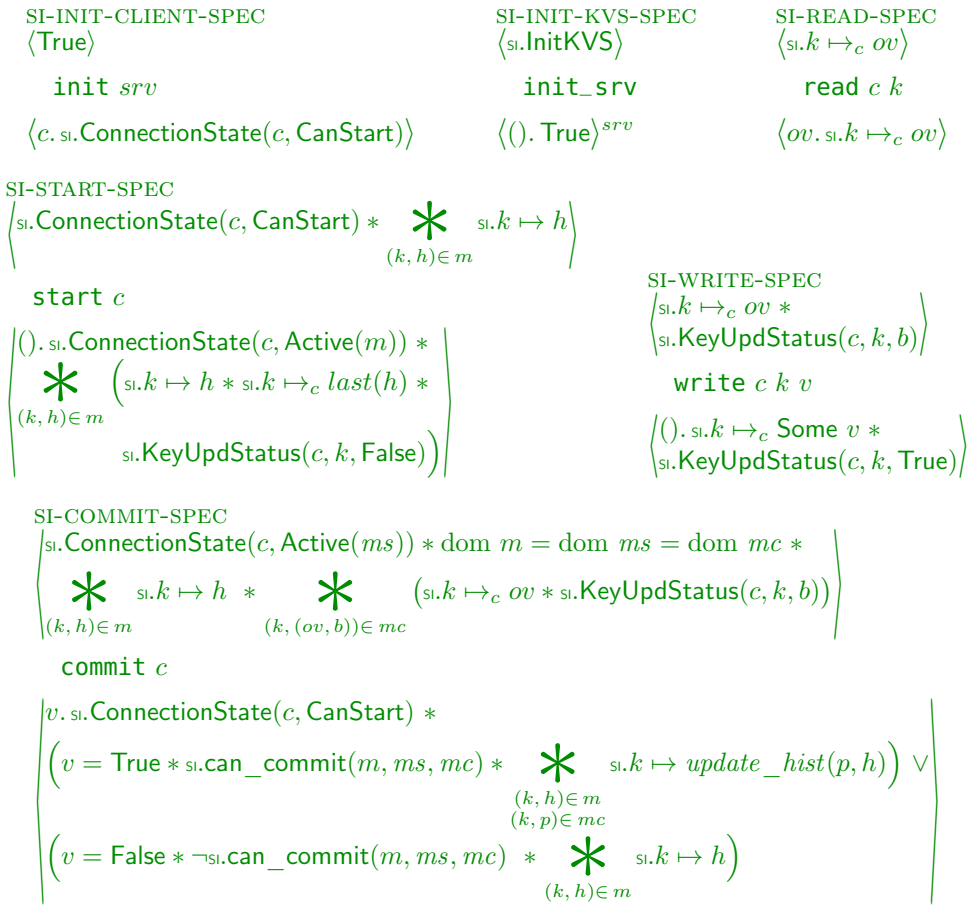


Figure 2.10: Specification for snapshot isolation (SISpecs).

As the start of a transaction does not apply any updates, the global points-to resources are given back without modifications for the client to reestablish its invariant. When the transaction starts modifying the local points-to resources by doing writes, we lose the ability to distinguish the original snapshot from client changes. That is why the SI-START-SPEC rule includes the resource $\text{si}.\mathsf{KeyUpdStatus}(c, k, \mathsf{False})$ in the postcondition. This resource states, for a particular key, whether the client did an update to this key. When talking about the SI-WRITE-SPEC rule next, we will see how the resource is updated and that it is this point that distinguishes our write specification from a store operation for the heap.

Read and Write The read and write operations are not concerned with the global points-to resources, as they are working on a snapshot of the database, initially captured at start time in the local-points to resources. Hence, a local points-to resource $\text{si}.k \mapsto_c vo$ is used in the SI-READ-SPEC rule to obtain a

specification that is similar to the standard specification of a load operation on the heap. Likewise, the SI-WRITE-SPEC rule is manipulating the points-to predicate similarly to a store operation for the heap, except the change is recorded in the $\text{SI.KeyUpdStatus}(c, k, b)$ resource which we need at commit time.

Commit Before delving into the details of the SI-COMMIT-SPEC rule, we remind the reader how transactions are supposed to commit under snapshot isolation. According to the snapshot isolation commit-check, we have to check if there are any write-conflicts between what the transaction wrote and what other concurrent transactions have done in the meantime. Therefore, to use the SI-COMMIT-SPEC rule, one must provide (1) the update status for each key in the starting snapshot, in the form of $\text{SI.KeyUpdStatus}(c, k, b)$, together with the updated value from the local points-to resources (mc holds the information for these resources); (2) the starting snapshot of the transaction, in the form of $\text{SI.ConnectionState}(c, \text{Active}(ms))$, where ms is exactly the snapshot gained from using the SI-START-SPEC rule; (3) the current snapshot of the key-value store, that is, the commit snapshot denoted as m, in the form of global points-to resources for all the keys in the start snapshot. Having provided the necessary resources in the precondition of the SI-COMMIT-SPEC rule, the postcondition expresses that the operation can proceed in one of two ways: Either the transaction unsuccessfully commits and the state of the key-value store remains the same as the current state, or the transaction successfully commits and the state of the key-value store is updated with the updates of the transaction. To express the update to the key-value store we use the function $update\_hist$, which takes as arguments both a pair consisting of value and a boolean, and a history. The history is updated, by appending the value to it, if the update-status is true (the update status being the boolean argument). In either case of the postcondition, evidence is provided as to why the case happened: If the transaction committed successfully, $\text{SI.can\_commit}(m, ms, mc)$ will be true, which expresses that there are no write-conflicts with other transactions. Dually, if the transaction is unsuccessful in committing, $\text{SI.can\_commit}(m, ms, mc)$ is false, entailing at least one write-conflict with another transaction. In Figure 2.11, we display the predicate for which SI.can_commit is a decidable procedure. The predicate captures that if the transaction did an update, ($mc[k] = \textbf{Some}\,(p, \text{True})$), then the start snapshot and the commit snapshot must be equal ($m[k] = ms[k]$).

$$\text{CanCommit}(m, ms, mc) \triangleq \forall k \in \text{Keys},\ \ mc[k] = \textbf{Some}\,(p, \text{True}) \Rightarrow m[k] = ms[k].$$

Figure 2.11: The CanCommit predicate for which the can_commit function is a decidable procedure.

Observations This completes our explanation of our specifications of the snapshot isolation operations. As for read uncommitted and read committed,

clients are able to make observations using the Seen resource. In the exclusion property (2.8) and the creation property (2.9) below, we see how Seen resources work on histories instead of sets of values. Moreover, we see that from the exclusion property, a client is able to conclude that an observed history must be a prefix of the history representing the key-value store state. In Appendix 2.10, we discuss the differences between observations made in read uncommitted, read committed and snapshot isolation.

$$\text{RU.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \text{SI.}\mathsf{Seen}(k, h') * \text{SI.}k \mapsto h \vdash \Rrightarrow \text{SI.}k \mapsto h * h' \leq h \tag{2.8}$$

$$\text{SI.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * \text{SI.}k \mapsto h \vdash \Rrightarrow \text{SI.}k \mapsto h * \text{SI.}\mathsf{Seen}(k, h) \tag{2.9}$$

To conclude the presentation of the snapshot isolation specification, we will a state the complete specification similar to (2.1) and (2.5) that formally defines the snapshot specification (SISpec), see (2.10).

$$\mathsf{SISpec} \triangleq \Rrightarrow \text{SI.}\mathsf{InitKVS} * \mathop{\Large *}_{k \in \mathsf{Keys}} \text{SI.}k \mapsto [\,] * \mathsf{SISpecs} * \text{SI.}\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} \tag{2.10}$$

Snapshot isolation examples. For snapshot isolation, we have proven three examples based on phenomena from the literature (Adya [1999]; Berenson et al. [1995] and 1992 ANSI SQL standard) and a bank transfer example. The examples based on phenomena are the *write skew* example in Figure 2.12, the *read skew* example in Figure 2.13 and the *non-repeatable read* example in Figure 2.14. The formal proofs of these three examples proceed in a manner similar to the proof shown for read uncommitted. For details see the accompanying Rocq formalization. In addition to the examples for snapshot isolation presented in this section, we have verified a number of additional examples, displaying important properties of snapshot isolation, which can be found in Appendix 2.11.

The write skew example asserts that all executions between a transaction that reads from one key and writes to another key and another transaction that does the same, but swaps the keys, will commit. Note that even the execution in which both transactions share the same snapshot, in form of the initial snapshot, succeeds in snapshot isolation whereas it is not allowed under serializability because serializability requires there to be an order amongst transactions. Naturally, proving the assertions in this example relies on the inclusion of the CanCommit predicate in the SI-COMMIT-SPEC rule. The read skew example asserts that a transaction will either see all the writes or no writes from other transactions. Finally, the non-repeatable read example asserts that once a transaction sees a write from another transaction it can not unsee it.

Bank transfer As opposed to the arguably contrived examples based on phenomena from the literature, the bank transfer example in Figure 2.15

`start`	`start`
`read` y	`read` x
`write` x 1	`write` y 1
`assert(commit)`	`assert(commit)`

Figure 2.12: Write skew.

`start`	`start`
`write` x 1	**let** $v_x =$ `read` x **in**
`write` y 1	**let** $v_y =$ `read` y **in**
`assert(commit)`	`assert`$(v_x = v_y)$
	`assert(commit)`

Figure 2.13: Read skew.

`start`	`start`
`write` x 1	**let** $v_1 =$ `read` x **in**
`assert(commit)`	**let** $v_2 =$ `read` x **in**
	`assert`$(v_1 = v_2)$
	`assert(commit)`

Figure 2.14: Non-repeatable read.

represents a common use case. Namely, the use case in which an amount of money is transferred from one account to another account in a banking system, given that there is sufficient funds in the source account. The example in Figure 2.15 contains a single transaction, which we imagine could be part of a larger system, and in comments it is shown how each line of code interacts with the separation logic resources, and, most importantly, how the invariant of the system is updated. The invariant for this example, which could be a part of a larger invariant with multiple accounts, is simply: $\exists h_{src}\, h_{dst}\, v_{src}\, v_{dst},\ \text{si}.src \mapsto h_{src} +\!\!+ [v_{src}] * \text{si}.dst \mapsto h_{dst} +\!\!+ [v_{dst}]$ (it has been boxed and the existential quantification has been omitted in Figure 2.15). It states that there are two accounts, named src and dst, which points to histories with the latest updates being v_{src} and v_{dst} respectively (these are the current amounts in the accounts). Using the invariant and SI-START-SPEC, a snapshot is created in Figure 2.15 for the duration of the transaction in which the amount in the accounts are v_{src} and v_{dst}. The if-statement is used to check whether the balance in the source account is sufficient to withdraw the amount that will be transferred (we

$// \boxed{\text{SI}.src \mapsto h_{src} +\!\!+ [v_{src}] * \text{SI}.dst \mapsto h_{dst} +\!\!+ [v_{dst}]}$
`start`
$// \text{SI}.src \mapsto_c v_{src} * \text{SI}.dst \mapsto_c v_{dst}$
let $bal_{src} =$ `read` src **in**
$// \text{SI}.src \mapsto_c v_{src} * \text{SI}.dst \mapsto_c v_{dst} \wedge bal_{src} = v_{src}$
If $(bal_{src} \geq amount)$ then
{
  $// \text{SI}.src \mapsto_c v_{src} * \text{SI}.dst \mapsto_c v_{dst} \ \wedge$
    $bal_{src} = v_{src} \wedge v_{src} \geq amount$
  `write` src $(bal_{src} - amount)$
  $// \text{SI}.src \mapsto_c v_{src} - amount * \text{SI}.dst \mapsto_c v_{dst} \wedge$
    $bal_{src} = v_{src} \wedge v_{src} \geq amount$
  let $bal_{dst} =$ `read` dst **in**
  $// \text{SI}.src \mapsto_c v_{src} - amount * \text{SI}.dst \mapsto_c v_{dst} \wedge$
    $bal_{src} = v_{src} \wedge v_{src} \geq amount \wedge bal_{dst} = v_{dst}$
  `write` dst $(bal_{dst} + amount)$
  $// \text{SI}.src \mapsto_c v_{src} - amount * \text{SI}.dst \mapsto_c v_{dst} + amount$
    $\wedge \ bal_{src} = v_{src} \wedge v_{src} \geq amount \wedge bal_{dst} = v_{dst}$
}
`commit`
$//$ *If commit = true and* $v_{src} \geq amount$ *:*
$\boxed{\begin{array}{l}\text{SI}.src \mapsto h_{src} +\!\!+ [v_{src}] +\!\!+ [v_{src} - amount] * \\ \text{SI}.dst \mapsto h_{dst} +\!\!+ [v_{dst}] +\!\!+ [v_{dst} + amount]\end{array}}$
Otherwise :
$\boxed{\text{SI}.src \mapsto h'_{src} +\!\!+ [v'_{src}] * \text{SI}.dst \mapsto h'_{dst} +\!\!+ [v'_{dst}]}$

Figure 2.15: Bank transfer example. [7]

want to avoid a negative balance). In case the balance is sufficiently large, the amount is withdrawn from the source account and inserted into the destination account using the SI-WRITE-SPEC rule. Hence, the starting snapshot (the local points-to resources) is changed to a snapshot in which the accounts contain the updated values $v_{src} - amount$ and $v_{dst} + amount$, respectively. At commit time, it is the updates in this new snapshot that we want to push to the key-value store state, represented by the global points-to resources in the invariant. If the commit is successful, we get to do the updates atomically, i.e., write the updates to both accounts. In case the commit is unsuccessful, it is because there is a conflict with another transaction updating the same accounts, cf.

the snapshot isolation commit check captured by the CanCommit predicate in SI-COMMIT-SPEC, resulting in the account balances being described by fresh values.

In Chang et al. [2023], a similar bank transfer example is proven in separation logic using the stronger isolation level serializability. As we are able to prove the bank transfer using snapshot isolation, this example demonstrates why applications do not always have to opt for the strongest isolation levels; there can be a weaker isolation level, which provides sufficient consistency guarantees. Weaker isolation levels always come with the benefit that they enhance the throughput of a system due to the increased concurrency — something which is of importance in a banking system with many customers.

We remark that only the snapshot isolation specification is strong enough to prove the examples in this section; they can not be proven using the read uncommitted or the read committed specifications.

2.6 Snapshot Isolation Implies Read Committed

We can now formally prove that snapshot isolation implies read committed:

Theorem 2.6.1 (Snapshot isolation implies read committed)**.** *The specification for snapshot isolation* (2.10) *implies the specification for read committed* (2.5)*:*

$$\mathsf{SISpec} \to \mathsf{RCSpec}.$$

The theorem is proven using the same structure as described in Chapter 2.4, and the proof amounts to approximately 800 lines of Rocq proof code. We emphasize that while the proof follows the obvious path, coming up with an encoding of the read committed resources using the snapshot isolation resources that makes the proof work is non-trivial. We will refer to the Rocq formalization for the details of the proof.

2.7 Verifying that a Multi-Version Concurrency Control Key-Value Store Implements Snapshot Isolation

In this section we show that our implementation of the original multi-version concurrency control algorithm for snapshot isolation [Berenson et al., 1995], as a single-node key-value store in a distributed system, satisfies the specification for snapshot isolation. The section is divided into three main parts each constituting a subsection:

[7] For the sake of presentation, this example hides the use of optionals cf. the specification.

1. *Implementation*: We give an overview of the original multi-version concurrency control algorithm for snapshot isolation from Berenson et al. [1995] in the context of a distributed system.

2. *Client Proxy Proof*: We outline the client state and how the proof of the client proxy interacts with the server-side using specifications for remote procedure call handlers. We include an excerpt of our ghost theory, which captures relationships among the separation logic predicates we have defined and used.

3. *Server-Side Proof*: We go into more detail with proving remote procedure call handler specifications. We include a description of the server side resources and our model capturing key properties of snapshot isolation.

We remark that we do not cover the full details of the proof, and use simplifications, as the whole proof is too large to be included in the paper. For the full details, see the Rocq formalization. Note further that a formal consequence of Theorems 2.4.1 and 2.6.1 is that the multi-version concurrency control implementation of snapshot isolation also satisfies the specifications for read committed and read uncommitted.

2.7.1 Implementation

The data structure of the key-value store implementation is a map from keys to lists of value-timestamp pairs. The value-timestamp pairs correspond to the values written in the past with associated commit times, with the last value being the most recent. The server state, moreover, consists of an integer reference, for generating new timestamps to transactions, and a lock for guarding concurrent access to the key-value map and the timestamp integer. The server implements three remote procedure call handlers, in the form of start, read and commit, which together with client proxy code makes up the start, read and commit operations that clients can use. The write operation has no remote procedure call as it is handled locally at the client proxy. The client proxy state can be in one of two modes: The state is empty, because no transaction is active, or the state contains the cached writes and the start timestamp of the active transaction. Note that at most one transaction can be active at a time per client proxy. For a client to have multiple active transactions, a client connection must be initialized using SI-INIT-CLIENT-SPEC for each transaction. In fact, the client proxy state we talk about in this section is actually per connection. Upon using the start operation, the client proxy asks the server to provide it with a start timestamp. The server serves this by incrementing its timestamp counter and returning a fresh value. Having started a transaction, the read and write operations can now be invoked. In its implementation of the write operation, all the client proxy does is to record the update in its cache. The read operation is more complicated. First, it is checked whether the key, on

which the read operation is invoked, has an update in the client cache. If there is an update, the update is returned, otherwise, the read handler of the server is asked to provide a value for the key. Together with the key, the read handler is called with the start timestamp of the transaction, as the reading must be done according to the start snapshot. The start snapshot is represented by the start timestamp. The server uses the key and starting timestamp to retrieve the correct value by going back into the list of value-timestamp pairs. The value with the largest timestamp strictly less than the starting timestamp is returned. Upon commit time, when the commit operation is invoked on the client proxy, the client proxy invokes the commit handler on the server by providing the start timestamp and all the cached writes of the transaction. The commit handler checks that for all the keys in the cache, no updates have been made after the starting timestamp (this is the snapshot isolation commit check, cf. Figure 2.11). If the check goes well, all the cached updates are appended to the lists of value-timestamp pairs. The timestamp, in a newly added pair, will be a fresh timestamp from the server state higher than all previous timestamps.

In the following two sections we will provide more detailed explanations of the implementation when needed to describe proof details.

2.7.2 Client Proxy Proof

The client proxy proof serves as a middle layer between the top level specifications of Figure 2.10 and the server side and makes use of remote procedure call handler specifications. When a client invokes the start, read, write or commit operation, it is the client proxy that gets called. Therefore, it is the client proxy who receives the resources in the precondition of the specifications, and it is also the client proxy who is ultimately responsible for returning the resources in the postcondition.

Client-side resources Upon gaining the resources in the precondition of the specifications in Figure 2.10, the client proxy takes a lock guarding its state and resources. The lock is used to prevent undefined behavior if the client proxy is used concurrently by multiple threads of the client. The lock is implemented as a spin lock, similarly to what is presented in Birkedal and Bizjak [2017, Section 8.6]. The essence of the specifications for the acquire and release operations of the lock is that one gets to have ownership of some resources protected by the lock. Acquire takes ownership of the resources, and release gives back the ownership to the lock for others to take. The structure of the resources protected by the client proxy lock is displayed (2.11) ("..." is notation for resources which we have omitted for presentation purposes). The resources consist of a pointer ℓ, which points to the client state. The client state is either active, pointing to a starting snapshot t and a client cache *cache*, or the state is in the inactive mode and points to nothing. It is by using the client proxy resources in the lock, together with the resources from the preconditions of the

top level specifications that the client proxy gathers the necessary resources for communicating with the server.

$$Lock_{client}(\ell) \triangleq \exists\ state.\ \ell \hookrightarrow state * \big((state = \textbf{Some}\ \ (t, cache) * \dots) \vee (state = \textbf{None} * \dots)\big) \tag{2.11}$$

Network communication The network model of Aneris is an unreliable network based on UDP entailing that messages can be dropped, duplicated and reordered. We have abstracted away from dealing with the low-level details of an unreliable network by utilizing the remote procedure call library of Gondelman et al. [2023], which implements reliable communication channels on top of the unreliable network. Hence we can think of the client-server communication as the client invoking handlers provided by the server. The interaction between the client and the server in the proof will therefore be happening through the specification of the handlers. Specifications corresponding to SI-INIT-CLIENT-SPEC and SI-INIT-KVS-SPEC come directly, for free, from the reliable communication library, and the write operation does not communicate with the server. Therefore, we have provided handlers for the start, read and commit operations. In Figure 2.16, we present the specification for the read handler. The read handler is the simplest of the three handlers, as the start and commit handler has preconditions consisting of multiple non-trivial view shifts (formally $P \twoheadrightarrow \Rightarrow Q$ is view shift from P to Q). View shifts in the precondition of handlers forces the client proxy to prove that given resources P, the server can update P to the resources Q. To explain the read handler specification, we first introduce our ghost theory.

HT-READ-HANDLER
$\{\mathsf{TimeFrag}(t) * \mathsf{SnapFrag}(t, M_t) * M_t[k] = \textbf{Some}\ \ H\}$
$read\_handler\ k\ t$
$\{ov.\ (ov = \textbf{None} * H = [\,]) \vee (\exists\ v, t'.\ ov = \textbf{Some}\ \ v * last(H) = (v, t'))\}^{srv}$

Figure 2.16: Read handler specification.

Ghost theory Similar to the proof sketch in Chapter 2.4, we will be working with a ghost theory of separation logic predicates together with rules that relate the different predicates. Our ghost theory consists, among others, of the following resources: $\mathsf{TimeGlo}(t)$, $\mathsf{TimeLoc}(t)$, and $\mathsf{TimeFrag}(t)$, all parameterized by a timestamp t; $\mathsf{SnapFrag}(t, M_t)$ and $\mathsf{SnapAuth}(S)$ parameterized by a timestamp t and a map from keys to histories M_t and a map S from timestamps to maps from keys to histories; $\mathsf{MemGlo}(M)$ and $\mathsf{MemLoc}(M)$, parameterized by a map M from keys to histories; and $k \mapsto H$ parameterized by a key k and a history H. Note that in this section histories are different from the terminology used in Chapter 2.5. In this section, histories are lists of value-timestamp pairs

as opposed to lists of values (in the client-facing specs in Chapter 2.5 there is no need to use timestamps). The resource $k \mapsto H$ is also different from the key-value store points-to resources in Chapter 2.5, but it is ultimately $k \mapsto H$ that makes up the key-value store points-to resources we have seen in Chapter 2.5. Therefore, in this section we will think of $k \mapsto H$ as the key-value store points-to resources used in the SI-START-SPEC and SI-COMMIT-SPEC rules. An excerpt of the laws of our ghost theory are shown in Figure 2.17. Going forward, we will go into details with our ghost theory laws as needed for explaining proof details. Let us now return our attention back to the read handler.

$$\mathsf{TimeGlo}(t) * \mathsf{TimeLoc}(t) * t < t' \vdash \mathsf{TimeGlo}(t') * \mathsf{TimeLoc}(t') * \mathsf{TimeFrag}(t') \quad \text{(Law 1)}$$

$$\mathsf{TimeGlo}(t) * \mathsf{TimeLoc}(t') \vdash \mathsf{TimeGlo}(t) * \mathsf{TimeLoc}(t') * t = t' \quad \text{(Law 2)}$$

$$\mathsf{MemGlo}(M) * \mathsf{MemLoc}(M') \vdash \mathsf{MemGlo}(M) * \mathsf{MemLoc}(M') * M = M' \quad \text{(Law 3)}$$

$$k \mapsto H' * \mathsf{MemGlo}(M) * \mathsf{MemLoc}(M) \quad \text{(Law 4)}$$
$$\vdash k \mapsto H * \mathsf{MemGlo}(M[k \mapsto H]) * \mathsf{MemLoc}(M[k \mapsto H])$$

$$\mathsf{SnapFrag}(t, M_t) * \mathsf{SnapAuth}(S) \vdash \mathsf{SnapAuth}(S) * S[t] = M_t \quad \text{(Law 5)}$$

$$\mathsf{SnapAuth}(S) * t \notin \mathrm{dom}(S) \vdash \mathsf{SnapAuth}(S[t \mapsto M]) * \mathsf{SnapFrag}(t, M) \quad \text{(Law 6)}$$

Figure 2.17: Excerpt of ghost theory laws.

The precondition includes some client state: The transaction has a starting timestamp t with associated starting snapshot M_t. Furthermore, the client knows that the key k has the history H in this snapshot. The postcondition expresses that the read handler must return a value in accordance with the history H.

Next, we turn our attention to the server side to, among other things, see how the read-handler specification is proven.

2.7.3 Server-Side Proof

The server-side proof is exactly the proof that the three remote procedure call handlers (start, read and commit) implement their specifications. As soon as one of the handlers gets invoked, they will have to acquire the server lock before proceeding.

Server-side resources The server lock resources are displayed (2.12). Similarly to the client lock, the server lock has memory pointers to local state: ℓ_{kvs} points the key-value store itself (kvs maps keys to lists of value-timestamp pairs) while ℓ_t points to the integer/timestamp variable used to generate start and commit timestamps. Furthermore, the lock resources includes the timestamp resource $\mathsf{TimeLoc}(t)$ and the memory resource $\mathsf{MemLoc}(M)$. Here $\mathsf{isMap}(kvs, M)$ asserts that the ghost map M, defined in the logic, corresponds to kvs, which represent the key-value store in the program. Lastly, we assert that our "model"

holds, i.e., $\mathsf{Model}(M, S, t)$ for the snapshots S (S is a map from timestamps to maps from keys to histories).

$$Lock_{server}(\ell_{kvs}, \ell_t) \triangleq \exists\ kvs, t, m, M, S. \ell_{kvs} \hookrightarrow kvs * \ell_t \hookrightarrow t * \mathsf{TimeLoc}(t) * \mathsf{MemLoc}(M) * \mathsf{isMap}(kvs, M) * \mathsf{Model}(M, S, t) \quad (2.12)$$

Mathematical model As part of our verification effort, we have defined a mathematical model to reflect key properties of snapshot isolation in the state of the system. The state of the system is here captured by the logical representation of the key-value store M, all the snapshots captured in S and the latest timestamp t. The model does not own any separation logic resources, it consists purely of logical facts (i.e., the $\mathsf{Model}(M, S, t)$ predicate is a persistent predicate). An important property of our model is that it defines a notion of a *cut* for the state, see the definition (2.13).

$$Cut(t, H_k^t, H_k) \triangleq \exists H_k',\ H_k = H_k^t +\!\!+ H_k' \wedge \big(\forall\,(v, t') \in H_k^t,\ t' < t\big)\ \wedge\ (\forall\,(v, t') \in H_k',\ t < t') \quad (2.13)$$

A cut captures that a history H_k can be split in two parts according to a timestamp t, such that all the values in the first part (H_k^t) have timestamps strictly less than t, and all the values in the latter part (H_k') have timestamps strictly greater than t. For our model, it holds that if the timestamp t is associated with a snapshot of the key-value store M_t, and H_k^t is the history in the snapshot for the key k, while H_k is the current history for the key, then t will cut H_k and the first part of the cut be H_k^t. We remark that only start timestamps have cuts, as all the timestamps in histories are commit timestamps because values are written at commit time (otherwise the strict inequalities in a cut would not make sense). Likewise, we will only associate starting timestamps with a snapshot of the key-value store in our proof development (the domain of S only has starting timestamps).

$$\mathsf{Model}(M, S, t) \vdash \big(S[t'] = M_t \wedge M_t[k] = H_k^t \wedge M[k] = H_k\big) \Rightarrow Cut(t', H_k^t, H_k) \quad (2.14)$$

This cut property of the model is formally defined (2.14), and it is in fact this property the proof of the read handler relies on.

Read handler The code for the read handler in Figure 2.18 goes through the list of value-timestamp pairs, starting with the most recent for the specified key k, until it finds a value that has a timestamp strictly less than the argument timestamp t. In accordance with the read handler specification in Figure 2.16, we must show that this value is indeed the last value of H if H is non-empty. Specifically, at this point in the proof we know that there exists some H_1 and H_2 s.t. H_k (the current history of key k) is composed by H_1 and H_2 in the following way: $H_k = H_1 +\!\!+ H_2$. Furthermore, we can conclude that $\forall\ v, t'.\ (v, t') \in H_1 \Rightarrow\ t' < t$ and $\forall\ v, t'.\ (v, t') \in H_2 \Rightarrow\ t < t'$. Notice how

```
let start_handler time =
  let timeNext = !time + 1 in
  time := timeNext;
  timeNext

let rec get_last opList sTime =
  match opList with
  | None → None
  | Some list →
    let ((v, time), tail) = list in
    if time = sTime then assert false
    else if time < sTime then Some v
    else get_last tail

let kvs_get_last timeKeyPair kvs =
  let (k, sTime) = timeKeyPair in
  get_last (kvs_get k kvs) sTime

let read_handler kvs timeKeyPair =
  kvs_get_last timeKeyPair !kvs
```

```
let commit_handler kvs cacheData time =
  let cTime = !time + 1 in
  let (sTime, cache) = cacheData in
  if map_is_empty cache then true
  else
    let b = map_forall
        (fun k _ →
          let opList = (map_lookup k !kvs) in
          let list =
            if opList = None then list_nil
            else unSOME opList
          in
          check_key k sTime list) cache
    in
    if b then
      time := cTime;
      kvs := update_kvs !kvs cache cTime;
      true
    else false
```

Figure 2.18: Excerpt of server-side handling code.

this entails that t cuts H_k with regard to H_1, that is, $Cut(t, H_1, H_k)$ holds. As it is the value of H_1 which will be returned, the proof of the read handler specification boils down to showing that $H = H_1$. To reach our goal, we must use a resource from the global invariant, shown in (2.15).

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} \triangleq \exists t_G, M_G, S_G.\mathsf{TimeGlo}(t_G) * \mathsf{MemGlo}(M_G) * \mathsf{SnapAuth}(S_G) * \mathsf{Model}(M_G, S_G, t_G) \quad (2.15)$$

We have defined the global invariant to contain the resources $\mathsf{TimeGlo}(t_G)$, $\mathsf{MemGlo}(M_G)$, $\mathsf{SnapAuth}(S_G)$ and the logical facts of the model $\mathsf{Model}(M_G, S_G, t_G)$. Having $\mathsf{SnapFrag}(t, M_t)$ in the read handler precondition, we can use $\mathsf{SnapAuth}(S_G)$ from the global invariant together with Law 5 of our ghost theory (Figure 2.17) to conclude $S_G[t] = M_t$. As we know $M[k] = H_k$ (H_k is the current history of k), we can use $\mathsf{MemGlo}(M_G)$ from the global invariant and $\mathsf{MemLoc}(M)$ from the server lock together with Law 3 of our ghost theory to obtain the equality $M_G[k] = H_k$. This gives us the logical facts needed to conclude that t cuts H_k with regard to H $\big(Cut(t, H, H_k)\big)$ using the cut property of the model (2.14). Thus, we have established two cuts $Cut(t, H_1, H_k)$ and $Cut(t, H, H_k)$. But, if a timestamp cuts the same history with regard to two different histories, then the two histories must be equal, giving us $H = H_1$, as needed.

Start handler The job of the start handler in Figure 2.18 is to hand out a fresh timestamp for starting transactions. This is done using the timestamp reference of the server state, which is safe as the server lock is held upon using the start handler. Using $\mathsf{TimeGlo}(t_G)$ from the global invariant and $\mathsf{TimeLoc}(t)$ from the server lock, together with Laws 1 and 2 of our ghost theory, the timestamp is updated to match the server state. Moreover, $\mathsf{SnapAuth}(S_G)$

in the global invariant is updated using Law 6 of the ghost theory, with $\mathsf{SnapFrag}(t, S_G)$ being given to the client proxy, such that the newly generated start timestamp points to the correct snapshot and $\mathsf{SnapFrag}(t, S_G)$ can be used by the client with HT-READ-HANDLER later. To satisfy the model constraints in the global invariant and the server lock upon returning, we must update our model. Figure 2.19 defines two valid updates to the model, in this case we will be using property 1.

$$\mathsf{Model}(M, S, t) * t < t' \vdash \mathsf{Model}(M, S[t' \mapsto M], t') \qquad \text{(Property 1)}$$
$$\mathsf{Model}(M, S, t) * t < t' \vdash \mathsf{Model}(\mathit{update\_kvs}(M, C, t'), S, t') \qquad \text{(Property 2)}$$

Figure 2.19: Model update properties.

Commit handler The code of the commit handler Figure 2.18 essentially consists of two parts. First, it is checked whether the key-value store can be updated in a sound way according to the snapshot isolation commit-check. The call to *check_key* is the snapshot isolation commit-check for a single key. The *check_key* function takes the starting timestamp of the transaction as an argument and tests whether there is a more recent write from another transaction in the history of the argument key, after the start timestamp. It is this part of the code which generates the $\mathsf{can\_commit}$ status in the SI-COMMIT-SPEC rule. Given that the commit-check went well, the second part of the code is responsible for updating the key-value store with the writes of the transaction. Now, the server state must be updated in a way that is sound with regard to our model. Luckily, model-update property 2 in Figure 2.19 imitates the changes of the *update_kvs* operation used to update the key-value store. When the *update_kvs* operation updates kvs using the memory pointer in the server lock (2.12), the resource $\mathsf{MemLoc}(M)$ together with $\mathsf{MemGlo}(M_G)$ from the global invariant and the key-value store points-to resources from the precondition of the SI-COMMIT-SPEC rule, can be updated to match the updated key-value store. This is done using Laws 3 and 4 of our ghost theory. Moreover, the timestamp resources in the global invariant and the server lock are updated using the same approach as in the start handler. As all resources are updated in a way that satisfies our model, the global invariant can be closed, and the server lock is re-established and can be released upon returning.

This completes our overview of the implementation and its proof; for more details, see the accompanying Rocq formalization.

2.8 Related Work

Transactions in separation logic While this paper presents the first separation logic specifications for weak isolation levels, there has been earlier

work on separation logic specification and verification of transactions achieving strict serializability [Papadimitriou, 1979]. Strict serializability is serializability (transactions appear to execute one at a time) but with the added requirement that the ordering of committed transactions respect real time order. GoTxn [Chajed et al., 2022] is a verified transaction library, which uses two-phase commit locking for concurrency control and it is verified in the Perennial separation logic [Chajed et al., 2019] (which, like Aneris, is also defined on top of Iris). GoTxn stores data on disk in a non-distributed setting and is implemented on top of GoJournal [Chajed et al., 2021] which provides crash safety. Likewise, the vMVCC [Chang et al., 2023] transaction library uses multi-version concurrency control to implement a more advanced form of concurrency control than GoTxn's approach. Chang et al. [2023] implement an in-memory key-value store in a non-distributed setting, and specify and verify it in Rocq using the Perennial separation logic, although Chang et al. [2023] do not use Perennial's support for reasoning about durable storage and crash safety.

Naturally, specifications for (strict) serializability and weak isolation levels are vastly different. To compare with the strongest isolation level we have specified, i.e., snapshot isolation, we note that a specification for snapshot isolation must view the state of the database twice, once to see the starting snapshot on which reads are made, and once to see the commit snapshot on which updates are committed (technically, this is reflected in our SI-START-SPEC and SI-COMMIT-SPEC, where both of the preconditions include global points-to resources). In contrast, a specification for serializability must only view the state of the database once, as a transaction reads and performs updates on the same snapshot (informally, serializability can be viewed as a special case of snapshot isolation where the start snapshot and the commit snapshot are the same). This difference becomes even more apparent if one tries to specify a higher-order `run` method, which wraps start and commit operations around a body of transaction code. Indeed, the specification for `run` based on serializability in Chang et al. [2023] is more concise than the run specification for snapshot isolation we have proven in Appendix 2.10, due to the relative simplicity of serializability (In Appendix 2.10 we also include a discussion of why a run specification for read committed and read uncommitted would be impractical.) Contrary to what might seem intuitive, and what one can see in more theoretically inclined database papers, serializability does not always imply snapshot isolation. It depends on which notion of implication is being used. As we explain in Appendix 2.12, the implication between the separation logic specification of serializability and snapshot isolation is not provable.

In contrast to our specification of snapshot isolation, neither vMVCC nor GoTxn provides information about why a commit has not succeeded, i.e., a resource in the style of the one expressed using the CanCommit predicate in the postcondition of SI-COMMIT-SPEC. This means that assertions about commit events can not be proven using the vMVCC or GoTxn specification.

Closing the gap between high level specifications and implementations. Other work has also aimed at closing the gap between the seemingly high-level properties of isolation levels and specifications for runnable implementations. Cerone et al. [2015b] gives operational specifications, closer to specifications of actual implementations, for a number of isolation levels, and prove that these are equivalent to specifications in the style of weak memory models. Crooks et al. [2017] has developed a state-based model for reasoning about transactional consistency which has been embedded in TLA$^+$[Soethout et al., 2020]. TLA$^+$[Lamport, 1992] is a specification language and model checker. A version of its specification language, PlusCal, resembles mainstream programming languages but still has to undergo manual translation to executable code. Raad et al. [2019] shows how a pseudo-code implementation of a local transactional library can implement an axiomatized model of snapshot isolation equivalent to the model of Cerone and Gotsman [2018]. The approach is not based on a program logic or mechanized, and can not reason about client programs. Lesani et al. [2022a] lifts linearizable data structures to serializable transactional memory objects using interaction trees as program representations in a mechanized Rocq framework. There is also a line of work on checking database implementations against transaction workloads [Biswas et al., 2021; Bouajjani et al., 2023; Liu et al., 2024; Zhang et al., 2023]. While checkers have shown to be effective at discovering bugs, they can not show the absence of bugs.

Verification of distributed systems using separation logic. Mechanized verification of distributed systems using higher-order separation logics have been carried out a number of times with other foci than transactions. For example, Aneris has been used to verify distributed algorithms with high-availability and lower consistency guarantees, such as conflict-free replicated data types, [Nieto et al., 2022, 2023] and a causally-consistent replicated key-value store [Gondelman et al., 2021]. Grove [Sharma et al., 2023] is another higher-order logic for distributed systems based on Perennial with support for reasoning about durable state and crashes in which a replicated key-value store has been verified.

Weak memory The area of weak memory is another area in which separation logic has been successfully used to formalize relaxed guarantees, see [Kaiser et al., 2017; Turon et al., 2014; Vafeiadis and Narayan, 2013] with Kaiser et al. [2017] being an Iris based logic. While the guarantees of weak isolation for database transactions do not have a direct relationship with the guarantees in weak memory, both areas exhibit very relaxed behavior that makes them difficult to reason about. In the weak memory world, one starts with an operational semantics for the memory, whereas in the work of this paper we

show how an implementation of a weak isolation level, in the form of a database, can satisfy our specification.

2.9 Conclusion and Future Work

In this paper, we have established that separation logic can be used to give modular specifications of and reason about consistency guarantees for transactions. As this work has laid the theoretical foundation for reasoning about transactions in separation logic, there are two obvious paths for extending this work.

First, this work can be seen as a stepping stone to relating specifications for executable code, i.e., the separation logic specifications presented here, to well-established database theory models for reasoning about transactional consistency, i.e., models such as the ones mentioned in Chapter 2.1, which define transactional consistency using, for instance, high level operational semantics or dependency graphs. We remark that the per-operation separation logic specifications presented in this paper already do make a connection to existing database theory, since we have shown how they can be used to exclude the phenomena used to define isolation levels in the literature [Adya, 1999; Berenson et al., 1995] and 1992 ANSI SQL standard, which for many years was, and by the SQL Standard authors still is, the de-facto standard. Making a stronger connection to one of the mentioned models, e.g., by proving formally that our separation logic specifications relate precisely to dependency graphs, is highly non-trivial. To create such a formal connection, it is necessary to use an extra layer of logical machinery, not part of the standard instantiation of Iris, such as Trillium [Timany et al., 2024a], for establishing refinements of state transition systems, or free theorems [Birkedal et al., 2021], for stating invariants on traces created by emitting tags in "ghost-code".

Second, it would be interesting to see the theory applied to various transaction systems. While the database we have implemented is indeed the first executable database that verifiably implements a weak isolation level, and uses the multi-version concurrency control algorithm of Berenson et al. [1995], modern databases comes with many features; e.g., distributed transactions [Corbett et al., 2013; Garcia-Molina and Salem, 1987; Lampson and Sturgis, 1981], where data is scattered across a number of different databases. Further, it would be interesting to see the developed theory applied to databases that are resilient to crash failures. This would necessitate using a distributed program logic with support for writing to durable storage and reasoning about crashes, such as Grove [Sharma et al., 2023].

2.10 Appendix: Utility Code

In addition to the operations of the key-value store API, we have implemented utility operations `wait` and `run`. The `wait` and `run` operations are implemented on top of the key-value store operations and proven correct with regard to their specifications (the specifications are presented in this section) using the specifications from Figure 2.3 Figure 2.5 and Figure 2.10. The `run` is similar to the one presented for serializability in Chang et al. [2023], but has a vastly different specification as discussed in Chapter 2.9. As we will see in the following, while the utility operations provide new functionality, they are good at showcasing the differences between the isolation levels.

```
let wait cst cond k =
  let rec aux () =
    start cst ;
    match read cst k with
    | None →
      commit cst; aux ()
    | Some v →
      if cond v
      then (commit cst)
      else (commit cst; aux ())
  in aux ()
```

```
let weak_wait cst cond k =
  start cst;
  let rec aux () =
    match read cst k with
    | None → aux ()
    | Some v →
      if cond v
      then (commitU cst)
      else (aux ())
  in aux ()
```

```
let run cst handler =
  start cst;
  handler cst;
  commit cst
```

Figure 2.20: Utility code.

Wait operation The `wait` operation comes in two versions: a *weak* version, for read uncommitted and read committed, and a non-weak version for snapshot isolation. The operation works much like a memory fence: It takes as arguments a connection, key and condition. When the operation returns, the condition holds for the value pointed to by the key. The `wait` operation can be used to synchronize transactions across clients to impose a particular order. The `wait` and weak-`wait` are implemented as recursive functions, see Figure 2.20. Starting with the non-weak `wait` operation: Upon every call, a new transaction is started to obtain the current snapshot of the key-value store. The snapshot is then checked for the condition by reading from the specified key and seeing whether the condition holds for the value pointed to be the key. If the condition is true, the operation commits the transaction and terminates, otherwise the transaction is committed, and a recursive call is made to check the condition on a more recent snapshot. Notice how every call starts and commits a transaction. Had the `wait` operation only started one transaction and done the recursive calls inside this transaction, it would never get a new start snapshot and hence not observe changes done by other transactions, including changes that could make the condition true. Now, this part is what distinguishes the non-weak `wait` from the weak `wait`: The weak `wait` operation does not start a new transaction for every recursive call to check the state of the key. This is not needed for read uncommitted and read committed as transactions can see updates made by other transactions during their own execution. Do note that

the key-value store does not *have* to show updates to transactions that are already execution, e.g., the key-value store we have implemented in Chapter 2.7 have realized the specifications for read uncommitted and read committed through the implications proofs without propagating new values to active transactions, but if you do not, then you might as well implement a stronger isolation level. The specification for the non-weak `wait` operation is shown in

WAIT-SPEC

$$\left\{\begin{array}{l}\text{SI.ConnectionState}(c, \mathsf{CanStart}) * \\ \forall v'.\{\mathsf{True}\}\ \mathsf{cond}\ v'\ \{b.\, b \Rightarrow v = v'\}\ * \\ \Box\ {}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\exists h,\ \text{SI.}k \mapsto h * \triangleright \left(\text{SI.}k \mapsto h\ {}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}\right)\right)\end{array}\right\}$$

`wait` c k cond

$$\{().\ \text{SI.ConnectionState}(c, \mathsf{CanStart}) * \text{SI.Seen}(k, h +\!\!+ [v])\}$$

WEAK-WAIT-SPEC

$$\left\{\begin{array}{l}\text{RC.ConnectionState}(c, \mathsf{CanStart}) * \\ \forall v'.\{\mathsf{True}\}\ \mathsf{cond}\ v'\ \{b.\, b \Rightarrow v = v'\}\ * \\ \Box\ {}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\exists V,\ \text{RC.}k \mapsto V * \triangleright \left(\text{RC.}k \mapsto V\ {}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}\right)\right)\end{array}\right\}$$

`wait` c k cond

$$\{().\ \text{RC.ConnectionState}(c, \mathsf{CanStart}) * \text{RC.Seen}(k, V \cup v)\}$$

Figure 2.21: Wait specification and weak wait specification.

Figure 2.21. It has three resources in its precondition. First, one must provide the ability to start a transaction in the form of $\mathsf{ConnectionState}(c, \mathsf{CanStart})$. Second, the user must show that the provided condition is a test for equality for a specific value v. Third, the user must provide the ability to look at the state of the key-value store all the possibly many times the `wait` operation will be opening and closing transactions to check for the condition. This is done in the form of an update modality and a view shift ($P \Rrightarrow Q$ is a view shift and is notation for $P \mathrel{-\!\!*} \Rrightarrow Q$) [Jung et al., 2015] under the persistence modality (going forward we will refer to both $\Rrightarrow$ and $\Rrightarrow$ as view shifts). We recap that the update modality ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ states that given that the invariants in $\mathcal{E}_1$ are active ($\mathcal{E}_1$ is a set of invariant names commonly referred to as a mask, we omitted invariant names, used as a mean to identify each invariant, in the main body of the paper), updates can be made to satisfy P and establish the invariants in $\mathcal{E}_2$. If the two masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are equal, we can choose to write the update modality as $\Rrightarrow_{\mathcal{E}}$ or simply ignore the mask $\mathcal{E}$. The persistence modality ensures that the view shift can be reused all the possibly many times the `wait` operation will be opening and closing transactions. Specifically, the view shifts allow us, possibly by opening an invariant, to get the key-value store points-to resource for the argument key and later close any invariant using the same key-value store points-to resource. Given the mentioned preconditions, if the `wait` operation terminates the user obtains evidence of the observation in

the form of $_{\text{SI}}.\mathsf{Seen}(k, h +\!\!+ [v])$. One can then use the snapshot isolation version of exclusion property (2.2) (see Chapter 2.5) to exclude future key-value store states where the value v has not been written to the key k yet. We remark that in our Rocq development, a more general specification that does not restrict the condition to be an equality is provided. The specification of the weak `wait` operation is much similar to its stronger counterpart. Figure 2.21 displays the weak `wait` specification for read committed, we remark that read uncommitted has an equivalent specification (all resources should be prefixed with $_{\text{RU}}.$ instead of $_{\text{RC}}.$). The difference between the weak `wait` operation and the non-weak operation, on the specification level, lies in the kind of observations one gains when the operation returns. The read committed (and read uncommitted observations) $_{\text{RC}}.\mathsf{Seen}(k, V \cup v)$ are on sets which imposes no order compared to the histories of snapshot isolation. This only lets the user exclude states based on whether a given write have happened or not but not based on the order it happened.

Using the non-weak `wait` specification for snapshot isolation, we have proven two examples for concurrently executing transactions that make assertions based on the observations gained from using `wait`. The first example, *Capturing causality* in Figure 2.22, uses wait to capture a causality relation between transactions. The second example, *Sequential writes commit* in Figure 2.23, asserts that committing will be successful as, even though the two transactions write to the same key, there can not be a write-conflict because the second transaction is waiting for the first transaction to finish.

`start`	`wait` x 1	`wait` y 1
`write` x 1	`start`	`start`
`commit`	`write` y 1	**let** $v_x =$ `read` x **in**
	`commit`	`assert`$(v_x = 1)$
		`commit`

$$Inv \triangleq \exists h_x\ h_y,\ _{\text{SI}}.x \mapsto h_x * {}_{\text{SI}}.y \mapsto h_y\ * \big((h_x = [\,] * h_y = [\,]) \vee\ (last(h_x) = \mathsf{Some}\ 1)\big)$$

Figure 2.22: Capturing causality.

`start`	`wait` x 1
`write` x 1	`start`
`assert(commit)`	`write` x 2
	`assert(commit)`

$$Inv \triangleq \exists h,\ _{\text{SI}}.x \mapsto h\ * ((h = [\,]) \vee\ (h = [1] * \mathsf{Tok}_1) \vee\ (h = [1;2] * \mathsf{Tok}_1 * \mathsf{Tok}_2))$$

Figure 2.23: Sequential writes commit.

Run operation The `run` operation is a higher-order function, which wraps start and commit operations around a body of transaction code. Its specification, shown in Figure 2.24, naturally follows from the specifications in Figure 2.10: To be able to start a transaction, the RUN-SPEC rule must have $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ and a snapshot of the key-value store. The view

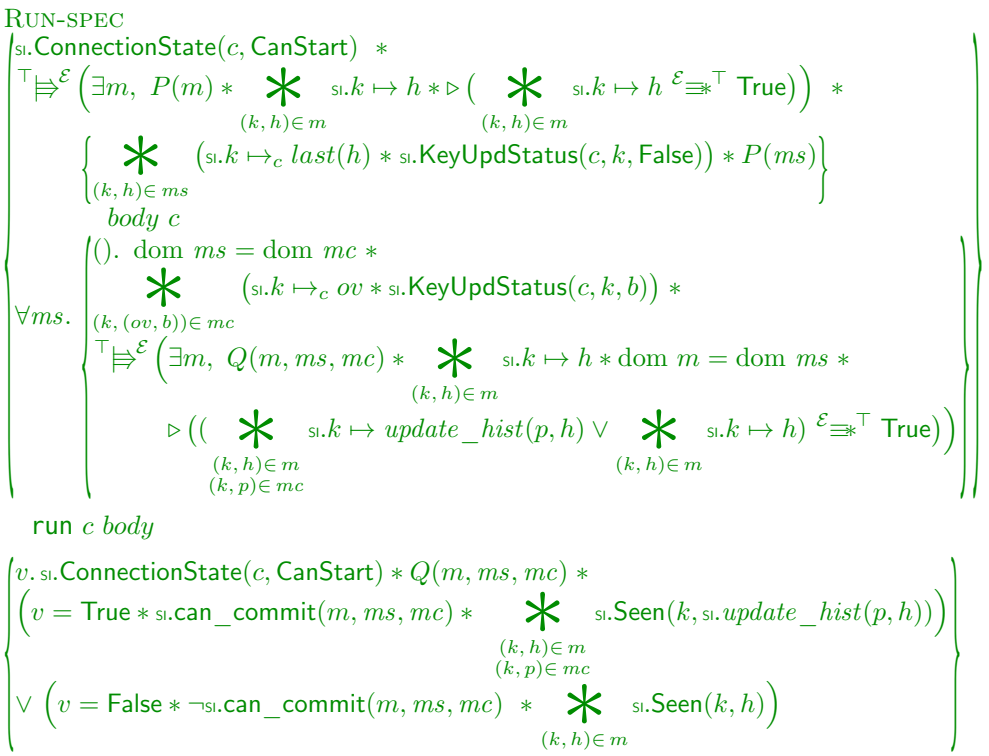


Figure 2.24: Run specification.

shifts used for taking a snapshot of the key-value store follows the same idea as in the precondition of the `wait` specification. The differences are that the persistence modality is not needed, as the view shift is only used once to start the transaction, and also that the snapshot can be described by a user-defined predicate P. Having started the transaction, the `run` specification will have to reason about the body of the transaction based on the snapshot described by P. As the body of the transaction is provided by the user of the specification, the RUN-SPEC rule includes in its precondition a Hoare triple for the transaction body. The Hoare triple for the transaction body has itself as precondition connection points-to resources and update status resources corresponding to the ones obtained from starting a transaction using the first two resources of the RUN-SPEC rule precondition. From using the transaction body Hoare Triple, the connection points-to and updates status resources potentially gets modified. The changes are captured in the map mc in the postcondition of the triple. For committing the transaction, based on the modifications in mc and the snapshot

ms, the RUN-SPEC rule uses the two view shifts it also obtains from using the transaction body Hoare triple. These view shifts describe exactly how one can obtain, possible by opening invariants, a commit snapshot of the key-value store and close any invariants using one of two snapshots: The unmodified commit snapshot corresponding to the unsuccessful commit case, or the commit snapshot with the updates described by *mc* corresponding to the successful commit case. The postcondition of the RUN-SPEC rule reflects the commit status of the transaction body: The return status of the `run` operation is attached to `can_commit` expressed using a commit snapshot m, a start snapshot *ms* and modifications *mc*. All three maps are related by the user-provided predicate Q, just as in the postcondition of the transaction body. For excluding future key-value store states, `Seen` predicates are also returned based on the commit status of the transaction body.

While the `run` specification for snapshot isolation is complex, and most likely it is more convenient to reason about transactions running under snapshot isolation using the per operation specifications of Figure 2.10, the `run` specifications for read committed and read uncommitted would be far more complex. As the specifications of read and write access shared key-value store state in read uncommitted and read committed, a `run` specification must take as argument resources in the form of view shifts that can be used to reason about the access to shared key-value store state for all the operations of the transaction body — not only the start and commit operations as in the `run` specification for snapshot isolation Figure 2.24.

2.11 Appendix: Example Portfolio

In addition to the examples presented in the body of the paper, we have verified a number of additional examples for the strongest isolation level of snapshot isolation displaying important properties of snapshot isolation. All the examples have been proven in Rocq using our specifications in Figure 2.10 and can be found in the accompanying Rocq files. The invariant is listed together with each example, unless an invariant was not needed. This is contrary to the body of the paper where we sometimes omit invariants for presentation purposes. In the following, we will attach some comments to each of the examples.

The *atomic transactions* example in Figure 2.25 xpress that transactions are atomic using an assert statement: Either all writes in a transaction are committed, or no writes are committed — it is not possible to read one value from one transaction and another value from the other transaction.

The *convenience of points-to* example in Figure 2.26 shows the convenience of working with points-to predicates as it allows for modularity: One can make a specification for the function f that only mentions the points-to-predicate for the key it takes as argument.

The *disjoint writes* example in Figure 2.27 asserts that transactions writing

to disjoint sets of keys will commit successfully. This holds as the snapshot isolation commit-check in Figure 2.11 is only concerned with write-conflicts which can not be created from disjoint writes. Furthermore, the disjoint writes example shows the modularity of the specs: No invariant is needed as the state of the key-value store is expressed using points-to predicates. This allows us to distribute the points-to resources for keys x and y to the left and right transactions respectively.

The *read-only transaction* example in Figure 2.28 asserts that a read-only transaction will commit successfully. As mentioned before, this holds because the snapshot isolation commit-check is only concerned with write-conflicts.

The last example in Figure 2.29 displays the difference between snapshot isolation and serializability: Snapshot isolation is a weaker property than serializability as any execution of committed transactions allowed under serializability is also allowed under snapshot isolation. We can think of serializability as a special case of snapshot isolation where the start snapshot and the commit snapshot is the same. Now, this comes with the caveat that snapshot isolation guarantees that all transactions will be reading from a valid snapshot, while serializability gives no guarantees about transactions that do not successfully commit. The implication of this is that under snapshot isolation we can make assertions before a transaction has committed. If we refrain from considering that assertions must be made after successfully committing in serializability, we can not create an example, which is only provable under snapshot isolation as an algorithm implementing snapshot isolation can then behave exactly as one implementing serializability. To illustrate this point, observe that the left case in the disjunction of the assertion in Figure 2.29, $v_x + v_y = \text{-}2$, corresponds to the only execution not allowed under serializability. Namely, the execution in which the writing transactions have the same start snapshot. Had this case not been included in the assertion, we would only have been able to prove the (remaining) assertion for an algorithm implementing serializability, but not for one implementing snapshot isolation.

`start`	`start`	`start`
`write` x 1	`write` x 2	**let** $v_x =$ `read` x **in**
`write` y 1	`write` y 2	**let** $v_y =$ `read` y **in**
`commit`	`commit`	`assert`$(v_x = v_y)$
		`assert`(`commit`)

$$Inv \triangleq \exists h,\ \text{si}.x \mapsto h * \text{si}.y \mapsto h$$

Figure 2.25: Atomic transactions.

`start`	`start`
`write` x 1	**let** $r = f(x)$ **in**
`commit`	`write` y r
	`commit`

$$Inv \triangleq \exists h,\ \text{si}.x \mapsto h$$

Figure 2.26: The convenience of points-to.

`start`	`start`
`write` x 1	`write` y 1
`assert(commit)`	`assert(commit)`

Figure 2.27: Disjoint writes commit.

`start`	`start`
`write` x 1	`read` x
`commit`	`assert(commit)`

$$Inv \triangleq \exists h,\ \text{si}.x \mapsto h$$

Figure 2.28: Read-only transactions commit.

`start`	`wait` z 1	`wait` z 1	`wait` z 1
`write` x 1	`start`	`start`	`start`
`write` y 1	**let** $v_x =$ `read` x **in**	**let** $v_y =$ `read` y **in**	**let** $v_x =$ `read` x **in**
`write` z 1	If $(v_x = 1)$ then	If $(v_y = 1)$ then	**let** $v_y =$ `read` y **in**
`commit`	{`write` y (-1) }	{`write` x (-1) }	`assert`$(v_x + v_y =$ -2 $\vee$
	`commit`	`commit`	$v_x + v_y \geq 0)$
			`commit`

$$Inv \triangleq \exists h_x\ h_y\ h_z,\ \text{si}.x \mapsto h_x * \text{si}.y \mapsto h_y * \text{si}.z \mapsto h_z\ *$$
$$\Big((h_x = [\,]) \vee (\exists\ v_x, v_y.\ last(h_x) = \mathsf{Some}\ v_x * last(h_y) = \mathsf{Some}\ v_y\ *$$
$$(v_x = 1 \vee v_x = \text{-1}) * (v_y = 1 \vee v_y = \text{-1}))\Big)$$

Figure 2.29: Capturing the difference between snapshot isolation and serializability.

2.12 Appendix: Serializability Does not Imply Snapshot Isolation in Separation Logic

Consider an example of two transactions with a single write operation both writing to the same key. All executions for these transactions are valid under serializeability. For snapshot isolation, only some executions are: the execution in which the two transactions share the same starting snapshot is not. The notion of implication we use (which is the natural one when thinking of specifications in terms of Hoare Triples), requires us to show that any execution valid for serializability is also valid under snapshot isolation. Hence, the implication can not be proven. In other papers, the implication is sometimes interpreted as: if there exists an execution valid for serilizability, then there must exist some execution (for the same set of transaction of course) which is valid under snapshot isolation. Hence, using this definition the implication can be proven.

Chapter 3

Verifying Isolation Levels of Database Implementations for Free Using Separation Logic

Abstract

Modern databases are highly concurrent and provide transactions as a mean of grouping several database operations into atomically applied units. Database vendors and software engineers use isolation levels to describe the consistency guarantees of transactions. The popular isolation levels give weak guarantees, with intricate semantics, to optimize performance of applications. The problem of assuring that database implementations actually implement the isolation level guarantees that application developers build their systems upon has received a great deal of attention from the testing community. But until now, there exists no method for formally verifying that a database implementation actually implements the isolation level that database vendors says it provides. In this paper, we present a method for verifying that a database implements an isolation level: we derive isolation levels directly, as formalized in transactional consistency models by the database community, from the structure of separation logic specifications. By doing so, we consider all program executions that a database and arbitrary clients of the database could produce. The result is a so-called free theorem meaning that any database implementation, whose operations are verified against a specific set of separation logic specifications, actually implements its isolation level. As all proofs in this paper are mechanized in the Rocq proof assistant and build upon a detailed semantic model of program execution, we believe this contribution raises the bar for the achievable robustness of databases.

3.1 Introduction

Transactions, the idea of viewing a number of operations as a single unit, are a cornerstone of database theory and application — most, if not all, modern

databases support transactions (*e.g.,* Oracle, MySQL, Microsoft SQL Server, PostgreSQL and MongoDB). However, when many transactions run concurrently, the consistency guarantees provided by databases become subtle and counterintuitive. Isolation levels describe how isolated concurrent transactions are from each other, and it is the canonical way of presenting transactional guarantees. Production databases use weak isolation levels, such as *read committed* and *snapshot isolation*, as default, *e.g.,* all the aforementioned databases use weak isolation levels. In contrast to strong isolation, which is often referred to as serializability, weak isolation levels do not correspond to executing transactions one at time. It is important that software engineers who build applications, while assuming these isolation level guarantees, can be assured that databases actually implement the isolation level vendors claim. The database community has a long tradition of formalizing and precisely defining weak isolation level guarantees of abstract models of databases in so-called *transactional consistency models*. Existing formal correctness proofs of concrete implementations of databases, on the other hand, only argue informally that their specifications capture the expected isolation level.

Consistency models Much work has been conducted to create transactional consistency models to formally specify transaction isolation guarantees. These models often fall into one of three categories [Xiong, 2020]: operational semantics [Cerone et al., 2015b; Crooks et al., 2017; Kaki et al., 2017; Xiong, 2020], abstract executions [Burckhardt et al., 2012; Gilbert and Lynch, 2002; Herlihy and Wing, 1990; Ketsman et al., 2023], or dependency graphs [Adya, 1999; Adya et al., 2000]. Historically, the SQL standard of 1992 introduced four isolation levels (informally, an isolation level is a consistency guarantee between concurrently executing transactions) that SQL compliant systems must make available to its users: read uncommitted, read committed, repeatable read, and serializability [Papadimitriou, 1979]. The definitions of the ANSI SQL 1992 standard were given using so-called "phenomena", *i.e.,* examples showcasing behavior that should be prohibited when executing concurrent transactions. Critiques that the definitions were imprecise followed [Berenson et al., 1995], leading to the proposal of better phenomena and another popular isolation level called *snapshot isolation*. Adya [1999] and Adya et al. [2000] gave the first widely accepted formal definitions, using dependency graphs, for a plethora of isolation levels including the ANSI SQL isolation levels and snapshot isolation. Though extremely useful, dependency graph models are neither amenable to formal proofs that a concrete implementation of a database is correct with respect to such models, nor are they suitable for formal reasoning about correctness of clients of a database.[1]

[1]Here, by formal verification we mean proving correctness with respect to a detailed semantic model of program execution, *e.g.,* using a program logic à la Hoare proved sound wrt. such a semantic model.

Towards connecting consistency models with executable database implementations Crooks et al. [2017] present a state-based model for transactional consistency and prove that their specifications of different isolation levels are equivalent to those of the recognized dependency graph model. The advantage of the state-based model, compared to the dependency graph model, is that it is amenable to formal reasoning about implementations. The recent work of Soethout et al. [2020] makes the first steps in this direction by checking a TLA$^+$ specification of the state-based model against a database implementation (written in the non-executable PlusCal language inside TLA$^+$). TLA$^+$ is a specification language, used to model local and distributed systems, with a model checker to check temporal logic assertions. While this is a step towards closing the gap between transactional consistency models and executable code, we are not fully there yet for a number of reasons:

- There is a significant gap between TLA$^+$ implementations and concrete implementations, leaving room for implementation bugs even if the TLA$^+$ database implementation written in the non-executable PlusCal language itself is correct [Chandra et al., 2007].
- Correctness in Soethout et al. [2020] means model checking a given transactional workload against the PlusCal implementation, but checking all possible interleavings of significantly sized workloads is infeasible due to the runtime complexity.
- A TLA$^+$ specification of a system is not suitable for modular reasoning about clients, which may use other libraries in addition to the database API.

When it comes to testing databases for correctly implementing isolation levels, a splurge of papers have been published recently Biswas and Enea [2019]; Gu et al. [2024]; Huang et al. [2023]; Kingsbury and Alvaro [2020]; Liu et al. [2024]; Møldrup and Pavlogiannis [2025]; Pick et al. [2025]; Zhang et al. [2023]. The focus of these papers is on *black box testing*, meaning to check whether an isolation level is violated in an observed execution of a transactional workload from a database (notice the difference between checking one execution of a transactional workload and enumerating all interleavings as in Soethout et al. [2020]). For instance, Pick et al. [2025] check transactions for isolation levels violations against the model of Crooks et al. [2017], and Kingsbury and Alvaro [2020] uses the model of Adya et al. [2000]. While the application of black-box testing techniques on industrial database systems is impressive, they can only show the presence of bugs, not the absence of bugs — to this end, we need to use approaches with stronger guarantees, such as separation logic.

Recent work in separation logic Separation logic [Reynolds, 2002a] has been very successful in reasoning about the absence of bugs in concrete sys-

tems' implementations. In fact, the state-of-the-art separation logic, the Iris framework [Jung et al., 2015, 2016, 2018; Krebbers et al., 2017], has recently been used to give Hoare-style specifications and correctness proofs for database implementations of the strong isolation level serializability [Chang et al., 2023] as well as the weak isolation levels read uncommitted, read committed, and snapshot isolation [Alnor Mathiasen et al., 2025]. These papers build upon the Grove [Sharma et al., 2023] and Aneris Krogh-Jespersen et al. [2020] program logics, respectively, which are both instantiations of Iris with a network semantics. Grove works with a subset of the Go programming language, and Aneris uses a subset of OCaml. The specifications in these papers reason about real program executions and are highly modular. In fact, the specifications are parametric in the choice of resources; the implementation details of the separation logic resources used in the specifications are opaque to the clients using them. However, these formal proofs fall short because they make no connection to any of the transactional consistency models.

Contributions In this paper, **we contribute a method for verifying executable database implementations against their isolation levels**. We do so by formally deriving isolation levels, as formalized in the dependency graph and the state-based models, directly from the structure of the separation logic specifications from Alnor Mathiasen et al. [2025]. This is done by adapting the method of creating invariants on execution traces from Birkedal et al. [2021], where execution traces are derived from ghost-code instrumentation that can be removed at runtime: we formulate the state-based consistency model as a predicate on execution traces that must invariantly hold. Using the embedding of the state-based model as an invariant on execution traces, **we establish the connection with no additional assumptions about the verified database implementations, and for any possible usage by clients of the database**. We remark that as correctness is proven invariantly, for any client usage of the database, the correctness result holds for any workload of transactions, and thus it is a much stronger property than that provided by existing model checking approaches, which only work given a specific transaction workload. The connection is a so-called *free theorem*, as we prove the theorem only by assuming that the database implementations satisfy the specifications of Alnor Mathiasen et al. [2025]. Thus, our result only has to be proven once for each isolation level before all database implementations gains the result, this is no matter how complex the database implementation may be, *e.g.,* it can be distributed over many nodes and use advanced locking mechanisms, as long as it satisfies the separation logic specifications. We provide the proofs for the isolation levels read uncommitted, read committed and snapshot isolation in this paper.

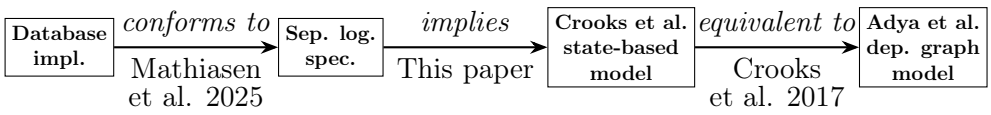
Concretely, in this paper we make the following technical contributions:

- A formulation and formalization of a realistic version of the state-based

transactional consistency model [Crooks et al., 2017]. Lifting the existing formulation of the model to a more realistic version, useable for reasoning about real transaction workloads, forces us to remove strong simplifying assumptions.

- We embed the state-based transactional consistency model into the Iris separation logic via invariants on traces of executions. This is done through multiple transformation phases using insights about linearization points of database operations.
- We prove that the separation logic specifications of the weak isolation levels read uncommitted, read committed, and snapshot isolation all adhere to their state-based model specifications by utilizing the specifications' parametricity in separation logic resources. Thus, we achieve modular per-operation specifications on the level of program executions that are verifiably correct with respect to the state-based consistency model and, by the equivalence in Crooks et al. [2017], the dependency graph model, with no observable changes in the specifications or runtime overhead. In essence, we derive isolation levels directly from the structure of separation logic specifications.
- All our results are mechanized on top of the Rocq proof assistant using the Iris [Jung et al., 2018] and Aneris [Krogh-Jespersen et al., 2020] frameworks.

Below, we visualize exactly how the contribution of this paper fits into the larger picture; the work of Alnor Mathiasen et al. [2025] invented separation logic specifications for transactional databases with weak isolation levels, and showed how to verify that an executable database implementation conforms to these specifications. This paper shows that any database which conforms to the separation logic specifications also conforms to the state-based consistency model of Crooks et al. [2017] without any additional proof obligations. As the state-based consistency model is proven equivalent to the dependency graph-based model in Crooks et al. [2017], we get that any database implementation verified in separation logic is correct as defined in the seminal papers of the dependency graph-based model Adya [1999] and Adya et al. [2000].



To create the implication from separation logic to the transactional consistency models, we reuse nothing but the specifications from Alnor Mathiasen et al. [2025], and the definition of the model from Crooks et al. [2017] (which we

must amend, *c.f.* the first technical contribution listed above). To prove the implication, we use the idea from Birkedal et al. [2021] of capturing trace properties solely based on separation logic specifications. Thus far, the technique has only been used in a concurrent setting for one illustrative example. What we do is much more challenging, as we embed the state-based consistency model of Crooks et al. [2017] as a trace property, which involves grouping several database operations, each of which has its own individual linearization point, into transactions.

Mechanization All of our results have been mechanized in the Rocq proof assistant, *i.e.,* all the proofs of our theorems have been machine-checked.[2] Together with the fact that all the proofs of the separation logic specifications of Alnor Mathiasen et al. [2025] have been machine-checked, and the fact that the soundness of the Iris program logic with respect to the semantic model of program execution has also been machine-checked, this means that **we obtain complete formal theorems in Rocq connecting the database implementations to the state-based transactional consistency models.** The Rocq formalization accompanies the paper and consist of more than 13 thousand lines of proof code.

3.2 Technical Overview and Prior Work

Database implementations The database implementations we consider in this paper, are databases written in AnerisLang. This is a general purpose programming language and a subset of OCaml with fork-based concurrency, mutable references, and networking primitives for communicating over UDP network sockets (Gondelman et al. [2023] builds a verified reliable communication library on top of UDP sockets in AnerisLang). We have included a subset of AnerisLang below, for the full details about the language, we refer to its documentation [Aneris-Team, 2023] which contains a detailed model of execution including the memory and network state.

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & v \mid x \mid e_1\ e_2 \mid \circledcirc_1\, e \mid e_1 \circledcirc_2 e_2 \mid \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \mid (e_1, e_2) \mid \mathbf{fst}\ e \\
& \mid \mathbf{snd}\ e \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid (\mathbf{match}\, e\, \mathbf{with}\, \mathbf{inl}\ x_1 \Rightarrow e_1 \mid \mathbf{inr}\ x_2 \Rightarrow e_2\, \mathbf{end}) \\
& \mid \mathbf{fork}\ \{e\} \mid \mathbf{ref}\ e \mid\ !e \mid e_1 \leftarrow e_2 \mid \mathbf{CAS}\ e_1\ e_2\ e_3 \mid \mathbf{makeaddress}\ e_1\ e_2 \\
& \mid \mathbf{getaddress}\ e \mid \mathbf{socket}\ \mid \mathbf{socketbind}\ e_1\ e_2 \mid \mathbf{sendto}\ e_1\ e_2\ e_3 \\
& \mid \mathbf{receivefrom}\ e \mid \ldots
\end{aligned}
$$

We consider database implementations with operations for starting and committing transactions together with operations for reading and writing to keys inside a transaction, that is `start` c, `read` $c\ k$, `write` $c\ k\ v$ and `commit` c, where c is a connection to the database acquired in an initialization phase. We

[2]The Coq proof assistant has recently been renamed to the Rocq prover.

make no assumptions about database implementations other than that the specific separation logic specifications from Alnor Mathiasen et al. [2025] have been proven for these operations; we go over the specifications in Chapter 3.4. Database implementations may be local or distributed and use all sorts of fancy concurrency measures.

Crooks model To verify the isolation levels of database implementations, we create a direct formal link to Crooks' state-based transactional consistency model [Crooks et al., 2017]. In the state-based model, isolation levels are formalized as predicates on executions [Crooks et al., 2017]. An execution in this context is a sequence of state and transaction pairs, a state is mapping from keys of the database to the values they point to. For any pair of a state and a transaction (s_i, T_i) in an execution E, s_i is the result of applying the write operations of T_i to the previous state s_{i-1} in E (we consider transactions as transitions between states). For any given isolation levels I, we have a predicate P_I on executions. If for all the transactions a database produces, we can build an execution E out of those transactions for which $P_I(E)$ is true, then the database implements isolation level I. We formalize a more realistic version of the state-based model in Chapter 3.3, where we remove a few simplifying assumptions from the original presentation to be able to reason about program executions of database implementations.

Extraction of transactions through ghost code instrumentation To establish that a database implementation written in AnerisLang implements an isolation level I, we must extract all the transactions it can produce. Extracting transactions is not trivial, as we need to extract any combination of transactions a database possibly can produce by interacting with clients programs of the database, and we can not make any assumptions about the behavior of clients. The technique of Birkedal et al. [2021], which so far only has been applied to small synthetic examples, utilizes ghost-code instrumentation to capture trace properties (everything happens statically, and no code is ever run). To use this technique, we have taken AnerisLang and extended its program state with a global trace t of so-called *events*. There are two types of events, *pre events* and *post events*, marking the start and end of a database operation. To create such events, AnerisLang is augmented with two ghost-code operations: **`fresh`** and **`emit`**. Using these ghost-code operations, we instrument each of the database operations to emit a pre event and a post event into the global trace t, see *e.g.,* the instrumentation of the `read` operation below:

$$\begin{aligned}\mathbf{Wrap}(\texttt{read}\ c\ k) \triangleq\ & \mathbf{let}\ \tau = \texttt{fresh}\ (c, \mathbf{R}, k)\ \mathbf{in}\\ & \mathbf{let}\ ov = \texttt{read}\ c\ k\ \mathbf{in}\\ & \texttt{emit}\ (\tau, c, \mathbf{R}, k, ov); ov\end{aligned}$$

Above, **`fresh`** $(c, \mathbf{R}, k)$ emits the pre event $(\tau, c, \mathbf{R}, k)$ into the global trace where $\mathbf{R}$ is a label signifying that it is a pre event of a `read` operation, and τ

is a fresh identifier that we use to tie a pair of pre- and post events together; **emit** $(\tau, c, \mathbf{R}, k, ov)$ emits the post event $(\tau, c, \mathbf{R}, k, ov)$ into the global trace without generating a new identifier, this time we use the identifier τ from the pre event. The fundamental insight that we utilize in this paper, is that a trace of pre events and post events is enough information to verify that a database implements an isolation levels. **This is the core technical contribution of the paper: we can create an invariant on a trace of events that lets us reason about transactions in Crooks model**. The invariant is shown in Figure 3.1, which in essence is our extraction method for extracting any potential transactions from a database implementation. We explain the details of the invariant, and the formulation of our main theorem (Theorem 3.5.1), in Chapter 3.5. Briefly, level 1 is our trace of pre- and post-events from the ghost code instrumentation, level 2 are linearization points created as part of the proof for each pair of pre- and post-events, level 3 is grouping of the linearization points into transactions that we can reason about in the state-based model at the top level. At the top level, we build an execution E and show that $P_I(E)$ holds for the isolation level I that the database implements.

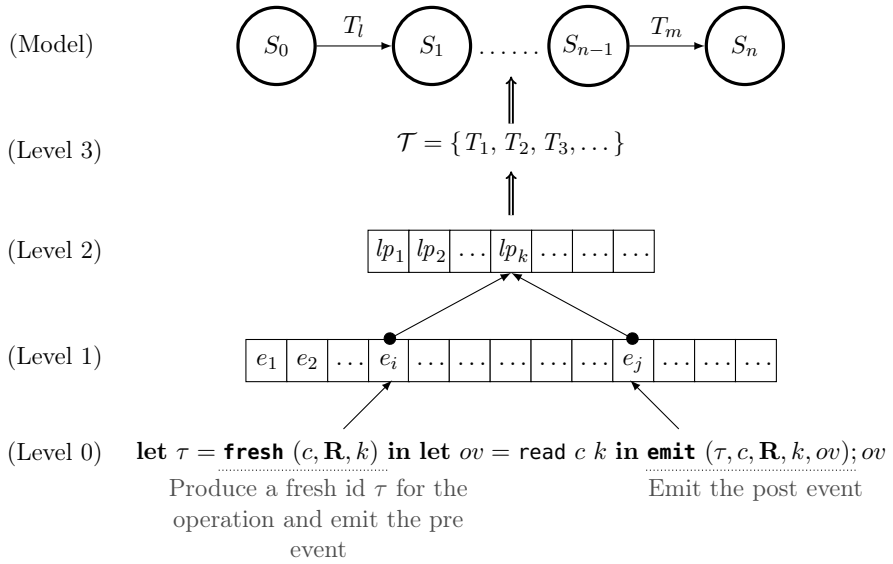


Figure 3.1: Extraction method exemplified with the read operation.

Proof obligations in separation logic To show that the invariant holds for all possible usages of a database implementation, our proof obligation becomes to prove that emitting the pre event and emitting the post event into the global trace does not break the invariant for each of the database operations; where we get to assume that the database operations satisfy their separation logic specifications. Thus, in summary, we are proving isolation levels from the structure of separation logic specifications alone. In Chapter 3.6, we present

the proof structure in detail and carry out a detailed proof sketch for snapshot isolation in Chapter 3.7. To explain what we mean by only relying on the structure of separation logic specifications, consider the specifications for a very simple database below:

$$\{\mathsf{idle}\}\ \texttt{start}\ c\ \{\mathsf{active}\} \qquad \{\mathsf{active}\}\ \texttt{commit}\ c\ \{\mathsf{idle}\}$$
$$\{\mathsf{active}\}\ \texttt{write}\ c\ k\ v\ \{\mathsf{active}\} \quad \{\mathsf{active}\}\ \texttt{read}\ c\ k\ \{\mathsf{active}\}$$

The preconditions and postconditions of these specifications only assert whether there is an active transaction or not. But, from the structure of these specifications, we can, for instance, deduce that the `read` and `write` operations never can be used before the `start` operation, and that a transaction can not be committed twice. The specifications of Alnor Mathiasen et al. [2025] are, of course, much more involved and they contain enough information to let us prove the invariant on the global trace that encapsulates isolation levels in Crooks' state-based model.

Comparison with other proof-oriented approaches for non-executable code Outside the world of separation logic, Raad et al. [2019] have created two high-level non-executable snapshot isolation reference implementations, as local software systems on the release/acquire memory model, and made a pen and paper proof in their framework that the reference implementations are correct regarding the model of snapshot isolation of Cerone and Gotsman [2018] (this model is based on dependency graphs and is a variant of Adya [1999] and Adya et al. [2000]). Similar work has also been done by Raad et al. [2018] for a variant of snapshot isolation called *parallel* snapshot isolation. Unlike the approach in this paper, other than being non-mechanized and non-executable, this line of work does not address verifying executable clients against the reference implementations. Neither does it come with the modularity of separation logic to be usable in conjunction with other software libraries, which makes it unfeasible to use for developments of significant size. It also remains unclear how one would verify other implementations than the reference implementations without having to undergo the same proof effort. The work of Doherty et al. [2013] (and comparable approaches like Lesani et al. [2012] and Schellhorn et al. [2025]) has similar limitations: Transactional guarantees are specified as an I/O automaton, and they prove that a transactional memory implementation, also written as an I/O automaton, is a *forward simulation* of the specification. Analogously, Attiya et al. [2013] (and the successor work in Attiya et al. [2017]) create *observable refinements* between transactional memory implementations written in a non-executable programming language. In Lesani et al. [2022b], a different approach is taken based on *interaction trees* [Xia et al., 2019]; the paper presents a mechanized framework for verifying classical concurrent libraries with linearizability [Herlihy and Wing, 1990] alongside transactional libraries. The specifications of the transactional libraries are restricted to serializability

formulated as linearizability of transactions (informally, serializability is the guarantee that transactions appear to be executed one after another), and the approach is applied to *Transactional Mutex Locks* [Dalessandro et al., 2010].

In Figure 3.2, we compare the contributions of this paper with the other proof-oriented approaches for showing the absence of bugs in database or software transactional memory (STM) implementations. We compare across six axes: (1) *Executeable code* (is the verified database/STM executable?), (2) *Generalizes over clients* (does the verification prove that all potential transactional workloads created by clients of the database/STM break no guarantees?), (3) *Generalizes over impls.* (do all future verifications have to undergo the same proof effort?), (4) *Abstract spec.* (is there a formal connection to an abstract transactional consistency model?), (5) *Modular* (is it a modular proof that can easily be combined with other verified efforts?) and (6) *Mechanized* (is the proof mechanized in a proof assistant?). As seen in Figure 3.2, this paper is the only method that is partially able to generalize over database implementations. Our approach is agnostic to the concrete implementation and only assumes that an implementation is verified against the separation logics specifications of Alnor Mathiasen et al. [2025], which make no assumptions about implementation details, or whether the system consists of one or more servers. Our results establish a connection from the separation logic specifications to the transactional consistency model of Crooks et al. [2017] stated as a free theorem that does not have to be reproved for other implementations as long as they satisfy the specifications of Alnor Mathiasen et al. [2025].

	Executable code	Generalizes over clients	Generalizes over impls.	Abstract spec.	Modular	Mechanized
Attiya et al. [2013]	○	○	○	○	○	○
Doherty et al. [2013]	○	○	○	○	○	●
Raad et al. [2019]	○	●	○	●	○	○
Soethout et al. [2020]	○	○	○	●	○	●
Lesani et al. [2022b]	○	●	○	○	●	●
Chang et al. [2023]	●	○	○	○	●	●
Alnor Mathiasen et al. 2025	●	○	○	○	●	●
This paper	●	●	◐	●	●	●

Figure 3.2: Comparison of prior work about verification of database and software transactional memory implementations.

3.3 A Realistic State-based Model

The original formulation of the state-based model, presented in Crooks et al. [2017], makes a number of simplifying assumptions to ease its presentation. While this is a natural thing to do, we need to remove some of these assumptions as they are overly restrictive for capturing the behavior of clients interacting

with actual database implementations. Concretely, the assumptions we remove are:

- Every value written (as a part of a write operation) is unique among all other values written.
- There is at most one write operation per transaction for each key.

While these simplifying assumptions can seem somewhat minor, removing them requires some definitions to be reformulated (we will show how shortly). Contrary to the above, there is one assumption we do not remove:

- All transactions read from their most recent write (if there is one).

It is important to understand that we do still prove that transactions from a database implementation verified with our approach read from their most recent write, but we prove it prior to reasoning at the transactional model level. We provide more details on this when we explain our method for extracting transactions from executable code in Chapter 3.5. In the rest of this section, we formalize the state-based model, and along the way we explain the changes we have made to remove the above-mentioned assumptions. Afterwards, we present a formal definition of what it means for a database implementation to implement an isolation level in this model.

Transactions Consider transaction operations on a set of keys and values. We work with the transaction operations read, write and commit. $\mathbf{R}(id, k, ov)$ denotes a read operation reading the optional value ov from key k and having identifier id, similarly $\mathbf{W}(id, k, v)$ denotes a write operation writing value v to key k and having identifier id. We assume each operation has a unique identifier. The commit operation is denoted $\mathbf{C}(id, b)$ with the boolean b indicating whether the transaction commits or aborts. We define a transaction as a set of operations together with a total order on the operations representing the program order:

Definition 3.3.1 (Transaction)**.** *A* transaction T *is a tuple of* $(\mathsf{O}_T, <_T)$ *where* O_T *is a set of operations and the program order* $<_T$ *is a total order on* O_T*.* O_T *must contain exactly one* commit *operation, and it must be the greatest element in* $<_T$*.*

We remark that removing the simplifying assumptions listed in the beginning of this section is not enough to allow for all realistic client behavior: because transactions are defined as having a *set* of operations, we have added the (unique) identifier component to transaction operations, to allow for multiple operations having the same keys and values in a single transaction. Transaction operations usually do not have an identifier component in other transactional models. In Chapter 3.5, we show how unique identifiers for transaction operations are created.

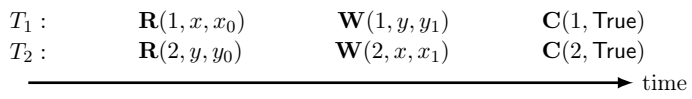


Figure 3.3: The *write skew* example.

Isolation levels Having formalized what a transaction is, the next step is to define transactional guarantees, *i.e.,* isolation levels. We can think of an isolation level as all the executions of a transactional workload we will allow. Throughout this paper, we primarily focus on one particular isolation level, namely snapshot isolation, to explain our ideas. Snapshot isolation, together with the two other isolation levels we treat in this paper, read uncommitted and read committed, fall into the category of weak isolation levels. Strong isolation, in the form of serializability, is the isolation level where each transaction gets to execute one at a time with no overlapping. Snapshot isolation is more relaxed than serializability and hence weaker. The *write skew* example is commonly used to explain how snapshot isolation differs from serializability. In the *write skew* example, we observe two concurrent transactions T_1 and T_2 that both perform a read operation followed by a write operation before they commit. As seen in Figure 3.3, transaction T_1 reads from the key x and writes to the key y, whereas it is the opposite for transaction T_2. Both transactions read from the initial snapshot of the database, namely the snapshot where all keys have not been written to yet and therefore return None when read from. For notational convenience, we often omit the optional part from the result of a read operation; instead, *e.g.,* for the key x, we write x_0 for the initial version corresponding to None. We denote subsequent versions of a key with an index larger than zero. Because neither of the two transactions in the *write skew* example see the effects of the other, this example cannot come from an execution where transactions execute one at a time. Therefore, it is not allowed under serializability. But it is allowed under snapshot isolation. The principle of snapshot isolation is that *all transactions read from a snapshot of the database, and any transaction is allowed to commit when it has no write conflicts with other concurrent transactions* [Berenson et al., 1995]. By a write conflict, we mean two transactions that write to the same key. Notice that snapshot isolation, and any other isolation level, always have the baseline guarantee, mentioned in the beginning of this section, that transactions read their own writes.

While an example such as the *write skew* can help us strengthen our intuition and explain the concept of snapshot isolation, we need a transactional model to formally decide whether an arbitrary execution of a transactional workload is considered safe. In the state-based model, we use a labeled state transition system and, on top of that, we define the notion of *executions.*

Definition 3.3.2 (States and Executions)**.** *A* state *s is a mapping from keys to*

values. We write $s[k]$ for the value of key k in state s. We define a transition relation *for transactions which we write as $s \xrightarrow{T} s'$ when the state s' results from running (the* write *operations of) transaction T starting from state s. An* execution *E is a sequence of state-transaction pairs, $(s_1, T_1), (s_2, T_2), \ldots, (s_n, T_n)$ such that $\forall i < n, s_i \xrightarrow{T} s_{i+1}$. We write $E[i]$ for the pair (s_i, T_i) in the execution sequence.*

To express that an execution E is based upon a set $\mathcal{T}$ of transactions, we write $\mathsf{CoherenceExec}(\mathcal{T}, E)$, which formally expresses that no transactions occur more than once in an execution E and $\forall T, (\exists i,\ E[i] = (\_, T)) \Leftrightarrow T \in \mathcal{T}$.

For the *write skew* example in Figure 3.3, we show its two possible executions in Figure 3.4.

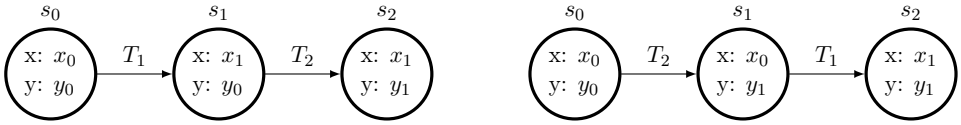


Figure 3.4: The *write skew* example in the state-based model with its two possible executions.

Both executions start from the initial state, here we represent the initial values as version zero, *e.g.,* x_0, but they differ in which transaction applies its write first. The fact that snapshot isolation allows the *write skew* example means that it contains both executions in Figure 3.4. We define an *isolation level* to be a set of executions.

Commit tests Whether an execution is in an isolation level I is determined by a *commit test* [Crooks et al., 2017], *i.e.,* a predicate on executions and transactions, written as $\mathsf{CT}_I(T, E)$. An execution E, for the subset of committed transactions in $\mathcal{T}$ (*i.e.,* assuming $\mathsf{CoherenceExec}(\mathsf{comTrans}(\mathcal{T}), E)$ holds where $\mathsf{comTrans}(\mathcal{T})$ is the subset of committed transactions in $\mathcal{T}$), is in an isolation level I when the test holds for all the commited transactions in $\mathcal{T}$, *i.e.,* $\forall T \in \mathsf{comTrans}(\mathcal{T}),\ \mathsf{CT}_I(T, E)$. We define the commit test for the weak isolation levels read uncommitted, read committed and snapshot isolation using a number of predicates. Figure 3.5 summarises the definition of the commits tests.

Starting with snapshot isolation, we write $\mathsf{HasWrite}(T, k)$ to say that the transaction T contains at least one write operation with key k, and write $\mathsf{NoPriorWrite}(T, op)$ to state that in transaction T there is no write operation on the same key as the operation *op* prior to *op* itself. Using these definitions we define the predicates Complete and NoConf in Chapter 3.3.3 which are used in the commit test of snapshot isolation. The $\mathsf{Complete}(E, (\mathsf{O}_T, <_T), s, i)$ predicate asserts that a transaction $(\mathsf{O}_T, <_T)$ has a snapshot in the past of the execution E. The snapshot is represented by the state s, while we use i as the

index of s in E. The $\mathsf{NoConf}(E, T, i)$ predicate asserts that the transaction T, with snapshot at index i in E, has no write conflict with any concurrent transaction. Here a concurrent transaction is one that has committed in between the start time and the commit time of transaction T.

Definition 3.3.3. *We define the predicates* Complete*, and* NoConf *as follows:*

For a transaction $(\mathsf{O}_T, <_T)$*, execution* E*, state* s*, and index* i

$$\begin{aligned}\mathsf{Complete}(E, (\mathsf{O}_T, <_T), s, i) \triangleq\ & \forall j\ id\ k\ ov,\ E[j] = (\_, (\mathsf{O}_T, <_T)) \Rightarrow \\ & \mathbf{R}(id, k, ov) \in \mathsf{O}_T \Rightarrow \\ & \mathsf{NoPriorWrite}((\mathsf{O}_T, <_T), \mathbf{R}(id, k, ov)) \Rightarrow \\ & i < j \wedge s[k] = ov.\end{aligned} \tag{Complete}$$

For a transaction T*, execution* E*, and index* i

$$\begin{aligned}\mathsf{NoConf}(E, T, i) \triangleq\ & \forall j\ i'\ T'\ k,\ E[j] = (\_, T) \Rightarrow \\ & E[i'] = (\_, T') \Rightarrow \\ & i < i' < j \Rightarrow \mathsf{HasWrite}(T, k) \Rightarrow \neg\mathsf{HasWrite}(T', k).\end{aligned} \tag{NoConf}$$

The commit test of snapshot isolation in Figure 3.5 then states that all transactions in a valid execution E must have a snapshot state for which the Complete and NoConf predicates hold.

The read committed isolation level is weaker than snapshot isolation. It guarantees that any read by a committed transaction will be from its own previous write, and if no previous write exists, it will be from another committed transaction. Notice it does not guarantee that transactions read from a stable snapshot, just that reading happens from *some* committed transaction. To define the commit test of read committed, we use the predicates ReadState and Preread which we define in Chapter 3.3.4. The $\mathsf{ReadState}\big(E, i, k, v, s\big)$ predicate asserts that the state s has an index j in the execution E which is smaller than the argument i, and that key k points to value v in the state. The $\mathsf{Preread}((\mathsf{O}_T, <_T), E)$ predicate says that all the read operations in a transaction $(\mathsf{O}_T, <_T)$, with no prior writes to the same key, must have a read-state, as defined by the ReadState predicate, in the execution E. The commit test of read committed then states in Figure 3.5 that all transactions in a valid execution E must satisfy the Preread predicate.

Definition 3.3.4. *We define the predicates* ReadState *and* Preread *as follows:*

For an execution E*, index* i*, key* k*, value* v *and state* s

$$\mathsf{ReadState}\big(E, i, k, v, s\big) \triangleq \exists j, j < i \wedge E[j] = (s, \_) \wedge s[k] = v. \tag{ReadState}$$

For a transaction $(\mathsf{O}_T, <_T)$ *and execution* E

$$\begin{aligned}\mathsf{Preread}((\mathsf{O}_T, <_T), E) \triangleq\ & \forall i\ id\ k\ v,\ E[i] = (\_, (\mathsf{O}_T, <_T)) \Rightarrow \\ & \mathbf{R}(id, k, v) \in \mathsf{O}_T \Rightarrow \\ & \mathsf{NoPriorWrite}((\mathsf{O}_T, <_T), \mathbf{R}(id, k, v)) \Rightarrow \\ & \exists s, \mathsf{ReadState}\big(E, i, k, v, s\big).\end{aligned} \tag{Preread}$$

The last isolation level, read uncommitted, can be understood as a baseline level in which we do not get any additional guarantees than our baseline transaction assumptions (transactions read from their most recent write). Thus, the commit test for read uncommitted is simply True. The definitions of the read uncommitted and read committed levels are equivalent to the ones stated in the original formulation of the state-based model [Crooks et al., 2017]. The definition of the snapshot isolation level has been changed as a result of removing the simplifying assumptions listed in the beginning of this section. Specifically, the previous formulation of the NoConf predicate did not take into account that the same value can be written multiple times.

Isolation Level	Commit Test
Read Uncommitted $\big(\mathsf{CT}_{RU}(T, E)\big)$	True
Read Committed $\big(\mathsf{CT}_{RC}(T, E)\big)$	$\mathsf{Preread}(T, E)$
Snapshot Isolation $\big(\mathsf{CT}_{SI}(T, E)\big)$	$\exists s\ i,\ E[i] = (s, T) \wedge$ $\mathsf{Complete}(E, T, s, i) \wedge \mathsf{NoConf}(E, T, s)$

Figure 3.5: Isolation levels defined using state-based commit tests.

Correctness of implementation In this paper, our goal is to show correctness of database implementations. The correctness criterion we use is a straightforward extension of the correctness criterion used by Crooks et al. [2017] to prove equivalence between their state-based model and the dependency graph model.

Definition 3.3.5 (Implementation Correctness)**.** *A database implements an isolation level I when, for any set of transactions $\mathcal{T}$ it produces, there exists an execution E such that* $\mathsf{CoherenceExec}(\mathsf{comTrans}(\mathcal{T}), E)$ *holds, and we have* $\forall T \in \mathsf{comTrans}(\mathcal{T}),\ \mathsf{CT}_I(T, E)$.

We recognize that the word "produces" leaves some ambiguity in the definition. We make this notion precise in Chapter 3.5. Futher, a database in this paper is a program that implements the API presented in Chapter 3.4.1. The takeaway from this section is that we have now defined correctness at the transactional model level.

3.4 Specifying Transactional Consistency in Separation Logic

In this section, we recall the formulation of transactional consistency for program executions in separation logic from Alnor Mathiasen et al. [2025]. The paper presents separation logic specification for the three weak isolation levels read uncommitted, read committed and snapshot isolation. The works that we build

upon [Alnor Mathiasen et al., 2025; Birkedal et al., 2021; Krogh-Jespersen et al., 2020] are all based on the Iris program logic framework [Jung et al., 2018], a state-of-the-art framework for specifying and proving correctness of programs using separation (Hoare) logic [Hoare, 1969; Reynolds, 2002a]. We will present the (minor) details of separation logic (and the Iris framework specifically) that are necessary for understanding this paper on per-need basis.

In this paper, we focus on snapshot isolation to explain our methodology for connecting abstract transactional consistency models to program executions. We emphasize again that our approach has been formalized for all the isolation levels in Alnor Mathiasen et al. [2025], and refer to the Rocq formalization for the details about the remaining isolation levels. In Chapter 3.4.1, we explain the structure of the specification for weak isolation levels with a focus on parametricity of resources. This is a property we rely on, in Chapter 3.5 and Chapter 3.6, to achieve our connection to the state-based model with no observable changes in the specifications. After this, we go into detail with the per-operation specifications for snapshot isolation in Chapter 3.4.2 and explain how they expose linearization points through atomic specifications — another crucial property for establishing our connection to the state-based model. For readers unfamiliar with separation logic, we show how the per-operation specifications can be used to prove safety of small examples, exemplified with the *write skew* example, in Appendix 3.10. Proving safety of small examples that capture interesting properties about isolation levels is, until the contribution of this paper, the best assurance we have that separation logic specifications capture isolation levels correctly.

3.4.1 Specification Structure

The snapshot isolation specification ($spec_{SI}$) is shown in Figure 3.6. (Our presentation of the specification omits resources and specifications related to initialization of the database. These elements are not of importance to the proofs of this paper, because the correctness criteria (Chapter 3.3.5) we prove is only about what is observable to clients of the database.) The specification for read uncommitted and read committed have the same structure. The specification takes as argument a transactional library *lib*. This is the client API of the database implementation with the method signatures $\mathsf{init} : \mathsf{ip} \to \mathsf{conn}$, $\mathsf{start} : \mathsf{conn} \to \mathsf{unit}$, $\mathsf{read} : \mathsf{conn} \times \mathsf{key} \to \mathsf{option\ val}$, $\mathsf{write} : \mathsf{conn} \times \mathsf{key} \times \mathsf{val} \to \mathsf{unit}$ and $\mathsf{commit} : \mathsf{conn} \to \mathsf{bool}$, where conn is the type for connections. Any implementation verified against the specification $spec_{SI}$ must have these signatures, as this is what the specification assumes. The specification starts by stating the existence of a number of resources. These are all Iris predicates (as signified by the type iProp, which is the type of Iris propositions). The first resource, $\mathsf{ConnectionState} : \mathsf{Conn} \times \mathsf{Status} \to \mathsf{iProp}$, is used to keep track of the state a client is in. A client can create this resource using the operation init, and it will show whether a client has an actively running transaction, on

its connection of type Conn, using a Status type argument, which we will see examples of shortly. The two resources, $\_ \mapsto \_ : \mathsf{Key} \times \mathsf{Hist} \to \mathsf{iProp}$ and $\_ \mapsto_{\_} \_ : \mathsf{Conn} \times \mathsf{Key} \times \mathsf{Val} \to \mathsf{iProp}$ are so called *points-to resources*; they both have a Key arguments which points to either a history or a value. We call $\_ \mapsto_{\_} \_ : \mathsf{Conn} \times \mathsf{Key} \times \mathsf{Val} \to \mathsf{iProp}$ a *local* points-to resource. It is used for describing the starting snapshot that a transaction reads from, thus it also takes a connection as argument. We name $\_ \mapsto \_ : \mathsf{Key} \times \mathsf{Hist} \to \mathsf{iProp}$ the *global* points-to resource. It describes the state of the database, where a history is a list of values holding all the values that has been written in the past. The resource $\mathsf{KeyUpdStatus} : \mathsf{Conn} \times \mathsf{Key} \times \mathsf{Bool} \to \mathsf{iProp}$ is used to keep track of whether a transaction made any updates to its local points-to resources. This is essential information to have when resolving whether a transaction has any write conflicts and safely can commit. The specification $spec_{SI}$ provides $\ast_{k \in \mathsf{Keys}_{\mathrm{SI}}} k \mapsto [\,]$ which are initial *global* points-to resources representing the state in which no value has been written to any key yet. The symbol $\ast$ is the iterated separating conjunction. The separating $*$ conjunction is much like an ordinary conjunction, except that any exclusively owned resources can only be used to satisfy one side of the conjunction, *i.e.,* it enforces a separation of resources.

Following the listing of all resources comes the per-operation separation logic specifications in the form of Hoare triples for each operation in the client API which the argument $spec_{SI}$ is an implementation of. Concretely, we have a Hoare triple specification for the operations init, start, read, write and commit. The operations read, write and commit, correspond to the ones we saw in our formalization of the state-based model in Chapter 3.3, while the start operation starts new transactions, and the init operation initializes a client connection with the transactional library.

$$
\begin{aligned}
spec_{SI}(lib) \triangleq\ & \exists\ \mathsf{ConnectionState} : \mathsf{Conn} \times \mathsf{Status} \to \mathsf{iProp},\\
& \_ \mapsto \_ : \mathsf{Key} \times \mathsf{Hist} \to \mathsf{iProp},\\
& \_ \mapsto_{\_} \_ : \mathsf{Conn} \times \mathsf{Key} \times \mathsf{Val} \to \mathsf{iProp},\\
& \mathsf{KeyUpdStatus} : \mathsf{Conn} \times \mathsf{Key} \times \mathsf{Bool} \to \mathsf{iProp}.\\
& \ast_{k \in \mathsf{Keys}} k \mapsto [\,] \ \wedge \textsc{si-init-client-spec holds} \wedge\\
& \textsc{si-start-spec holds} \ \wedge \textsc{si-write-spec holds} \wedge\\
& \textsc{si-commit-spec holds} \ \wedge \textsc{si-read-spec holds}
\end{aligned}
$$

Figure 3.6: Specification for snapshot isolation.

Parametric resources As mentioned above, all resources in the specification are existentially quantified. All a user of the specification knows about the

resources is that they work with the per-operation specifications. Thus, any client program, which is verified using $spec_{SI}$, can work with any database implementation that implements the transactional library as specified in $spec_{SI}$. When our proofs in later sections of the paper modifies the resources of $spec_{SI}$, it is not observable to any client because we make sure the modified resources can still satisfy $spec_{SI}$, and due to the existential quantification, the client will never know that the underlying resources have changed. Further, we preemptively emphasize that our proofs, showing that a database implementation *lib* is correct w.r.t. the state-based model, do not assume anything about the implementation except that it implements specification ($spec_{SI}(lib)$).

3.4.2 Per-operation (Atomic) Specifications

The per-operation specifications for all the database operations are shown in Figure 3.7. All operations, except the init operation, are not specified using ordinary Hoare triples but using so-called *atomic* Hoare triples. To explain the difference between the two, we first recall that a Hoare triple $\{P\}\, e\, \{v.\, Q\}$ says that if the resources in P are satisfied, the expression e is safe to execute, and if e terminates the resources in Q will hold, with the return value of e bound to v in Q. With a Hoare triple $\{P\}\, e\, \{v.\, Q\}$, the resources in P are consumed by the duration e executes and *only* when e terminates, the resources in Q are given back to the user of the specification. In highly concurrent systems where resources are shared using invariants, this is not strong enough. Thus, we specify operations using atomic Hoare triples $\langle P\rangle\, e\, \langle v.\, Q\rangle$ [da Rocha Pinto et al., 2014]. Intuitively, an atomic Hoare triple asserts that the expression being specified has a linearization point around which the resources P are transformed to Q. Thus, around atomic Hoare triples, we are allowed to soundly open invariants as shown in Inv-atomic. An invariant holding a resource R is written

$$\frac{\textsc{Inv-atomic}\quad \vdash \langle R * P\rangle\, e\, \langle v.\, Q * R\rangle}{\boxed{R} \vdash \langle P\rangle\, e\, \langle v.\, Q\rangle}$$

$\boxed{R}$. From the exclusive ownership of a resource R, we can create an invariant $\boxed{R}$ which does not assert exclusive ownership of R any longer. Instead, $\boxed{R}$ is *duplicable* and can be freely shared. Usually, invariants in Iris can only be opened around expressions that can step to a value in one step according to the operational semantics, atomic Hoare triples lets us lift this restriction. Next, we go into details with each of the operation specifications.

Init Starting with the init operation, used for initializing a client connection, it takes no precondition and returns the $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ resource. The resource $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ reflects that there is currently no active transaction on the connection c, and that a new transaction can be started using the start operation.

SI-INIT-CLIENT-SPEC

$$\{\mathsf{True}\}\ \texttt{init}\ srv\ \{c.\ \mathsf{ConnectionState}(c, \mathsf{CanStart})\}$$

SI-WRITE-SPEC

$$\langle k \mapsto_c ov * \mathsf{KeyUpdStatus}(c, k, b)\rangle$$

$$\texttt{write}\ c\ k\ v$$

$$\langle (). k \mapsto_c \mathsf{Some}\ v * \mathsf{KeyUpdStatus}(c, k, \mathsf{True})\rangle$$

SI-READ-SPEC

$$\langle k \mapsto_c ov\rangle\ \mathsf{read}\ c\ k\ \langle ov. k \mapsto_c ov\rangle$$

SI-START-SPEC

$$\left\langle \mathsf{ConnectionState}(c, \mathsf{CanStart}) * \mathop{\Large\ast}_{(k,h)\in m} k \mapsto h \right\rangle$$

$$\texttt{start}\ c$$

$$\left\langle (). \mathsf{ConnectionState}(c, \mathsf{Active}(m)) * \mathop{\Large\ast}_{(k,h)\in m} \Big(k \mapsto h * k \mapsto_c last(h) * \mathsf{KeyUpdStatus}(c, k, \mathsf{False})\Big) \right\rangle$$

SI-COMMIT-SPEC

$$\left\langle \mathsf{ConnectionState}(c, \mathsf{Active}(ms)) * \mathrm{dom}\ m = \mathrm{dom}\ ms = \mathrm{dom}\ mc * \mathop{\Large\ast}_{(k,h)\in m} k \mapsto h\ *\ \mathop{\Large\ast}_{(k,(ov,b))\in mc} \big(k \mapsto_c ov * \mathsf{KeyUpdStatus}(c, k, b)\big) \right\rangle$$

$$\texttt{commit}\ c$$

$$\left\langle \begin{array}{l} v.\ \mathsf{ConnectionState}(c, \mathsf{CanStart}) * \\ \Big(v = \mathsf{True} * \mathsf{CanCommit}(m, ms, mc) * \mathop{\Large\ast}_{\substack{(k,h)\in m \\ (k,p)\in mc}} k \mapsto update\_hist(p, h)\Big) \vee \\ \Big(v = \mathsf{False} * \neg\mathsf{CanCommit}(m, ms, mc)\ *\ \mathop{\Large\ast}_{(k,h)\in m} k \mapsto h\Big) \end{array} \right\rangle$$

Figure 3.7: Per-operation specification for snapshot isolation.

Start The start operation is specified using an atomic Hoare triple. This is because it takes as precondition the global points-to resource. As potentially many concurrent transactions will be starting and stopping, the database state, in the form of the global points-to resources, will typically reside in an invariant, shared among all clients of the database. The global points-to resources are described by a map m, which is the current snapshot of the database. The start operation takes the resource $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ and changes its state to active for the current snapshot m. In the postcondition, global points-to resources are returned unchanged and can be put back into a shared invariant. Further, local points-to resources are created for the client to use, based on the snapshot m, together with KeyUpdStatus resources that reflects no writes have been done yet.

Write and read The specifications for the write and read operations are almost identical to specifications for load and store operations in sequential memory; changes are immediately propagated and observed in the local points-to resources. Apart from updating the local points-to resources, the write specification also marks, using the KeyUpdStatus resource, that the current transaction has written to the key k. We use this information in the commit specification to detect write conflicts with concurrent transactions and decide whether the current transaction is allowed to commit.

Commit A client calls the commit operation with the resource $\mathsf{ConnectionState}(c, \mathsf{Active}(ms))$ stating that the current transaction has been reading from the starting snapshot ms. The precondition also takes the global points-to resources, recorded by the current snapshot of the database m, and the changes of the client, in the form of local points-to and KeyUpdStatus resources, described by the map mc. No matter the outcome of the commit operation, the postcondition includes $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ to reflect that the client has ended its transaction, and it can start a new one. The rest of the postcondition is a disjunction representing the successful and the unsuccessful commit case. To capture write conflicts with concurrent transactions, we use the CanCommit predicate in (3.1).

$$\begin{aligned}\mathsf{CanCommit}(m, ms, mc) \triangleq \forall k \in \mathsf{Keys},\ & mc[k] = \mathsf{Some}\ (p, \mathsf{True}) \\ & \Rightarrow m[k] = ms[k].\end{aligned} \tag{3.1}$$

The predicate is true if for each key the current transaction has written to (the keys where $mc[k] = \mathsf{Some}\ (p, \mathsf{True})$ holds as True comes from the KeyUpdStatus resource which is updated in the write specification) the transaction's starting snapshot ms is equal to the current snapshot of the database m. The database state is tracked with the global points-to resources using histories, *i.e.,* list of values representing all past writes, to be able to check for changes between snapshots of the database. Using values instead of histories is not sound when comparing different snapshots where the latest value is the same. In the successful commit case of the postcondition, the global points-to resources are updated with the writes made by the committing transaction by appending to the current histories (this is what *update_hist* does). The global points-to resources are left unchanged in the unsuccessful commit case.

3.5 Connecting Transactional Models to Semantic Models of Program Execution

In this section, we explain our method for extracting transactions from database implementations satisfying the Hoare triple specifications of Alnor Mathiasen et al. [2025]. Ultimately, we wish to prove that a database implementation for an isolation level I correctly implements that isolation level as per the

definition of implementation correctness from Chapter 3.3.5. As the definition of implementation correctness assumes a set of transactions as defined in the state-based model (Chapter 3.3.1), the transactions we extract must conform to this definition, implying that transactions must read their own writes. The extraction, from code to transactions usable in the state-based model, is depicted in Figure 3.1. Before going into detail with each level of the extraction in the coming subsections, we give a more detailed overview of the extraction than we did in Chapter 3.2.

The extraction works across four different levels, with the first level (level 0) being the executable code and the last level (level 3) being an extracted set of transactions $\mathcal{T}$ for which implementation correctness must hold, namely there must exist an execution at the model level that satisfies a given commit test. The core idea here is that we *instrument* the database code to produce a trace of *events* as the database runs. The instrumentation happens at level 0. (We emphasize that this instrumentation bears no effect on the execution of the program; it is *ghost code* [Owicki and Gries, 1976] which one can erase before compiling and running the code.) We instrument each operation so that it produces a so-called *pre event* and a *post event*, marking the start and end of the operation — these are tagged by a globally unique identifier for the operation in question. The trace of pre and post events constitute level 1; events emitted from any operation all go into the same trace. Throughout the extraction, for each operation we also produce its linearization point event (also tagged by the operation's unique identifier). Unlike the pre and post events, the linearization point events are not produced by code instrumentation because they are only determined at the logical level, taking into account all other concurrently running operations from other transactions. Thus, linearization point events are captured as a separate sequence of events which need to be suitably coherent with the trace of pre and post events (this will be made precise in the coming sections). Intuitively, the order of the linearization points is the order in which operations are observed by the database. The trace of linearization points makes up level 2 in the extraction. From the trace of linearization points, we have enough information to extract a set of transactions which makes up level 3. A single linearization point corresponds to a single operation, and as the trace holds the order in which the points are captured, we have enough information to soundly extract transactions. Once the set of transactions have been extracted, we have all the information we need to reason in the state-based model; implementation correctness Chapter 3.3.5 says that for any set of transactions a database produces, there must exist an execution for which the commit test of the isolation level the database implements must hold — our extraction method exactly captures the transactions produced by the database.

For the rest of this paper, we assume that we are given a database implementation (for each isolation level) that satisfies the specifications given by Alnor Mathiasen et al. [2025]. Note that we make no further assumptions on

the implementation, nor on how the specifications have been proven; indeed, we show that *any* implementation that satisfies those specifications correctly implements the relevant isolation level in the sense of Chapter 3.3.5.

Next, we explain each level in detail, and use the *write skew* example from Figure 3.3 as our running example.

3.5.1 Executable Code Instrumentation (Levels 0 and 1)

At level 0, we only instrument the client facing interface of the database implementation to emit pre and post events. Note that Figure 3.1 only shows the instrumentation for the read operation; all the operations of the interface are instrumented similarly as we display in Figure 3.8. In Figure 3.9, we show

$$
\begin{aligned}
\mathbf{Wrap}(\texttt{start}\ c) &\triangleq \mathbf{let}\ \tau = \texttt{fresh}\ (c, \mathbf{S})\ \mathbf{in}\ \texttt{start}\ c;\ \texttt{emit}\ (\tau, c, \mathbf{S}); () \\
\mathbf{Wrap}(\texttt{read}\ c\ k) &\triangleq \mathbf{let}\ \tau = \texttt{fresh}\ (c, \mathbf{R}, k)\ \mathbf{in}\ \mathbf{let}\ ov = \texttt{read}\ c\ k\ \mathbf{in} \\
&\quad\ \texttt{emit}\ (\tau, c, \mathbf{R}, k, ov); ov \\
\mathbf{Wrap}(\texttt{write}\ c\ k\ v) &\triangleq \mathbf{let}\ \tau = \texttt{fresh}\ (c, \mathbf{W}, k, v)\ \mathbf{in}\ \texttt{write}\ c\ k\ v; \\
&\quad\ \texttt{emit}\ (\tau, c, \mathbf{W}, k); () \\
\mathbf{Wrap}(\texttt{commit}\ c) &\triangleq \mathbf{let}\ \tau = \texttt{fresh}\ (c, \mathbf{C})\ \mathbf{in}\ \mathbf{let}\ b = \texttt{commit}\ c\ \mathbf{in} \\
&\quad\ \texttt{emit}\ (\tau, c, \mathbf{C}, b); b
\end{aligned}
$$

Figure 3.8: Instrumentation of database operations

one possible trace of pre and post emit events that can come from the *write skew* example from Figure 3.3. In this particular trace (notice there are several possible traces that the *write skew* example could produce), we display only the events for the read and write operations. To explain the structure of the events, let us consider the first event $(\tau_1, c_1, \mathbf{R}, x)$. It consists of the tag τ_1, the client connection c_1, a label $\mathbf{R}$, stating that the event is from a read operation, and the key being read x. Thus, this is the pre event for the read operation of transaction T_1 that the instrumentation has emitted. The next event in the trace is the pre event for the read operation of transaction T_2. It is not until the third slot of the trace that the post emit event $(\tau_1, c_1, \mathbf{R}, x_0)$ of the read operation from T_1 has been emitted. The post event $(\tau_1, c_1, \mathbf{R}, x_0)$ is similar to the corresponding pre event $(\tau_1, c_1, \mathbf{R}, x)$: the tag and connection are the same, tying the two events together, and they are marked as coming from a read operation using $\mathbf{R}$, but this time the value being read x_0 is returned. *i.e.,* the result of reading the key x is the value x_0 (the initial version of x). The interleaving of events show that read operations are concurrent. The later half of the trace display the pre and post emit events of the write operations from the *write skew* example. They too are interleaved and follow the instrumentation structure from Figure 3.8.

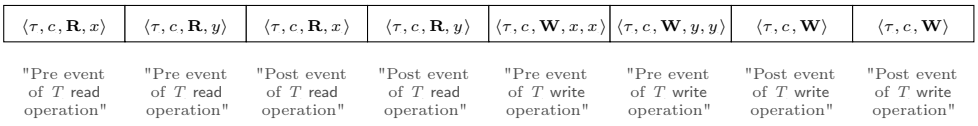


Figure 3.9: Extraction of the *write skew* example at level 1 (events from the start and commit operations are omitted).

In terms of the operational semantics, we have integrated the code instrumentation primitives **fresh** and **emit** of Birkedal et al. [2021] into the operational semantics of AnerisLang, the programming language of the Aneris program logic (an instantiation of Iris) [Krogh-Jespersen et al., 2020] used by Alnor Mathiasen et al. [2025] in their formalization. An excerpt of the operational semantics of AnerisLang (with the added code instrumentation primitives) is given in Figure 3.10. See Birkedal et al. [2021] and the Aneris documentation [Aneris-Team, 2023] for details.[3] Note that in the rule Emit-

$\boxed{(e, \sigma, t) \rightsquigarrow (e, \sigma, t)}$

$$\textsc{Pair-Proj-Step1}\quad (\mathbf{fst}\ (v_1, v_2), \sigma, t) \rightsquigarrow_{(n)} (v_1, \sigma, t)$$

$$\textsc{Function-Call-Step}\quad ((\mathbf{rec}\ f(x) = e)\ v, \sigma, t) \rightsquigarrow_{(n)} (e[(\mathbf{rec}\ f(x) = e)/f][v/x], \sigma, t)$$

$$\textsc{Load-From-Heap-Step}\quad \frac{\sigma.heap_n(\ell) = v}{(!\ell, \sigma, t) \rightsquigarrow_{(n)} (v, \sigma, t)} \qquad \textsc{Emit-Step}\quad (\mathbf{emit}\ v, \sigma, t) \rightsquigarrow_{(n)} ((), \sigma, t \cdot v)$$

$$\textsc{Fresh-Step}\quad \frac{\tau \text{ is fresh in trace } t}{(\mathbf{fresh}\ v, \sigma, t) \rightsquigarrow_{(n)} (\tau, \sigma, t \cdot \langle \tau, v \rangle)}$$

Figure 3.10: An excerpt of Aneris's operational semantics; σ tracks the state of heaps of all nodes, and the global state of the network; t is the global trace of all events; n is the name of the node in the network (ip address).

Step, the value v is added to the global trace t. Likewise, the fresh expression also appends its argument value v to the global trace, but it does so together with a freshly generated tag τ — see Figure 3.1 for an example.

The core idea of the methodology of Birkedal et al. [2021] is to enforce a so-called trace invariant, written $\mathsf{TraceInv}(\mathcal{L})$, which guarantees that the global trace of events must belong to the language $\mathcal{L}$ (a set of traces). In the

[3] Our presentation in Figure 3.10 is slightly simplified compared to the Aneris documentation [Aneris-Team, 2023] where the step relation $\rightsquigarrow$ is split into several layers and only the network layer is explicitly marked with the node name.

terminology of the Iris framework, $\mathsf{TraceInv}(\mathcal{L})$ is an example of an *invariant*, a property that all verified programs, regardless of their pre- and postconditions must uphold at all times. This is reflected in the proof rules HT-EMIT and HT-FRESH for the ghost operations **emit** and **fresh** in Figure 3.11. Here, $\mathsf{Trace}(t)$ is a predicate stating that the global trace is currently t; taken together with $\mathsf{TraceInv}(\mathcal{L})$, one concludes that $t \in \mathcal{L}$. Note how these rules ensure that the trace invariant is preserved by the execution steps of the code instrumentation. It is essentially our definition of $\mathcal{L}$ that makes up the extraction of Figure

$$\textsc{Ht-emit}\quad \frac{t \cdot v \in \mathcal{L}}{\mathsf{TraceInv}(\mathcal{L}) \vdash \{\mathsf{Trace}(t)\}\ \mathbf{emit}\ v\ \{\mathsf{Trace}(t \cdot v)\}}$$

$$\textsc{Ht-fresh}\quad \frac{\forall \tau,\ \tau \text{ is fresh in } t \Rightarrow t \cdot \langle \tau, v\rangle \in \mathcal{L}}{\mathsf{TraceInv}(\mathcal{L}) \vdash \{\mathsf{Trace}(t)\}\ \mathbf{fresh}\ v\ \{\tau.\, \tau \notin \mathsf{tags}(t) \wedge \mathsf{Trace}(t \cdot \langle \tau, v\rangle)\}}$$

Figure 3.11: Hoare-triple specifications for the **fresh** and **emit** code instrumentation operations.

3.1. We choose $\mathcal{L} \triangleq \{t \mid \mathsf{ValidTrace}(I, t)\}$ where the predicate $\mathsf{ValidTrace}(I, t)$, which we refer to as the global proof invariant, is defined in Figure The Global Proof Invariant. The argument I is the isolation level that the database is implementing, and t is the trace of pre and post events created by the instrumentation. The global proof invariant asserts the existence of a sequence of linearization points $\bar{t}$ (level 2), a set of transactions $\mathcal{T}$ (level 3) and an execution E. On the first line it states validity of $\bar{t}$ and $\mathcal{T}$ (we will explain these predicates below), before asserting implementation correctness, as in Chapter 3.3.5, in the first half of the second line. The last three predicates makes sure there is coherence between the different levels (we will explain these predicates below), *i.e.,* that a given level is consistent with its previous level.

$$\begin{aligned}\mathsf{ValidTrace}(I, t) \triangleq \exists\ \bar{t}\ \mathcal{T}\ E,\ & \mathsf{ValidSeq}(\bar{t}) \wedge \qquad \text{(The Global Proof Invariant)}\\ & \mathsf{ValidTrans}(\mathcal{T}) \wedge\\ & (\forall T \in \mathsf{comTrans}(\mathcal{T}).\ \mathsf{CT}_I(T, E)) \wedge \mathsf{CoherenceTrace}(t, \bar{t}) \wedge\\ & \mathsf{CoherenceTrans}(\bar{t}, \mathcal{T}) \wedge \mathsf{CoherenceExec}(\mathsf{comTrans}(\mathcal{T}), E)\end{aligned}$$

We remark that because our extraction method is expressed as an invariant, we capture *all* the possible transactions a database implementation produces under *any* client interaction, and show that these are sound transactions.

3.5.2 Linearization Points (Level 2)

At level 2, we have a trace of linearization points. A single linearization point must correspond to a pair of pre and post emit events from the global trace at level 1, as captured by the predicate $\mathsf{CoherenceTrace}(t, \bar{t})$, which asserts that:

1. The order of t is respected in $\bar{t}$: for any lp_1 that comes before a lp_2 in $\bar{t}$, the post event of lp_2 does not come before the pre event of lp_1 in t.

2. Every pair of pre and post events in t has a corresponding linearization point (agreeing on the tag, arguments, and return value of the operation).

3. Every linearization point in $\bar{t}$ must have a pre event in t.

4. Linearization points in $\bar{t}$ are unique.

Note that the predicate $\mathsf{CoherenceTrace}(t, \bar{t})$ essentially captures the standard linearizability correctness criteria for concurrent objects [Herlihy and Wing, 1990]; there is a single point, between the invocation and end of an operation, where the effect of the operation appears instantaneously. On the logical level, linearization points in separation logic are exposed through atomic Hoare triples. Recall the per-operation specifications from Chapter 3.4.2. They were specified using atomic Hoare triples (such that clients can use invariants to reason about concurrent threads, and invariants can be opened around the database operations). As the extraction we are defining in this section is a trace invariant, *i.e.,* an invariant in Iris, we can use the fact that the operations are specified with atomic Hoare triples to insert linearization points in the sequence at level 2, in between the emit of the pre event and the post event in the instrumented code. The insertion happens as part of the proof that from verified database implementations we extract valid transactions w.r.t. the state-based model, we will see this in detail in Chapter 3.6. Because linearization points are unique, and we define them to hold all information about the operations they correspond to, we can essentially treat them as operations. The predicate $\mathsf{ValidSeq}(\bar{t})$ asserts that operations (their linearization points) must appear in the expected order, *i.e.,* start must be called before read and writes can be performed, commit can only appear after start (and possibly some reads and/or writes after start), etc.

In Figure 3.12, we show one of the possible extractions of linearization points for the trace of pre and post events from the *write skew* example in Figure 3.9. Notice they way we have defined linearization points: they correspond exactly to the definition of transaction operations we defined in the state-based model Chapter 3.3. This sequence of linearization points is a sound extraction, as it satisfies the conditions of the ValidSeq predicate. Noticeable, each pair of pre and post events have one linearization point, and the linearization points respect the order imposed by the trace pre and post events on the previous level; as the read operation from the two transactions of the example are concurrent (their pre and post events overlap), their linearization points can be ordered either way, similarly for the write operations. But, it would be an unsound order of linearization points if one of the two read operations were after a linearization point from one of the two write operations, because the post event

of any read operation comes before the pre event of any write operation. Thus, we know for sure that the read operations ended before the write operations started.

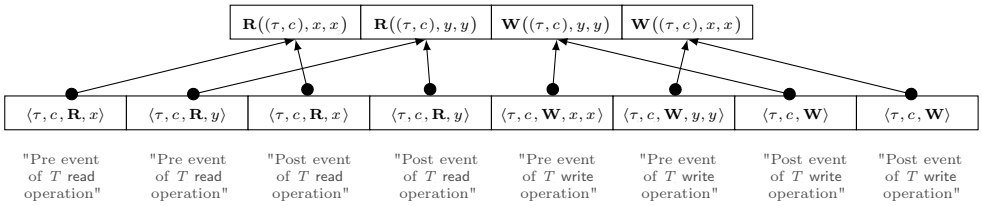


Figure 3.12: Extraction of the *write skew* example from level 1 to level 2 (start and commit operations are omitted).

3.5.3 Set of Transactions Adhering to the State-based Model (Level 3)

At level 3, we "chop" the sequence of operations (their linearization points) into sets of transactions. The predicate $\mathsf{CoherenceTrans}(\bar{t},\mathcal{T})$ in
(The Global Proof Invariant) asserts coherence by stating that transactions in $\mathcal{T}$ have the same operations as the trace $\bar{t}$, any non commit operation must be in the same transaction as the commit operation that follows in $\bar{t}$ from the same connection, and that for any two operations from a transaction T in $\mathcal{T}$ the program order of T must agree with the ordering of the two operations in $\bar{t}$. The predicate $\mathsf{ValidTrans}(\mathcal{T})$ ensures that the set of transactions $\mathcal{T}$ we extract meets the conditions expected by the state-based model:

1. There is at most one open transaction per connection. An open transaction is a transaction which is started but not yet committed.
2. No two transactions contain the same operation. (This includes the operation identifier/tag.)
3. For each transaction we have:
   - All operations come from the same connection.
   - A read of key k reads the most recent write to k (if any) in the same transaction.
   - There is at most one commit and it is the last operation of the transaction.

In Figure 3.13, we show the extraction of transactions from the sequence of linearization points in the *write skew* example. In contrast to the previous figure, we have included start and commit linearization points. Intuitively, for each connection, we create a transaction by going through the sequence of linearization points and add operations of the connection as they appear in

the sequence. Notice how start operations are in the sequence of linearization points, but we do not use them when creating transactions. This reflects that start operations are used in implementations, but they are not included in the state-based model.

Because the extraction is an invariant, we sometimes have partially completed transactions in the set of transactions. For partially completed transactions, we assert they read their most recent write, and are otherwise well-formed as defined above, but we only impose that they should be included in the reasoning at the state-based model level, to show implementation correctness, when they are complete.

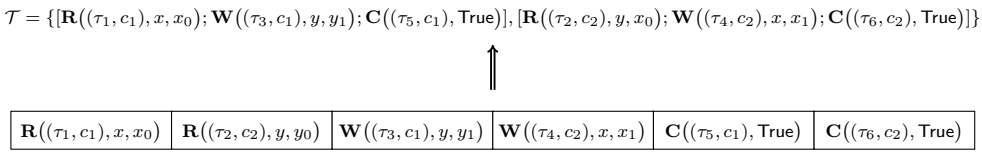


Figure 3.13: Extraction of the *write skew* example from level 2 to level 3.

3.5.4 Implementation Correctness in the State-based Model

The rest of (The Global Proof Invariant) asserts that there exists an execution E that is coherent with the set of committed transactions ($\mathsf{comTrans}(\mathcal{T})$ is the subset of committed transactions in $\mathcal{T}$), *i.e.,* $\mathsf{CoherenceExec}(\mathsf{comTrans}(\mathcal{T}), E)$ as defined in Chapter 3.3, and for which implementation correctness holds, *i.e.,* $(\forall T \in \mathsf{comTrans}(\mathcal{T}).\ \mathsf{CT}_I(T, E))$.

For the transactions extracted in the *write skew* example Figure 3.13, we create a valid execution in Figure 3.14; the execution is created by applying transaction T_1 followed by T_2, thus $\mathsf{CoherenceExec}(\mathsf{comTrans}(\mathcal{T}), E)$ holds. The execution is also valid with regard to the commit test for snapshot isolation, as this is one of the two executions for the *write skew* example we presented in Figure 3.4. Creating the other execution, by applying T_2 followed by T_1 instead, is also a valid choice, as implementation correctness merely asserts the existence of some valid execution.

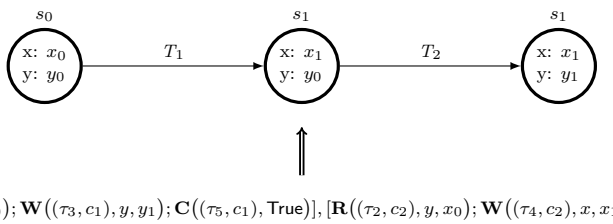


Figure 3.14: Extraction of the *write skew* example from level 3 to the model level.

Now that we have defined the extraction method, we can formally state our main theorem that gives us implementation correctness for a database implementation verified with the specifications of Alnor Mathiasen et al. [2025] without making any additional assumption. As our method for extracting transactions from code is stated as an invariant on traces, we get that any client interaction with a database verified using our method, will at any point in its execution only have produced valid transactions according to the state-based model. This is exactly what our main theorem, Theorem 3.5.1 states.

Theorem 3.5.1 (Adequacy)**.** *Let I be an isolation level and let $spec_I$ be the specifications ascribed by Alnor Mathiasen et al. [2025] for isolation level I. Let $e_1 \dots e_n$ be n programs,* i.e., *the main thread of n nodes running clients of the database, such that for each e_i we have that the quantified Hoare triple $\forall lib,\ \{spec_I(lib)\}\, e_i \oplus lib\, \{\mathsf{True}\}$ holds where $\oplus$ links the transactional library with its client. Finally, let lib_I be some database implementation for which $spec_I(lib_I)$ holds. Then, for any given initial state (of heaps and global network) σ_0 and any point in program execution we reach,* i.e., *for any list of expressions $e'_{1,1} \dots e'_{1,m_1}, e'_{2,1} \dots e'_{2,m_2} \dots e'_{n,1} \dots e'_{n,m_n}$ (one for each thread in each node), any state σ, and any trace t such that*

$$([e_1 \oplus \mathbf{Wrap}(lib_I)] \dots [e_n \oplus \mathbf{Wrap}(lib_I)], \sigma_0, \varepsilon) \longrightarrow^* ([e'_{1,1} \dots e'_{1,m}], [e'_{2,1} \dots e'_{2,m}] \dots [e'_{n,1} \dots e'_{n,m}], \sigma, t)$$

we have $\mathsf{ValidTrace}(I, t)$ holds, and furthermore, each $e'_{i,j}$ is not a crashed (runtime error) state.

The main theorem is a general result for any weak isolation level I with an implementation lib_I which satisfies the specifications $spec_I(lib_I)$ ascribed by Alnor Mathiasen et al. [2025] that we presented for snapshot isolation in Chapter 3.4. The theorem states that any verified client of the database running on n separate nodes (each starting with only a single thread e_i) when run for arbitrary number of steps ($\longrightarrow$ is the relation $\rightsquigarrow_{(\cdot)}$ lifted to the level of the entire network) reaches a configuration in which none of the threads is in a crash state (guaranteeing no runtime error), and furthermore, the produced trace t is a valid trace, $\mathsf{ValidTrace}(I, t)$, as per (The Global Proof Invariant). The latter ensures implementation correctness as in Chapter 3.3.5. The crucial assumption in this theorem is that clients are verified against an arbitrary implementation that satisfies the expected specifications. This generality allows us to show that a client also can be linked with the wrapped library ($\mathbf{Wrap}(lib_I)$), which includes the code instrumentation of Figure 3.8 to produce pre and post events, as long as the wrapped library also satisfies the expected specification. Recall that the code instrumentation bears no effect on programs' execution and can be erased before compilation. The crux of the proof of Theorem 3.5.1, which we will discuss in Chapter 3.6, is to show two things, one that $\forall lib, \{spec_I(lib)\}\, e_i \oplus lib\, \{\mathsf{True}\}$ implies $\forall lib, \{spec_I(\mathbf{Wrap}(lib))\}\, e_i \oplus \mathbf{Wrap}(lib)\, \{\mathsf{True}\}$, and that $spec_I(lib)$ implies

$spec_I(\mathbf{Wrap}(lib))$, for any database implementation lib. While the former is crucial for the proof of Theorem 3.5.1, it is trivial to prove. However, the latter is the main challenge and forms the bulk of the proof of Theorem 3.5.1. In particular, proving this requires maintaining (The Global Proof Invariant), $\mathsf{ValidTrace}(I, t)$ throughout the proof. Intuitively, $spec_I(lib)$ implies $spec_I(\mathbf{Wrap}(lib))$ because the specification $spec_I(\mathbf{Wrap}(lib))$ changes the internal definition of the resources used in $spec_I$, without making any external modifications, *i.e.,* "the client interface of the specification" is preserved. This can be done as all the resources in the per-operation specifications of $spec_I$ are existentially quantified as we emphasized in Chapter 3.4.2; all a client knows about the resources is that they work with the specifications. The wrapping of a specification thus means creating a new instantiation of the existentially quantified resources used in the specifications, and showing that all the Hoare triples still hold with the new resources. Having a proof of $\{spec_I(\mathbf{Wrap}(lib))\}\, e_i \oplus \mathbf{Wrap}(lib)\, \{\mathsf{True}\}$ gives us that all the clients can safely execute with code instrumentation while the invariant $\mathsf{ValidTrace}(I, t)$ is upheld — otherwise a client would get stuck according to the operational semantic and safety would be violated. As $\mathsf{ValidTrace}(I, t)$ encapsulates our extraction method presented throughout this section, implementation correctness follows and gives us the result of the main theorem. The construction of the new resources used in $spec_I(\mathbf{Wrap}(lib))$ is made as a part of the proof of Theorem 3.5.1; we show the proof structure of Theorem 3.5.1 in Chapter 3.6 and a proof sketch with more details for snapshot isolation in Chapter 3.7.

3.6 Proof Structure: Transactional Library Implements Transactional Model

In this section, we discuss the structure of the proof of Theorem 3.5.1. In Chapter 3.5, when we presented the extraction method leading to Theorem 3.5.1 we used a particular execution of the *write skew* example to explain how the extraction works. However, to prove Theorem 3.5.1 we must consider all executions of all possible clients. This can formally be seen in our Theorem 3.6.2 which states that if $spec_I(lib)$ holds then so does $spec_I(\mathbf{Wrap}(lib))$. This intuitively captures that all executions of clients of $\mathbf{Wrap}(lib)$ satisfy (The Global Proof Invariant) as long as those clients are verified against $spec_I(lib)$.

As we explained in Chapter 3.5, the crux of the proof of Theorem 3.5.1 consists of the following Lemma 3.6.1 and Theorem 3.6.2:

Lemma 3.6.1. *Let I be an isolation level, and let $e_1 \dots e_n$ be n programs, then we have the following:*

$$(\forall lib,\ \{spec_I(lib)\}\, e_i \oplus lib\, \{\mathsf{True}\}) \rightarrow$$
$$\forall lib,\ \{spec_I(\mathbf{Wrap}(lib))\}\, e_i \oplus \mathbf{Wrap}(lib)\, \{\mathsf{True}\}.$$

Proof. This statement is a tautology of higher-order separation logic. Thus, the proof is rather straightforward and does not depend on specifics of $spec_I$, any e_i, or the wrap function. We simply need to introduce the antecedent and the universal quantification over *lib*, and instantiate the antecedent with $\mathbf{Wrap}(lib)$. □

Theorem 3.6.2 (Main Theorem)**.** *Let I be an isolation level and $\mathcal{L}_I \triangleq \{t \mid \mathsf{ValidTrace}(I, t)\}$, then*

$$\forall lib.\ spec_I(lib) \twoheadrightarrow \mathsf{Trace}(\varepsilon) \twoheadrightarrow \mathsf{TraceInv}(\mathcal{L}_I) \twoheadrightarrow spec_I(\mathbf{Wrap}(lib)).$$

With Lemma 3.6.1 and Theorem 3.6.2 formally stated, we first present a sketch of the proof of Theorem 3.5.1 before proceeding to discuss the proof of Theorem 3.6.2.

A sketch of the proof of Theorem 3.5.1. In Theorem 3.5.1, we get to assume that the Hoare triples $\forall lib, \{spec_I(lib)\}\, e_i \oplus lib\, \{\mathsf{True}\}$ hold for all client programs $e_1 \dots e_n$. This, by Lemma 3.6.1 allows us to conclude $\forall lib, \{spec_I(\mathbf{Wrap}(lib))\}\, e_i \oplus \mathbf{Wrap}(lib)\, \{\mathsf{True}\}$ for $e_1 \dots e_n$. Now, to finish the proof of Theorem 3.5.1, we need to show the preconditions of these Hoare triples hold. This is because the adequacy theorem of Birkedal et al. [2021], roughly speaking, states that if we prove Hoare triples for our programs, and we prove their preconditions, then these programs are safe to run, *i.e.,* , they do not crash, and that all invariants hold throughout the execution. Furthermore, the adequacy of Birkedal et al. [2021] allows us to pick any $\mathcal{L}$, and only requires the Hoare triples to be proven under the assumption that we have $\mathsf{Trace}(\varepsilon)$ and $\mathsf{TraceInv}(\mathcal{L})$. Thus, what remains is to show the preconditions, *i.e.,* , $spec_I(\mathbf{Wrap}(lib))$, but this follows from Theorem 3.6.2 because Alnor Mathiasen et al. [2025] have proven $spec_I(lib)$ — our Rocq mechanization builds on that of Alnor Mathiasen et al. [2025]. □

The proof of Theorem 3.6.2 is what we discuss in the rest of this section and the next. The proof of $spec_I(\mathbf{Wrap}(lib))$ in Theorem 3.6.2 follows the structure imposed of $spec_I$ that we presented in Chapter 3.4.1: we need to show the existence of a number of predicates and that all operations satisfy their specifications for these predicates. As the operations of $\mathbf{Wrap}(lib)$ are the wrapped versions of the operations in *lib* using the code instrumentation displayed in Figure 3.8, a part of proving the Hoare triples for these operations is showing that the events emitted into the global trace t, tracked using the resource $\mathsf{Trace}(t)$, satisfy the trace invariant $\mathsf{TraceInv}(\mathcal{L}_I)$, *i.e.,* , $t \in \mathcal{L}_I$, at all times. Remember that the trace invariant encapsulates the extraction of Chapter 3.5 that we defined with the predicate $\mathsf{ValidTrace}(I, t)$ as $\mathcal{L}_I$.

While the proof of Theorem 3.6.2 varies for each isolation level for a number of reasons, such as the difference in per-operation specifications and the commit test, one of the main difficulties all isolation levels deal with is the insertion of linearization points. Specifically, insertions into the sequence at level 2 in

between the code instrumentation emitting the pre and post events at level 1. To this end, we set up a so-called *ghost theory*. A ghost theory consists of a number of propositions defined in terms of ownership of separation logic resources and a number *laws* that govern them; see [Jung et al., 2018] for details. The ghost theory we introduce for handling insertion of linearization points introduces predicates $\mathsf{Reflect}_I(t, \bar{t})$, $\mathsf{PermissionLin}(\tau)$, and $\mathsf{PermissionPost}(\tau)$. The laws of our ghost theory are displayed in Figure 3.15. The predicate $\mathsf{Reflect}_I(t, \bar{t})$ reflects the trace of pre and post events t and sequence of linearization points $\bar{t}$ to the level of the logic using separation logic resources. The definition of $\mathsf{Reflect}_I(t, \bar{t})$ varies for each isolation level, but the laws in Figure 3.15 are common laws that hold for all isolation levels. The details of how $\mathsf{Reflect}_I(t, \bar{t})$ is defined are not important here. What matters is that it allows us to reason about traces, linearization points, sets of transactions and executions in our logic — the sets of transactions and executions are existentially quantified inside $\mathsf{Reflect}_I(t, \bar{t})$. When proving the per-operation specification of $\mathit{spec}_I(\mathbf{Wrap}(\mathit{lib}))$ in Theorem 3.6.2, we take the $\mathsf{Trace}(t)$ resource, tracking the global proof trace, and create an internal proof invariant

$$\boxed{\exists t, \bar{t}.\ \mathsf{Trace}(t) * \mathsf{Reflect}_I(t, \bar{t})}$$

together with $\mathsf{Reflect}_I(t, \bar{t})$. This invariant is available to us when we reason about emitting events into the global trace that the instrumented operations do. When the global trace changes, we thus also have to update $\mathsf{Reflect}_I(t, \bar{t})$ using the laws of our ghost theory. Now, the point of having $\mathsf{Reflect}_I(t, \bar{t})$ in an invariant together with $\mathsf{Trace}(t)$ is that $\mathsf{Reflect}_I(t, \bar{t})$ implies $\mathsf{ValidTrace}(I, t)$ (by Law 1) which allow us to show that t is in $\mathcal{L}_I$ at all times. When the instrumented database operations insert a pre event or a post event into the global trace, we need to update $\mathsf{Reflect}_I(t, \bar{t})$ accordingly. These updates can be made using Law 2 and Law 3, respectively. As the code instrumentation, *i.e.,* the **`emit`** and **`fresh`** expressions, are atomic operations in Iris, we can open the internal proof invariant around them — an atomic operation is one that steps to a value in a single step of the operational semantics; see Figure 3.10. Thus, we can update $\mathsf{Trace}(t)$ with the code instrumentation specifications in Figure 3.11, and update $\mathsf{Reflect}_I(t, \bar{t})$ using the ghost theory laws. When inserting the pre event with Law 2, we gain a resource $\mathsf{PermissionLin}(\tau)$. This is an exclusive permission to update the sequence of linearization points at level 2 by inserting a linearization point with tag τ. Having $\mathsf{PermissionLin}(\tau)$, using Law 4, we can ensure that no other linearization point has been inserted with tag τ. The insertion of linearization points into the sequence $\bar{t}$ at level 2 of the extraction requires operation and isolation level specific reasoning to update $\mathsf{Reflect}_I(t, \bar{t})$. The reason for this is that the insertion of the linearization point of operations enforces changes in the set of transactions and execution that $\mathsf{Reflect}_I(t, \bar{t})$ tracks, and which it uses to establish $\mathsf{ValidTrace}(I, t)$ in Law 1. In Chapter 3.7, we show how $\mathsf{Reflect}_I(t, \bar{t})$ can be updated with a linearization point for the `read` operation in snapshot isolation. This involves the creation of new resources

LAW 1
$\mathsf{Reflect}_I(t, \bar{t}) \vdash \mathsf{Reflect}_I(t, \bar{t}) * \mathsf{ValidTrace}(I, t)$

LAW 2
$$\frac{\tau \notin \mathsf{tags}(t) \qquad \mathsf{PreEvent}(\langle \tau, v \rangle)}{\mathsf{Reflect}_I(t, \bar{t}) \vdash \mathsf{Reflect}_I(t \cdot \langle \tau, v \rangle, \bar{t}) * \mathsf{PermissionLin}(\tau)}$$

LAW 3
$$\frac{\mathsf{PostEvent}(\langle \tau, v \rangle) \qquad \mathsf{CoherenceTrace}(t \cdot \langle \tau, v \rangle, \bar{t})}{\mathsf{Reflect}_I(t, \bar{t}) * \mathsf{PermissionPost}(\tau) \vdash \mathsf{Reflect}_I(t \cdot \langle \tau, v \rangle, \bar{t})}$$

LAW 4
$\mathsf{Reflect}_I(t, \bar{t}) * \mathsf{PermissionLin}(\tau) \vdash \mathsf{Reflect}_I(t, \bar{t}) * \mathsf{PermissionLin}(\tau) * \tau \notin \mathsf{tags}(\bar{t})$

LAW 5
$\mathsf{Reflect}_I(t, \bar{t}) * \mathsf{PermissionPost}(\tau)$
$\vdash \mathsf{Reflect}_I(t, \bar{t}) * \mathsf{PermissionPost}(\tau) * \neg(\exists event.\ \mathsf{tag}(event) \in \mathsf{tags}(t))$

LAW 6
$\mathsf{Reflect}_I(t, \bar{t}) \vdash \mathsf{Reflect}_I(t, \bar{t}) * \mathsf{Prefix}(t, \bar{t})$

LAW 7
$\mathsf{Reflect}_I(t, \bar{t}) * \mathsf{Prefix}(t, \bar{t}) \vdash \mathsf{Reflect}_I(t, \bar{t}) * \mathsf{Prefix}(t', \bar{t}') * t \leq t' * \bar{t} \leq \bar{t}'$

Figure 3.15: Excerpt of ghost theory.

to instantiate the existentially quantified resources in $spec_I(\mathbf{Wrap}(lib))$. Recall that it is due to the read operation being specified with an atomic Hoare triple that we can open our internal proof invariant and insert a linearization point at level 2: An atomic Hoare triple specification asserts that the operation is safe to execute and exposes the linearization point of the program at which point invariants can be accessed. When inserting linearization points, we also need to establish that the pre event for the linearization point we are inserting has been emitted to the trace t. This is done by creating a Prefix resource using LAW 6 at the time when the pre event is inserted by the code instrumentation. Then, using LAW 7, when inserting the linearization point we can assert that the pre event is already in the trace of events at level 1. A similar reasoning applies when the code instrumentation inserts the post event; we need to show the existence of a corresponding linearization point in the global trace.

In the next section, we give more details of how a per-operation specification of $spec_I(\mathbf{Wrap}(lib))$ is proven for a particular operation, *i.e.,* , the read operation in snapshot isolation.

3.7 Proof Sketch: Transactional Library Implements Transactional Model

In this section, we present a proof of the read operation part of Theorem 3.6.2 for the snapshot isolation level using the ghost theory from Chapter 3.6. The proof for other operations and other isolation levels are similar in nature; see our Rocq mechanization for more details. We begin by discussing the important point that no client can distinguish between *lib* and **Wrap**(*lib*), because all the client relies on is that $spec_{SI}$ holds, and $spec_{SI}$ existentially quantifies all the resources that are used in the specification of operations. To understand this subtle point, let us look at the specification of the read operation in $spec_{SI}$ for snapshot isolation:

$$\langle k \mapsto_c ov \rangle \ \mathsf{read}\ c\ k \ \langle ov.\, k \mapsto_c ov \rangle \qquad \text{(SI-Read-Spec)}$$

Intuitively, $k \mapsto_c ov$ is a logical proposition (resource) stating that the key k has value ov in the (current) connection c. That is, either ov is the value of the last write to k in the current transaction of c, or there is no prior write to k in the current transaction of connection c, and ov is the value of k at the moment (snapshot) when the transaction started. However, a proof of a client does not in any way rely on the exact definition of the predicate $k \mapsto_c ov$; to prove a client correct we *only* need to know that (SI-Read-Spec) holds, for some abstract definition of $k \mapsto_c ov$. This is the parametric property of the specification structure we highlighted in Chapter 3.4.2, *i.e.,* the fact that the resources of the specification are existentially quantified. Thus, we can change the underlying definition of $k \mapsto_c ov$ in $spec_{SI}$ without affecting the proof of the clients that are verified against it. This is what our resource wrapping $\mathsf{Wrap}(k \mapsto_c ov)$ does: it changes the underlying definition of all the predicates involved in $k \mapsto_c ov$ without changing its "interface". That is, in $spec_{SI}(\mathbf{Wrap}(lib))$, the specification for the read operation is as follows:

$$\langle \mathsf{Wrap}(k \mapsto_c ov) \rangle\ \mathbf{Wrap}(\mathsf{read}\ c\ k)\ \langle ov.\, \mathsf{Wrap}(k \mapsto_c ov) \rangle \qquad \text{(SI-Read-Spec-Wrapped)}$$

where **Wrap**(read c k) is exactly as shown in Figure 3.1. Thus, to establish Theorem 3.6.2, we need to show that (SI-Read-Spec) implies (SI-Read-Spec-Wrapped), and similarly for the other database operations: init, start, write, and commit. In this section, we focus on the *read* operation and refer the reader to the Rocq formalization for details about the other operations. The proof obligation for the *read* operation is formally stated as follows (note that the original read specification (SI-Read-Spec) is included in the assumption $spec_{SI}(lib_{SI})$):

$$\begin{array}{l} spec_{SI}(lib_{SI}) * \mathsf{TraceInv}(\mathcal{L}_{SI}) * \qquad \text{(Read-Implies-Wrapped)} \\ \boxed{\exists t, \bar{t}.\ \mathsf{Trace}(t) * \mathsf{Reflect}_{SI}(t, \bar{t})} \\ \vdash \langle \mathsf{Wrap}(k \mapsto_c ov) \rangle\ \mathbf{Wrap}(\mathsf{read}\ c\ k)\ \langle ov.\, \mathsf{Wrap}(k \mapsto_c ov) \rangle \end{array}$$

Notice here that the internal proof invariant $\boxed{\exists t, \bar{t}.\ \mathsf{Trace}(t) * \mathsf{Reflect}_{SI}(t, \bar{t})}$ that we presented in Chapter 3.6 is included in the assumptions, as is also the trace invariant $\mathsf{TraceInv}(\mathcal{L}_{SI})$ asserting that the global trace t must remain in $\mathcal{L}_{SI}$, *i.e.,* $t \in \mathcal{L}_{SI}$ (recall $\mathcal{L}_{SI} \triangleq \{t \mid \mathsf{ValidTrace}(SI, t)\}$). To make sure the global trace stays in $\mathcal{L}_{SI}$ during the proof of (Read-Implies-Wrapped), the crucial point is to get the resource wrapping $\mathsf{Wrap}(k \mapsto_c ov)$ right. It must be strong enough to show all the properties that we list below about the set of transactions $\mathcal{T}$ and execution E. As we explained in Chapter 3.6, $\mathcal{T}$ and E are existentially quantified in $\mathsf{Reflect}_{SI}(t, \bar{t})$. The proposition $\mathsf{Reflect}_{SI}(t, \bar{t})$ ensures that the set of transactions $\mathcal{T}$ and execution E that it tracks internally are coherent with t and $\bar{t}$ such that $\mathsf{ValidTrace}(SI, t)$ can be satisfied — this is why we can prove $\mathsf{ValidTrace}(SI, t)$ in Law 1. The proposition $\mathsf{Wrap}(k \mapsto_c ov)$ must be defined in such a way as to ensure that the following properties hold for the set of transactions $\mathcal{T}$ and execution E that $\mathsf{Reflect}_{SI}(t, \bar{t})$ tracks internally:

1. There must exist a *start* linearization point with connection c in $\bar{t}$ after which there is no *commit* linearization point with connection c.

2. If the operation is not the first of the current transaction, then there is an open transaction in $\mathcal{T}$ holding the previous operations.

3. If there is a previous $\mathbf{W}(id, k, v)$ operation in the current transaction, then $ov = \mathsf{Some}\ v$, otherwise ov is the value of key k in a state s corresponding to the current transaction's snapshot in E.

To create the wrapping $\mathsf{Wrap}(k \mapsto_c ov)$, we define a proposition $\mathsf{LatestWrite}(k, ov, c)$ using separation logic resources and extend our ghost theory. This proposition works in conjunction with $\mathsf{Reflect}_{SI}(t, \bar{t})$ and lets us establish the properties listed above. With $\mathsf{LatestWrite}(k, ov, c)$, we can prove Law 8 in our ghost theory which we use when we insert the linearization point for the *read* operation. The wrapping of $\mathsf{Wrap}(k \mapsto_c ov)$ is then defined to hold $\mathsf{LatestWrite}(k, ov, c)$ together with the original non-wrapped version of the resource $k \mapsto_c ov$ from $spec_{SI}(lib_{SI})$:

$$\mathsf{Wrap}(k \mapsto_c ov) \triangleq k \mapsto_c ov * \mathsf{LatestWrite}(k, ov, c) \qquad \text{(Wrapped-Key)}$$

Now, by (Wrapped-Key) above, we can show the implication (Read-Implies-Wrapped) using our combined ghost theory of Figure 3.15 and Law 8. As highlighted in Chapter 3.6, we need to take care of the fact that the

Law 8
$\mathsf{Reflect}_{SI}(t, \bar{t}) * \mathsf{PermissionLin}(\tau) * \mathsf{LatestWrite}(k, ov, c)$
$\vdash \mathsf{Reflect}_{SI}\big(t, \bar{t} \cdot \mathbf{R}((\tau, c), k, ov)\big) * \mathsf{PermissionPost}(\tau) * \mathsf{LatestWrite}(k, ov, c)$

wrapped code contains code instrumentation which inserts pre- and post-events into the global trace. Thus, to ensure that the global trace remains within $\mathcal{L}_{SI}$, we need to insert a linearization point into the trace of linearization points (see

condition (2) of $\mathsf{CoherenceTrace}(t, \bar{t})$ on page 92). Crucially, the Hoare triple (SI-Read-Spec) is a logically atomic Hoare triple. The upshot of this is that (the specification of) the read operation can be treated atomically, *i.e.,* we can access invariants when reasoning about the read operation — recall that this access to invariants in practice takes place around the linearization point of the expression for which we have the atomic Hoare triple. Thus, before (the linearization point of) the call to $\mathsf{read}\ c\ k$ inside $\mathbf{Wrap}(\mathsf{read}\ c\ k)$, we can assume that $\exists t, \bar{t}.\ \mathsf{Trace}(t) * \mathsf{Reflect}_{SI}(t, \bar{t})$ holds, and we need to reestablish it afterwards. This allows us to insert a linearization point for the read operation into $\bar{t}$, which in turn means that when the code instrumentation inserts the post event, the trace remains coherent, *i.e.,* , $\mathsf{CoherenceTrace}(t, \bar{t})$ continues to hold. As we get to assume the wrapped resource $\mathsf{Wrap}(k \mapsto_c ov)$ around the insertion of the linearization point, we can split $\mathsf{Wrap}(k \mapsto_c ov)$ into the non-wrapped version $k \mapsto_c ov$, to satisfy the precondition of $\mathsf{read}\ c\ k$, and the resource we added $\mathsf{LatestWrite}(k, ov, c)$, to update $\mathsf{Reflect}_{SI}(t, \bar{t})$ using Law 8 of our ghost theory.

To summarize, the proof of Theorem 3.6.2, which we use to establish Theorem 3.5.1, amounts to doing what we have done in this section for the read operation for all the operations of the database interface. The complexity here lies in coming up with wrapped resource definitions that let us raise per-operation specifications to the level of abstraction of a global execution. The crux of the difficulty, stems from all the layers of our extraction method in Figure 3.1 being tightly connected through the coherence predicates of the global proof invariant. Updating one layer, using the reflection in the logic ($\mathsf{Reflect}_{SI}(t, \bar{t})$), requires reestablishing assertions about all the other layers internal to $\mathsf{Reflect}_{SI}(t, \bar{t})$. In fact, the proof of Theorem 3.6.2 consists of several thousand lines of Rocq proof code for *each* isolation level.

3.8 Related Work

Recent work on the higher-order separation logic Trillium [Timany et al., 2024a], which is based on Iris, developed a method for showing that a labeled state transition systems is refined by an executable implementation. While Trillium is language generic, implementations can be written in AnerisLang [Krogh-Jespersen et al., 2020]. In particular, they prove that AnerisLang implementations of two-phase commit and single-degree Paxos refine their respective TLA$^+$models. The refinement is then used to carry over properties to the implementations that were established at the model level in TLA$^+$. A natural question to ask is why we did not use Trillium to reason about the state-based model for database implementations, as the state-based model has been embedded in TLA$^+$[Soethout et al., 2020]. The reason is that with Trillium, we can gain that a database implementation refines the model, but we do not get a main theorem, such as Theorem 3.5.1, stating that any client interaction

will produce a valid set of transactions. The TLA$^+$model of Soethout et al. [2020] has to be model-checked against each workload one wants to verify which quickly leads to state space explosion.

Much work is done on verifying distributed systems using separation logic without making connections to other models. Also in the context of AnerisLang, Nieto et al. [2022, 2023] builds a framework for conflict-free replicated data types allowing users to define highly-available distributed applications, and Gondelman et al. [2021] creates a distributed non-transactional key-value store guaranteeing causal consistency. Using Grove [Sharma et al., 2023], a distributed separation logic with support for reasoning about node crashes built on the Iris framework, a distributed key-value store with primary backup replication, based upon a replicated state machine library, is verified [Sharma et al., 2023]. As the isolation level specifications of Alnor Mathiasen et al. [2025] have been formalized in Aneris (Aneris is the program logic for reasoning about AnerisLang), the Aneris instantiation of Iris is the choice in this paper, but all of our techniques will also work in other instantiations of Iris such as Grove.

There is a line of work on creating databases with so-called *verifiable transactions* [Xia et al., 2022; Zhao et al., 2023]. That is, the database produces at runtime a proof that it does not break guarantees such as isolation levels for transactions. This method naturally comes with runtime overhead, and it is also not guaranteed that the verification mechanisms inside the databases are implemented correctly.

3.9 Conclusion and Future Work

This paper presents the first method for formally verifying that a database correctly implements its isolation level, by lifting Hoare triples specifications to transactional consistency models using invariants on traces. We have lifted specifications of the weak isolation levels read uncommitted, read committed and snapshot isolation, thus these results serve as free theorems for all future verification efforts; by verifying that a database implements the separation logic specifications, it automatically implements its isolation level.

In the future, it will be interesting to see our approach applied to more isolation levels than we do in this paper. There exists a plethora of other levels, such as *opacity* [Guerraoui and Kapalka, 2008] and *virtual world consistency* [Imbs and Raynal, 2012], which are popularized in software transactional memory systems and restricts the behavior of ongoing transactions, or variants of snapshot isolation with intricate behavior such as *parallel* snapshot isolation [Cerone et al., 2015a; Sovran et al., 2011]. Currently, these isolation levels do not have separation logic specifications, which, of course, is a prerequisite for using our approach. We do not expect the extension to other isolation levels to impose a major challenge, but it could involve new challenges to extend the

approach to other database APIs; currently we model a key-value store, but SQL compliant systems support a much more complex API. The predominant transactional models we have used in this paper, do not fully support SQL: Crooks et al. [2017] is limited to a key-value store while Adya [1999] does support *predicate*-based operations that can be used to model features of SQL.

3.10 Appendix: Proving safety of examples

In this section, we show how to use the per-operation specifications to prove safety of the *write skew* example. In Figure 3.16, we display the code of the example together with the invariant we use. The vertical bars separate two clients program each running their transaction of the *write skew* example. Initially, when using the snapshot isolation specification, we obtain initial points-to resources for the keys x and y: $x \mapsto [\,]$ and $y \mapsto [\,]$. With these resources, we create the invariant $\boxed{\exists h_x h_y,\ x \mapsto h_x * y \mapsto h_y}$. The proof of each client transaction can be done in parallel — this is a feature of concurrent/distributed separation logics. As the invariant is a persistent resource, *i.e.,* it does not assert exclusive ownership, we can use it when proving both clients. We go through the proof of the first transaction; the proof of the other transaction is similar. Having initialized a connection using the operation init (this is not shown as part of Figure 3.16), the client holds the $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ resource as specified in SI-INIT-CLIENT-SPEC. For the precondition of the start operation in SI-START-SPEC, we need the global points to resources. Because the start operation is specified using an atomic Hoare triple, we can make use of the rule in INV-ATOMIC, and get access to the global points-to resources $x \mapsto h_x * y \mapsto h_y$ from inside the invariant. Using the global points-to resources and the $\mathsf{ConnectionState}(c, \mathsf{CanStart})$ resource, we can satisfy the precondition of the start specification. In the postcondition, we get back the unchanged global points to resources $x \mapsto h_x * y \mapsto h_y$, which we use to reestablish the invariant, and the resources $\mathsf{ConnectionState}(c, \mathsf{Active}(\{x \mapsto h_x; y \mapsto h_y\})) * x \mapsto_c last(h_x) * y \mapsto_c last(h_y) * \mathsf{KeyUpdStatus}(c, x, \mathsf{False}) * \mathsf{KeyUpdStatus}(c, y, \mathsf{False})$, which we hold onto locally for the rest of the proof. The body of the transaction, the read operation "read y " followed by the write operation "`write` x 1 ", changes our local resources to $\mathsf{ConnectionState}(c, \mathsf{Active}(\{x \mapsto h_x; y \mapsto h_y\})) * x \mapsto_c \mathsf{Some}\ 1 * y \mapsto_c last(h_y) * \mathsf{KeyUpdStatus}(c, x, \mathsf{True}) * \mathsf{KeyUpdStatus}(c, y, \mathsf{False})$ through the SI-READ-SPEC and SI-WRITE-SPEC specifications; the read operation make no changes to the resources, while the write operation is responsible for the update reflecting that the value 1 is written to the key x. Notice how we do not need to access the resources of the invariant to reason about the body of the transaction. This is unlike the last operation, the commit operation, at which we again open the invariant to access the current state of the database: $x \mapsto h'_x * y \mapsto h'_y$, for some histories h'_x and h'_y. Together with our local resources, this is enough to satisfy the precondition of SI-COMMIT-SPEC. Whether the global points-to resources

are updated or not, as part of the postcondition of SI-COMMIT-SPEC, do not matter because we can always satisfy our invariant $\boxed{\exists h_x h_y,\ x \mapsto h_x * y \mapsto h_y}$ since the histories are existentially quantified. This concludes the proof of the *write skew* example.

`start`	‖	`start`
`read` y	‖	`read` x
`write` x 1	‖	`write` y 1
`commit`	‖	`commit`

$$Inv \triangleq \boxed{\exists h_x h_y,\ x \mapsto h_x * y \mapsto h_y}$$

Figure 3.16: *write skew* example program.

Proving safety of the *write skew* example, does little to strengthen our assurance that the separation logic specifications correctly captures snapshot isolation; snapshot isolation is known for allowing both transactions in the *write skew* example to commit, as it is impossible for the two transactions to have a write conflict. Thus, if we assert that both transaction always commit, we capture that no write conflicts can happen and therefore every execution is valid under snapshot isolation. Concretely, this amounts to proving a variant of the example in Figure 3.16 with both `commit` operations are wrapped in an assert statement: `assert(commit)`. This proof is more involved than the one we did above as we must create an invariant strong enough to show that both transactions commit. Specifically, when both transactions try to commit, we need enough information to prove that when the `commit` operation returns False, the proposition $\neg\mathsf{CanCommit}(m, ms, mc)$ does not hold. Recall that the $\mathsf{CanCommit}$ predicate is used to capture when there are no write conflicts, thus the negation $\neg\mathsf{CanCommit}(m, ms, mc)$ states that there is a write conflict. To have enough information to derive a contradiction from $\neg\mathsf{CanCommit}(m, ms, mc)$, we use the following invariant:

$$Inv \triangleq \boxed{\begin{array}{c}\exists h_x h_y,\ x \mapsto h_x * y \mapsto h_y * \big(h_x = [\,] \lor (h_x = [1] * \mathsf{Tok}_1)\big) * \\ \big(h_y = [\,] \lor (h_y = [1] * \mathsf{Tok}_2)\big)\end{array}}$$

This invariant uses a disjunction for each of the histories h_x and h_y to track the values they contain. Focusing on h_x (h_y has the same structure), it is either the case that it represents the initial state $h_x = [\,]$, or the first transaction has committed ($h_x = [1] * \mathsf{Tok}_1$). Here, Tok_1 is an exclusive token in Iris in the sense that having it twice is impossible, *i.e.,* $\mathsf{Tok}_1 * \mathsf{Tok}_1 \vdash \mathsf{False}$. In the proof of the *write skew* example with assertions around the `commit` operation, we initially create two tokens, Tok_1 and Tok_2, and take one for the proof of each of the clients. Focusing on the client program with the first transaction, Tok_1 can be used to disregard the disjunct $(h_x = [1] * \mathsf{Tok}_1)$ each time the invariant

is opened. This means that the initial state is always observed until the first transaction commits and updates the history of x from [] to [1], and puts the token in the invariant to satisfy the disjunct $(h_x = [1] * \mathsf{Tok}_1)$. Because the first transaction observes the global points-to resource of x pointing to the empty history, [], until it commits its own write, we can assert that there is no write conflict with the CanCommit predicate, as no other transaction has written to x. Therefore, the transaction must commit.

Prior to having mature transactional models in the literature for reasoning about transactional consistency, isolation levels were specified using phenomena: small examples that must be prohibited by particular levels Berenson et al. [1995]. The *write skew* example is one such phenomenon. It must be prohibited by serializability, while snapshot isolation allows it, as we have seen. Alnor Mathiasen et al. [2025] justify that their specifications for isolation level I capture level I's guarantees by showing that their specifications rule out the phenomena that isolation level I should. The work of this paper lifts the assurance from phenomena to transactional models for actual program executions. This is analogous to how it was done previously in the literature Adya [1999]; Adya et al. [2000] but without any links to program executions.

Chapter 4

Yarrow: Reconciling Effect Handlers and Region-Based Memory Management

Abstract

We present a new ML-like programming language Yarrow with algebraic effects and region-based memory management. Reconciling these programming language features into one language is challenging: the non-local control flow of algebraic effects break the stack discipline of function calls and returns that region-based memory management relies on, and multi-shot effect handlers break the invariant that regions can be exited at most once. We present a program logic, called Yarrow Logic (YL), that supports safe and modular reasoning about regions in the presence of one-shot and multi-shot effect handlers. We prove the logic sound w.r.t. the operational semantics of Yarrow which is inspired by the runtime of OCaml but refined for regions. We use YL to prove correctness of a number of case studies with algebraic effects, including checkpointing, asynchronous computation and a LIFO data structure implementation. Since all memory locations used in these case studies are allocated in regions, these case studies avoid using the less efficient garbage collected heap memory. We have formalized the Yarrow's operational semantics, the Yarrow program logic, and all our case studies using the Iris separation logic framework on top of the Rocq Prover.

4.1 Introduction

In recent years, programming language designers have adopted algebraic effects and effect handlers [Kammar et al., 2013; Plotkin and Pretnar, 2009] in a wide variety of programming languages, including OCaml [Sivaramakrishnan et al., 2021], Scala [Brachthäuser et al., 2020b], WebAssembly [Phipps-Costin et al., 2023], Koka [Leijen, 2014], Links [Hillerström and Lindley, 2016], Eff [Bauer

and Pretnar, 2015], Frank [Convent et al., 2020] and Flix [Lutze et al., 2023; Madsen, 2022]. Effect handlers give users the ability to suspend the computation by performing effects, handle the effects using user-defined handlers, and resume the computation through delimited continuations, where the effect handler is the delimiter, either at most once (one-shot effects) or multiple times (multi-shot effects). There are many successful applications of effect handlers; Dolan et al. [2015, 2017] show how effect handlers can be used to implement a number of examples ranging from I/O operations and web servers to asynchronous computation in OCaml. OCaml is known for its efficient runtime implementation of effect handlers; delimited continuations are implemented by pointers to stack segments called *fibers* [Sivaramakrishnan et al., 2021]. Each effect handler creates a new fiber when it is installed. This results in an efficient implementation of one-shot effects as delimited continuations are captured by obtaining a pointer to a stack segment without additional allocations; multi-shot effects are less efficient and require copying of continuations.

Region-based memory management Region-based memory management is a memory allocation strategy used to avoid garbage collection pressure by instead allocating and safely reclaiming memory in syntactically scoped program fragments [Tofte and Talpin, 1997; Tofte et al., 2004]. It is seen as a safer way to avoid garbage collection overhead than full manual memory management, as in C, because region-based memory management can be mostly automated [Tofte et al., 2001]. Recently, OxCaml [Georges et al., 2025; Lorenzen et al., 2024] introduced region-based memory management into OCaml by using a rich type system based on modes. One of these modes is the *locality mode* that controls (when applicable) whether a memory location is allocated using a *local* allocation or a *global* allocation. Global allocations happen on the heap, while local allocations are placed in memory regions that correspond to the scope of a function.[1] At runtime, Birkedal et al. [1996] showed that memory allocations of bounded regions can be placed on the call stack whereas unbounded regions can use linked-list structures on the heap. The OxCaml extensions do not support multi-shot effect handlers, and there does not exist any formalization of the interaction between one-shot effect handlers and the OxCaml extensions.

Yarrow: Incorporating effect handlers with region-based memory management In this paper, we formalize an operational semantics for a new language, Yarrow, an ML-like language with support for algebraic effects (one-shot and multi-shot) based on fibers [Sivaramakrishnan et al., 2021], and with region-based memory management for fine-grained memory control. A first natural question is what the combined semantics of effect handlers and

[1] OxCaml features special keywords provided whereby the programmer can have finer control over the allocation [Street, 2026].

region-based memory management should be. This is an inherently non-trivial question, as the non-local control flow that the delimited continuations of effect handlers introduce complicates reasoning about region configurations. Normally, regions are nested and follow a stack-like discipline, but with delimited continuations the control can jump between different groups of nested regions. Moreover, multi-shot effects break the invariant that regions can be exited at most once. For instance, consider the example below:

```
1 (* create a new region *)
2 region (
3   (* allocate a reference "r" pointing to "0" in this region *)
4   let r = ref local 0 in
5   (* perform the effect Foo with argument "r" *)
6   do Foo r;
7   (* load from the reference "r" *)
8   !r)
```

This example uses a memory region created with the **region**-construct; this means all local allocations within the parentheses following **region** happen inside the memory associated with this newly created region. Thus, in this example **ref local** allocates a reference inside the memory of the region which is automatically freed when the scope of the region ends (this is contrary to **ref global** which is a garbage-collected heap allocation).[2] Thereafter, the operation **do** `Foo r` performs the effect with name `Foo` and passes it `r` as an argument. Now, the questions are: What happens when we load from the reference `r`? Is it safe? Which value does the load produce? These are the types of questions we answer in this paper; for this particular example, we argue that if `Foo` is a one-shot effect, it is safe to load from `r` and the return value is `0`, whereas if `Foo` were a multi-shot effect, it would be unsafe to load from `r`. When `Foo` is a one-shot effect, loading from `r` is safe because the region that holds `r` is captured as part of the continuation of `Foo`, and is restored when the continuation is invoked. It is unsafe when `Foo` is a multi-shot effect because we want to support region-based memory management where the memory of regions are freed upon exiting them; using `r` is thus unsafe because multi-shot continuations exit regions multiple times, and therefore the first invocation of the continuation will free the region before all subsequent invocations.

Effect handlers in separation logic The separation logic framework Iris [Jung et al., 2015, 2016, 2018; Krebbers et al., 2017] has been used to make informal claims about the behavior of programming languages precise, *e.g.,* for Rust [Dang et al., 2019; Jung et al., 2017], C [Mansky and Du, 2024], OCaml [Allain and Scherer, 2026; Mével et al., 2020; Seassau et al., 2025], and WebAssembly [Legoupil et al., 2024; Rao et al., 2023]. One successful application of Iris is to study effect handlers; De Vilhena and Pottier [2021]

[2] In Chapter 4.2.2, we recap region-based memory management and the **region**-construct in more detail.

created a program logic for a language with effect handlers and mutable heap allocated references. De Vilhena and Pottier [2021] enable modular proofs in that they establish, separately, correctness of effect handlers and of the programs producing the effects handled by those handlers. This is achieved by specifying effect handlers through so called *protocols*; this is similar to how a function's specification serves as a contract between the caller and the callee.

The Yarrow logic: Reasoning about effect handlers and regions In this paper we present a new program logic, the Yarrow Logic (YL), which supports modular reasoning about effect handlers and region-based memory management in Yarrow. We use YL to specify and prove correctness of several examples featuring effect handlers, in order to exercise the Yarrow semantics, and a number of case studies, to show that effect handlers can safely avoid garbage collection by using region-based memory management. The case studies include checkpointing, asynchronous computation and a LIFO data structure implementation.[3] Reasoning about the language features of Yarrow is highly non-trivial mainly due to the following two primary challenges introduced by the non-well-bracketed control-flow of Yarrow, which YL solves *while retaining the usual modularity of higher-order impredicative separation logic*:

Fine-grained reasoning about region-allocated memory In region-based memory management we need to track the resources corresponding region-allocated memory and revoke them when the region goes out of scope. This tracking and revocation is very natural Georges et al. [2025] when control flow is well-bracketed [Timany et al., 2024b]. However, when control-flow is non-well-bracketed, *e.g.,* in the presence of effect handlers, tracking resources that should be reclaimed becomes very intricate, so much so that one may expect that the only way to cleanly track these resources is to forgo tracking individual memory locations that facilitates modular proofs, and instead explicitly reason about regions using a single high-level logical unit with all information about regions. In YL, however, we show how reasoning about regions can be decomposed into (1) a logical unit representing configuration of fibers with associated regions, which only tracks the high-level changes to regions at revocation points, and (2) stack-points-to propositions (similar to separation logic's points-to propositions used for tracking heap allocations), which allow for fine-grained (per memory location) reasoning about stack-allocated references. At the surface level YL's approach to enable fine-grained reasoning about region-allocated memory is not unlike existing approaches [Georges et al., 2025; Krogh-Jespersen et al., 2017; Timany et al., 2018]; what is particularly challenging here, and novel, is doing so in the presence of non-well-bracketed control-flow.

[3]We use the word LIFO here in place of *stack* to avoid confusing with the (call) stack of the program that we will regularly mention in the paper.

The highly dynamic nature of fibers The high-level configuration of fibers with regions that we track in YL to enable revocation is subject to sophisticated changes due to the way fibers can change upon suspension and resumption of computations in effect handlers; an effect handler may start and end several regions, or install and uninstall other effect handlers, before it uses its continuation. To solve this challenge in YL, we use the insight that the configuration of fibers and regions should be a part of the contract between an effect handler and the program performing the effect — continuations can capture regions when performing effects, making this a point of temporary revocation for one-shot effects and permanent revocation for multi-shot effects. This solution has led us to redesign the protocol concept of De Vilhena and Pottier [2021] to take into account the configuration of fibers and regions.

Contributions In summary, the contributions of this paper are:

- An operational semantics for Yarrow, an ML-like language that supports region-based memory management in the presence of (one-shot and multi-shot) effect handlers (Chapter 4.3). The presence of effect handlers makes supporting region-based memory management intricate. Prior work on region-based memory management was designed with the stack discipline of calls and returns which is violated by the delimited continuations of effect handlers. Capturing all the nuances of these intricacies requires our operational semantics to be more fine-grained than was previously needed for modeling languages with either region-based memory management, or effect handlers.

- A program logic Yarrow Logic (Chapter 4.4), abbreviated YL, for proving correctness of Yarrow programs. Reasoning about Yarrow is highly non-trivial as effect handlers break the property of well-bracketed control flow, which in the past was relied upon for revocation of resources tied to regions. Despite the challenges that come with the new combination of language features, YL retains the modularity and simplicity of existing program logics.

- Specification and verification of a number of case studies, including a LIFO data structure implementation (Chapter 4.5,) checkpointing (Chapter 4.6), and asynchronous computation (Chapter 4.7). These case studies show that in certain applications programs written using effect handlers in Yarrow can replace garbage collected heap references with region allocated references.

- A mechanization of all results in this paper using the Rocq Prover. The Rocq mechanization accompanies the paper.

In Chapter 4.2, we summarize existing work on effect handler fibers (Chapter 4.2.1) and region-based memory management (Chapter 4.2.2).

4.2 Background

We proceed by explaining how effect handler fibers and region-based memory management work separately. For the presentation of examples and case studies, we use an OCaml like syntax, the calculus of our formalization is shown in Chapter 4.3.2.

4.2.1 Background: Effect Handler Fibers

In the runtime of OCaml [Sivaramakrishnan et al., 2021], the stack is made up of a list of segments called *fibers*. A new fiber is created and appended to the list of fibers when an effect handler is installed. We proceed to explain effect fibers using the `Choose` and `State` effects: the `Choose` effect is used for backtracking in programs, and the `State` effect gives users access to a piece of state with the implementation hidden behind the effect abstraction.

Choose and State effects. Consider the effect handlers in Figure 4.1 for the `Choose` and `State` effects and ignore the code in the comments for now (we consider this code in Chapter 4.3). In the code, the `hdl_choose` definition

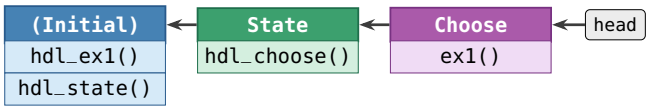
```
1  let hdl_choose f = (* region ( *)
2    try (* (global, many) *) f () with
3    | effect Choose p k -> let x = k p.1 in let y = k p.2 in (x, y)
4    | ret x -> x (* ) *)
5
6  let hdl_state init f = (* region ( *)
7    let r = ref (* local *) init in
8    try (global, once) f () with
9    | effect State arg k ->
10     (match arg with | Read -> k !r | Write x -> (r <- x; k ()))
11   | ret x -> x (* ) *)
12
13 let ex1 () = (* region ( *)
14   let x = do Choose (1, 2) in
15   do State (Write ((do State Read) + x)); do State Read (* ) *)
16
17 let hdl_ex1 () = (* region ( *)
18   assert (hdl_state 0 (fun () => hdl_choose ex1) = (1, 3))
19   (* ) *)
```

Figure 4.1: `Choose` and `State` effect handlers with a closed example program.

handles the `Choose` effect using the **`try`**-construct on line 2. Using the **`try`**-construct, it is safe to perform the `Choose` effect (on line 14) inside the function for which we install the effect handler (on line 18). Much like a **`match`** case, the **`try`**-construct has two cases: one for when `f` performs the `Choose` effect (the handler branch), and one for when `f` terminates with a value (the return branch). The handler branch takes as arguments the user-provided argument `p`, and a delimited continuation `k`, where the **`try`**-construct acts as the delimiter, for resuming computation inside f at the exact point where the effect was

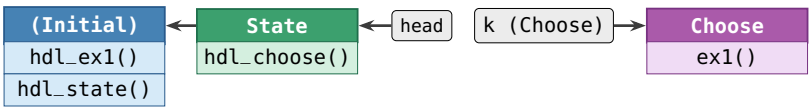
performed. The effect **Choose** is a multi-shot effect meaning the continuation can be used more than once; the implementation in Figure 4.1 makes use of this by evaluating the continuation `k` on both components of the argument pair `p`, after which it returns the result in a pair. The `State` effect is implemented on line 8 in `hdl_state` by internally using the heap allocated reference `r`. We use `Read` and `Write` as notation for sum types, to branch on whether the user of the effect wants to read or write to the state, *i.e.,* `Read` is `InjL ()` and `Write x` is `InjR x`.

Effect fibers Let us consider the layout of fibers at different points in the execution of the closed example on line 17 in Figure 4.1. The `ex1` definition on line 13 uses effect handlers for `Choose` and `State`. On line 18, `hdl_ex1` executes `ex1` using the two effect handlers `hdl_choose` and `hdl_state`. The state is initialized to `0` which results in the computation returning the pair `(1, 3)`, as both executions created on line 14 with the `Choose` effect share the same state of the `State` effect which behind the effect abstraction is a heap allocated reference. Just before line 14 is executed the fiber configuration looks like this:



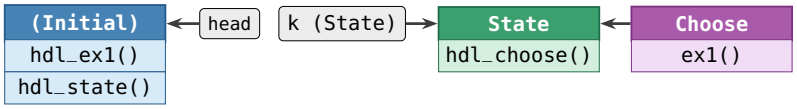
The layout of the fibers naturally follows the order in which they were installed (*i.e.,* they reflect the order in `hdl_ex1`): the head of the list of fibers always points to the most recently installed fiber, in this case the fiber for the `Choose` effect. Inside the fiber installed by the `Choose` effect handler in `hdl_choose`, we have the call stack for the `ex1` function. The `Choose` fiber then points back to the fiber installed by the `State` effect handler in `hdl_state`. This fiber contains the call stack for the `hdl_choose` function. Lastly, the `State` fiber points back to the *initial* fiber; there is always an initial fiber that contains the call stack of the top most functions of programs before any effect handlers are installed. In this case, the initial fiber contains the call stack for the `hdl_state` and `hdl_ex1` functions.

When the `Choose` effect is performed on line 14 in Figure 4.1, computation is suspended and the control is transferred to the effect handler inside `hdl_choose`. This has the following effect on the fiber configuration:

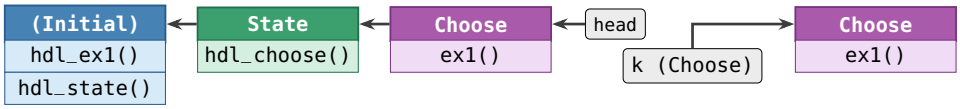


Above, we see that the head of the list now points to the `State` fiber. The delimited continuation `k` used inside the effect handler `Choose` is a pointer to the `Choose` effect fiber; when an effect is performed, we traverse the list of fibers from the head until the correct fiber is found and create the continuation as a

pointer to this fiber. The continuation `k` is thus a pointer to all the call stacks required to resume computation at the point where it was suspended when the `Choose` effect was performed. Similarly, when the `State` effect is performed, *e.g.*, on line 15 in Figure 4.1, the fiber configuration looks like this:



When the computation is resumed inside the effect handlers of `State` or `Choose` by using the continuation, the fibers that the continuation points to are reattached to the list of fibers that the head points to. Because the `Choose` effect is a multi-shot effect, in that its continuation `k` is used many times in the effect handler on line 3 in Figure 4.1, we must make copies of the fibers that the continuation of the `Choose` effect points to. Thus, when the continuation is used to resume computation in the `Choose` effect, we create a copy of the fibers that the continuation points to and attach those to the list of fibers that the head points to which still leaves the original fibers in memory:



4.2.2 Background: Region-based Memory Management

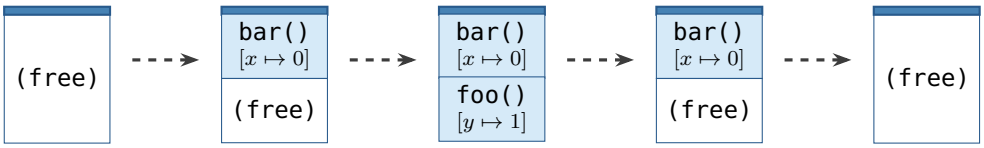
In this subsection, we focus on regions (without effects). In the introduction (Chapter 4.1), we already saw a glimpse of the kind of regions we consider in this paper, namely regions à la Tofte and Talpin [1997] but with a construct similar to that of Lorenzen et al. [2024] and Georges et al. [2025]. The work by Lorenzen et al. [2024] and Georges et al. [2025] differentiates between two types of allocations: local allocations in regions and global allocations on the heap. To be precise, the construct **`region`** (...), as seen in the introduction (Chapter 4.1), is used to create lexically scoped regions with a stack-like discipline. All local allocations are associated with the lexically nearest enclosing region construct. A *locality mode* l, which can be either **`local`** or **`global`**, is used to differentiate between local allocations in regions and global heap allocations; a reference to a value `v` is allocated using **`ref`** `l  v` where `l` is the locality mode. We remark that the mode notation used here are arguments to program constructs and not types (our examples are untyped).[4] To explain the semantics of regions, and in particular nested regions, we consider the example in Figure 4.2. In this example, both functions create a region around their function bodies. Indeed, as a default we associate regions with function bodies (insertion of **`region`** can be delegated to the compiler), but exceptions can be made to this to principle,

[4]In the work by Georges et al. [2025] they use mode annotations on types with the **`@`** character and superscript notation for modes of program constructs.

```
1 let foo x = region (let y = ref local 1 in (x, y))
2
3 let bar () = region (let x = ref local 0 in let p = foo x in !p.1)
```

Figure 4.2: Nested Regions Example.

we will see one example of where such an exception can be useful later in Chapter 4.7. The function `foo` returns a pair; the first entry in the pair is the argument that `foo` receives, and the second entry is a reference, locally allocated in its own region. The function `bar` creates a local allocation `x` in its region, and then calls `foo` with `x` before it loads from the first value of the pair that it has gotten from the call to `foo`. Below, we visualize how the stack evolves throughout the execution of the function `bar`, the arrows signify changes to the stack configuration:



Each stack configuration consists of zero or more stack frames. Under each stack frame, we also write the allocations made in the region associated with the function that uses this stack frame. We emphasize that regions are not necessarily a part of the stack frame, the memory underlying regions can be placed elsewhere on the heap (usually bounded regions go on the call stack and unbounded on the heap, see Birkedal et al. [1996]). However, the regions follow a stack discipline, enforced by the **`region`**-construct: before `foo` returns the pair in Figure 4.2, the stack configuration is as the third configuration above with the call stack, and thus the regions of both functions are still in memory. After `foo` returns its pair and execution resumes in `bar`, we move along to the fourth stack configuration, where the region of `foo` has been freed, and its contents deallocated. When `bar` loads from the first entry in the pair it received from `foo`, it is safe as the first entry is `x`, which is still alive, in `bar`'s own region. Note that if the load had been from the second entry in the pair, it would have been unsafe, as `y` lived in the region of `foo`, which was freed when `foo` returned.

4.3 Operational Semantics

In this section, we explain our new operational semantics for Yarrow, an ML-like language with effect handlers and region-based memory management. Before we explain the details of the operational semantics in (Chapter 4.3.2), we give an overview of the ideas behind the operational semantics in (Chapter 4.3.1).

4.3.1 Overview

We introduce our new operational semantics using two examples, the first example focuses on one-shot effects and the second example focuses on multi-shot effects.

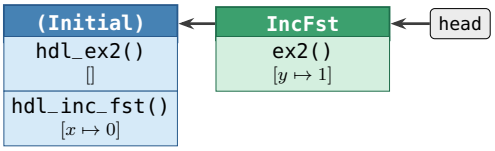
One-shot effects For one-shot effects, we illustrate the point that continuations capture regions, and we can resume the computation with captured regions as if nothing happened to these regions (this is not the case when we consider multi-shot effects next). Consider the *Increment First Example* in Figure 4.3. The figure contains two definitions `hdl_inc_fst` and `ex2`; at the end

```
1  let hdl_inc_fst f = region (
2    let x = ref local 0 in
3    try (global, once) f x with
4    | effect IncFst p k -> (p.1 <- !p.1 + 1); k ()
5    | ret x -> x)
6
7  let ex2 x = region (
8    let y = ref local 1 in let p = (x, y) in
9    do IncFst p; assert (!p.1 + !p.2 = 2))
10
11 let hdl_ex2 () = region (hdl_inc_fst ex2)
```

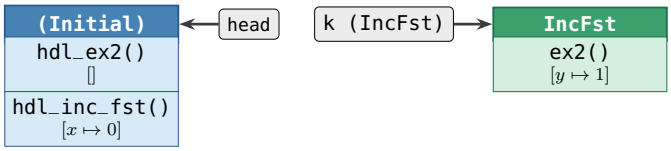
Figure 4.3: Increment First Example.

of the figure, `hdl_inc_fst` is applied to `ex2` in `hdl_ex2`. After creating a region, `hdl_inc_fst` allocates a reference `x` pointing to `0` locally in this region. Next, an effect handler with operation `IncFst` is installed around the application of the function argument to the newly allocated reference `x`. In our ML-like language, we use mode notation with the **`try`**-construct: the **`try`**-construct takes a pair of modes where the first entry is a locality mode that can be **`global`** or **`local`**, just like with references, and the second entry is an affinity mode that can be either **`once`** or **`many`**. The locality mode controls whether the closure of the continuation `k` in the handler branch is allocated locally in the region where the effect handler is situated, or globally on the heap. Later in Chapter 4.3.2, we will also see how function closures can be allocated in regions instead of on the heap, but all the functions we have seen so far have non-empty closures. The affinity mode controls whether the effect is a one-shot or multi-shot effect, *i.e.,* how many times the continuation `k` can be called in the handler branch. For the effect handler in the `hdl_inc_fst` definition of Figure 4.3, we use the **`(global, once)`** combination; **`once`** because it is a one-shot effect, and for simplicity to focus on the core parts of regions and effects, we use the **`global`** mode in this example (our case studies use continuations with locally allocated closures). The handler branch for the `IncFst` effect handler takes a pair and increments a reference in the pair's first entry. No assumptions are made about the second entry in the pair. Moving our attention to the `ex2` definition, this function performs the effect `IncFst` with argument `p`. The first

entry of the pair `p` is the function argument `x` of `ex2`, whereas the second entry is a reference `y` locally allocated inside the region `ex2`. The configuration of fibers just before **`do`** `IncFst p` is executed is this:

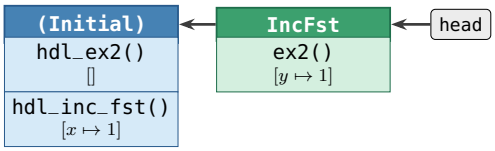


Above, we first notice how, contrary the fiber configurations in Chapter 4.2.1, we have regions inside the stack frames like we saw in Chapter 4.2.2. This fiber configuration is made up of two fibers: the head of the list points to the fiber installed by the `IncFst` effect, the `IncFst` fiber again points to the initial fiber that always marks the end of the stack. Observe how the two entries in the pair `p` used as argument to the effect `IncFst` are references allocated in different regions: `x` is allocated in the region created by `hdl_inc_fst`, and the reference `y` is allocated in the region created by the `ex2` function. Further, the `ex2` function is located in the fiber installed by the `IncFst` effect handler; this follows the structure of the program in Figure 4.3 where `ex2` is called after the `IncFst` effect handler is installed. When **`do`** `IncFst p` is executed on line 9, and control is transferred to the effect handler, the fiber configuration changes to this:



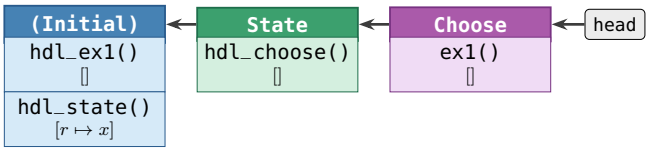
The head now points to the initial fiber, and the continuation available inside the handler branch of the `IncFst` effect points to the `IncFst` fiber. As the handler branch increments the reference `x`, through the first entry in the pair it receives, it executes safely; the reference `x` sits in a region which is a part of the current stack. We deem it undefined behavior to use local references that are not part of the current stack, *i.e.,* local references allocated in a region of a fiber that a continuation points to and not a fiber from the list of fibers that make up the current stack, such as if the increment had instead been done on the reference `y` through the second entry in the pair. We classify this as undefined behavior for two reasons: (1) to ensure that references can only be used inside the scope of the region they belong to; and (2) to ensure that reclamation of memory used by fibers of continuations should not depend on anything except on how the continuation itself is used.

After the handler branch of `IncFst` is executed, and the computation is resumed by using the continuation, the fiber configuration is this:

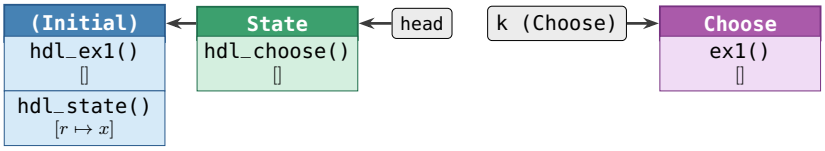


We are back to the configuration we had before **do** `IncFst p` was executed on line 9, except the handler branch of `IncFst` has been executed in the meantime which results in `x` now storing `1` inside the region of the `hdl_inc_fst` function. This fiber configuration implies that the assertion on line 9 holds.

Multi-shot effects For multi-shot effects, we can not resume computation using continuations that capture regions as part of the fibers they point to and keep using the reference allocated in these regions. To illustrate this, we reconsider the example with the multi-shot effect `Choose` and the one-shot effect `State` from Figure 4.1, but this time we also take into account the code in the comments for regions and modes. The reference `r` inside the handler for the `State` effect on line 7 is now a local reference. We use the **`once`** mode for the `State` effect and the **`many`** mode for the `Choose` effect, as they are respectively one-shot and multi-shot effects. Just before the `Choose` effect is performed on line 14, the fiber configuration is shown below:



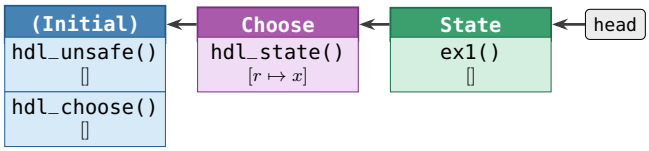
This is similar to the fiber configuration we saw in Chapter 4.2.1 when we also considered the closed example `hdl_ex1`. This time regions are included in the fiber configuration which illustrates that the reference used to implement the `State` effect sits inside the region of the `hdl_state` function. When the `Choose` effect is performed on line 14, and control is suspended to the effect handler in `hdl_choose`, the fiber configuration changes to this:



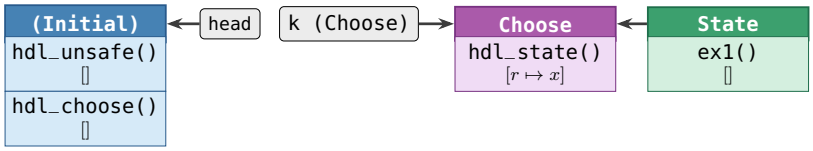
The continuation of `Choose` captures a fiber with an empty region, but this is not always the case; suppose the `Choose` effect handler is instead installed before the `State` handler as in the code below:

```
1 let hdl_unsafe () =
2   region (hdl_choose (fun () => hdl_state 0 ex1))
```

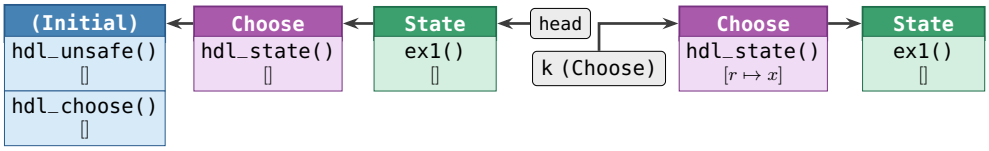
Before performing the `Choose` effect in `ex1` using `hdl_unsafe`, the fiber configuration looks like this:



This fiber configuration reflects that we have changed the order in which the effect handlers are installed. When the `Choose` effect is performed in `ex1`, the fiber configuration changes to the one below where the continuation `k` in the `Choose` effect handler now captures both the `Choose` and `State` fiber which includes a non-empty region:



The local reference inside the region of `hdl_state` is implicitly deallocated when the multi-shot effect `Choose` is performed. This is because the continuation `k` captures the region of `hdl_state`, and `k` is a multi-shot continuation; multi-shot continuations are created by making copies of the fibers they point to, but we can not make copies of regions and continue using the same local references inside these regions when computation is resumed as the underlying memory of the regions may have changed. When computation is resumed using the continuation `k` in the effect handler for `Choose`, we return to the fiber configuration we had before `Choose` was performed inside `ex1` on line 14 in Figure 4.1, except the reference `r` is no longer usable:



This fiber configuration implies that it is unsafe to use the `State` effect on lines 19 and 20 in the `ex1` function when using `hdl_unsafe`, as the reference `r` is no longer usable as part of the list of fibers that makes up the current stack (the data `x` may still be in the region of `hdl_state` at runtime but not at the location `r` represents).

Skippable detour about references and multi-shot continuations References captured by multi-shot continuations can be reused under certain limiting restrictions. For example, Muhcu et al. [2025] describes a runtime for a language with multi-shot effects where continuations can capture stack reference as part of their closures by using an indirection layer for references. In that setting, a stack reference points to a reference in an indirection layer

which again points to the actual stack. This approach [Muhcu et al., 2025], however, only works as long as no two continuations that capture the same fiber are simultaneously active (*i.e.,* one of the continuations must return before the other one is called). If we instead took the approach of reusing the same region, or parts of it, for multiple fibers to preserve the local references, we could no longer implement true region-based memory management where the memory underlying regions is freed when code exits the scope of a region; a region can be exited more than once, making it unsafe for all invocations of a continuation, apart from the very first, to use memory in regions.

4.3.2 Syntax and Operational Semantics

In this subsection, we formalize the operational semantics for the runtime model we gave an overview of in the previous subsection. We start by defining the expressions and values of our language below, where we omit products, sums, unary operators and binary operators for brevity.

$$
\begin{aligned}
& z \in \mathbb{Z} \qquad x \in \mathit{Var} \qquad \ell \in \mathit{Loc} \qquad \mathit{id} \in \mathrm{Cid} \qquad \mathit{op} \in \mathrm{Operation} \\
\alpha \in \mathrm{Addresses} ::= {} & \mathbf{sa}\,\ell \mid \mathbf{ha}\,\ell \qquad l \in \mathrm{Locality} ::= \mathbf{local} \mid \mathbf{global} \\
a \in \mathrm{Affinity} ::= {} & \mathbf{once} \mid \mathbf{many} \\
v \in \mathit{Val} ::= {} & () \mid z \mid x \mid \mathbf{true} \mid \mathbf{false} \mid \alpha \mid \lambda^{(\alpha,\mathit{id})} f\ x.\ e \mid \mathbf{cont}^{(\alpha,\mathit{id})} K\ \theta\ \ell \\
& \mid \mathbf{kont}^{(\alpha,\mathit{id})} K\ \theta \mid \ldots \\
e \in \mathit{Expr} ::= {} & v \mid x \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \mid e_1\ e_2 \mid \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \mid \mathbf{ref}\ l\ e \\
& \mid\ !e \mid e_1 \leftarrow e_2 \mid (\mathbf{match}\ e_1\ \mathbf{with}\ \mathbf{inl}\ x_1 => e_1 \mid \mathbf{inj}\ x_2 => e_2) \\
& \mid \lambda^{l}\ f\ x.\ e \mid \mathbf{region}\ e \mid \mathbf{end}\ e \mid \mathbf{do}\ \mathit{op}\ e \mid \mathbf{effect}\ \mathit{op}\ v\ K \\
& \mid (\mathbf{try}\ (l, a)\ e_1\ \mathbf{with}\ \mathit{op}\ x\ k.\ e_2 \mid \mathbf{ret}\ x.\ e_3) \\
& \mid (\mathbf{installed}\ (l, a)\ e_1\ \mathbf{with}\ \mathit{op}\ x\ k.\ e_2 \mid \mathbf{ret}\ x.\ e_3) \mid \ldots
\end{aligned}
$$

Expressions and values The values in the language include the unit value, integers, booleans, addresses, λ-abstractions, one-shot continuations and multi-shot continuations. Addresses are either stack addresses $\mathbf{sa}\,\ell$, with a logical location ℓ, or heap address $\mathbf{ha}\,\ell$ likewise with a location. In λ-abstractions, f is the recursive occurrence in the body e, x is an argument and the superscript pair (α, id) says that the closure of the abstraction, represented with closure identifier id, is located at address α. This way of tracking closures is similar to Georges et al. [2025], even though Georges et al. formalize a language without effect handlers, and it allows us to reason about the lifetime of closures; closures can be heap or stack allocated. For our continuation values, we use the same mechanism for tracking closures. The value for one-shot continuations $\mathbf{cont}^{(\alpha,\mathit{id})} K\ \theta\ \ell$, takes an evaluation context K (which we define momentarily), a list of fibers θ (which we also define momentarily) and a location ℓ that is used to enforce the affinity of one-shot continuations. The value for multi-shot

continuation $\mathbf{kont}^{(\alpha,id)} K\ \theta$ is like the value for one-shot continuations, except it does not have the location that we use to enforce affinity.

The expressions in our language include values, variables, conditionals, sequencing, function application, allocation of local references in regions and global references on the heap, and expressions for storing and loading from references. In addition, we have the expression version of λ-abstractions $\lambda^l\ f\ x.\ e$; this is used to allocate function closures, the local mode l in superscript determines whether the closure is allocated in a region or on the heap. Further, we have the **region** e expression for starting a region around an expression e, and **end** e which we use to mark when the scope of a region ends and its underlying memory is freed. Lastly, we have a number of expressions for algebraic effects: we saw the expression **try** $(l, a)\ e_1$ **with** $op\ x\ k.\ e_2$ | **ret** $x.\ e_3$ throughout Chapter 4.3.1 for installing an effect handler and its underlying fiber. Contrary to the previous work on program logics for effect handlers in Iris [De Vilhena and Pottier, 2021], our **try**-expression is parameterized with a pair of modes for controlling whether the effect is one-shot or multi-shot, and if the closure of the continuation k is allocated locally in a region or on the heap. Similarly to how **end** e marks the end of a region, **installed** $(l, a)\ e_1$ **with** $op\ x\ k.\ e_2$ | **ret** $x.\ e_3$ marks the end of a fiber: when the expression e_1 evaluates to a value, the return branch e_3 is executed with the value as argument, and at the same time, the fiber with operation op that the **try**-expression installed is removed from the list of fibers that make up the stack. When an effect is performed using **do** $op\ e$, we use the mechanism from De Vilhena and Pottier [2021] to capture the paused computation in continuation values, this is what the **effect** $op\ v\ K$ expression is for. We will see exactly how this works when we present the reduction rules of our language which are based on *evaluation contexts*.

Evaluation contexts We use evaluation contexts K below to define the language. $\bullet$ is the empty context, and $K[e]$ is notation for filling out the evaluation context K with the expression e. We list the evaluation contexts of our language below (again, we omit contexts for products, sums, unary operators and binary operators for brevity).

$$
\begin{aligned}
N' \in \mathit{NeutralCtx}' ::= {} & \mathbf{let}\, x = \bullet\, \mathbf{in}\, e \mid e\ \bullet \mid \bullet\ v \mid \mathbf{if}\ \bullet\ \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid \mathbf{ref}\ l\ \bullet \\
& \mid\ !\bullet \mid e \leftarrow \bullet \mid \bullet \leftarrow v \mid \mathbf{end}\ \bullet \mid \mathbf{do}\ op\ \bullet \\
& \mid (\mathbf{match}\ \bullet\ \mathbf{with}\ \mathbf{inl}\ x_1 => e_1 \mid \mathbf{inj}\ x_2 => e_2) \mid \ldots \\
K' \in \mathit{EvalCtx}' ::= {} & N' \mid (\mathbf{installed}\ (l, a)\ \bullet\ \mathbf{with}\ op\ x\ k.\ e_2 \mid \mathbf{ret}\ x.\ e_3) \\
N \in \mathit{NeutralCtx} ::= {} & \bullet \mid N'[N] \qquad K \in \mathit{EvalCtx} ::= \bullet \mid K'[K]
\end{aligned}
$$

We say that all evaluation contexts except those for the **installed**-expression are *neutral contexts*. We need neutrals contexts for when we capture delimited continuations using the **effect** $op\ v\ K$ expression in the reduction rules.

Reduction rules The program state (h, θ, ℓs, ids) used in our reduction rules is defined as follows:

$$h \in \mathit{Heap} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Val} + \text{Cid}) \quad \ell s \in \text{Locations} \triangleq \mathcal{P}_{\text{fin}}(\mathit{Loc})$$
$$\mathit{ids} \in \text{Identifiers} \triangleq \mathcal{P}_{\text{fin}}(\text{Cid}) \quad \delta \in \text{StackFrame} \triangleq \text{List}\ (\mathit{Loc} \times (\mathit{Val} + \text{Cid}))$$
$$\omega \in \text{StackFrames} \triangleq \text{List StackFrame} \quad \theta \in \text{Fibers} \triangleq \text{List}\ (\text{Operation} \times \text{StackFrames})$$
$$\sigma \in \text{State} \triangleq \mathit{Heap} \times \text{Fibers} \times \text{Locations} \times \text{Identifiers}$$

The state consists of the heap h, the installed fibers θ that make up the stack, a set of locations ℓs and a set of closure identifiers ids. In the definition of the heap h above, we see how the heap consists of both values and closure identifiers. We keep around a set ids in the program state that tracks all the previous used closure identifiers to make sure we are always generating new identifiers when allocating closures. Fibers, like the heap, can contain both values and closure identifiers. Fibers are built using a number of layers: we retain the association of regions with stack frames that we used throughout Chapter 4.3.1. Thus, fibers are defined as a list of stack frames, a stack frame is a list of locations paired with either the value or the closure identifier that it stores. The set of logical locations ℓs tracks all the locations that have previously been used for local allocations. When making a local allocation, we can not rely on the fibers in the program state alone when we want to generate a fresh location; continuation values also hold fibers with locations in them. Thus, we use the set ℓs for new local locations. Having defined the program state, we can define the reduction relations for the operational semantics of Yarrow.

Our small-step operational semantics consists of two reduction relations, the base step ($\rightsquigarrow$) and the context step ($\rightarrow$), both with signature $\mathit{Expr} \times \text{State} \rightarrow \mathit{Expr} \times \text{State}$. The context step only has one reduction rule shown below:

$$\frac{(e, \sigma) \rightsquigarrow (e', \sigma')}{(K[e], \sigma) \rightarrow (K[e'], \sigma')}\ \textsc{Eval-Ctx-Step}$$

This rule evaluates an expression in an evaluation context $K[e]$ to $K[e']$, if there is a base step reduction from e to e'. In Figure 4.4, we display an excerpt of base step reduction rules unrelated to algebraic effects, and in Figure 4.5 we display another excerpt that are about algebraic effects. First, we focus on the reduction rules that are not about algebraic effects.

In the Region rule, the **`region`** e expression steps to the **`end`** e expression, doing so, a new empty stack frame is inserted in the most recently installed fiber. When the expression e in **`end`** e evaluates to a value v, the top most stack frame δ is freed, in the End rule, which effectively ends the region as the local references that reside in the stack frame δ are no longer available. Local references are always allocated into the top most stack frame which corresponds to the most recently started region (Alloc-Local). Loads and

REGION
$$\big(\textbf{region}\ e, (h, \theta +\!\!+ [(op,\ \omega)], \ell s, ids)\big) \rightsquigarrow \big(\textbf{end}\ e, (h, \theta +\!\!+ [(op,\ \omega +\!\!+ [[]])], \ell s, ids)\big)$$

END
$$\big(\textbf{end}\ v, (h, \theta +\!\!+ [(op,\ \omega +\!\!+ [\delta])], \ell s, ids)\big) \rightsquigarrow \big(v, (h, \theta +\!\!+ [(op,\ \omega)], \ell s, ids)\big)$$

ALLOC-GLOBAL
$$\frac{\ell \text{ fresh in } h}{\big(\textbf{ref global}\ v, (h, \theta, \ell s, ids)\big) \rightsquigarrow \big(\textbf{ha}\,\ell, (h[\ell := v], \theta, \ell s, ids)\big)}$$

ALLOC-LOCAL
$$\frac{\ell \text{ fresh in } \ell s}{\begin{array}{c}\big(\textbf{ref local}\ v, (h, \theta +\!\!+ [(op,\ \omega +\!\!+ [\delta])], \ell s, ids)\big) \rightsquigarrow \\ \big(\textbf{sa}\,\ell, (h, \theta +\!\!+ [(op,\ \omega +\!\!+ [\delta +\!\!+ [(\ell, v)]])], \ell s \uplus \{\ell\}, ids)\big)\end{array}}$$

LOAD-GLOBAL
$$\big(!(\textbf{ha}\,\ell), (h[\ell := v], \theta, \ell s, ids)\big) \rightsquigarrow \big(v, (h[\ell := v], \theta, \ell s, ids)\big)$$

STORE-LOCAL
$$\begin{array}{c}\big((\textbf{sa}\,\ell) \leftarrow v_1, (h, \theta_1 +\!\!+ [(op,\ \omega_1 +\!\!+ [\delta_1 +\!\!+ [(\ell, v_2)] +\!\!+ \delta_2] +\!\!+ \omega_2)] +\!\!+ \theta_2, \ell s, ids)\big) \rightsquigarrow \\ \big((), (h, \theta_1 +\!\!+ [(op,\ \omega_1 +\!\!+ [\delta_1 +\!\!+ [(\ell, v_1)] +\!\!+ \delta_2] +\!\!+ \omega_2)] +\!\!+ \theta_2, \ell s, ids)\big)\end{array}$$

FUNCTION-GLOBAL
$$\frac{\ell \text{ fresh in } h \qquad id \text{ fresh in } ids}{\big((\lambda^{\textbf{global}}\ f\ x.\ e), (h, \theta, \ell s, ids)\big) \rightsquigarrow \big((\lambda^{(\textbf{ha}\,\ell, id)} f\ x.\ e), (h[\ell := id], \theta, \ell s, ids \uplus \{id\})\big)}$$

FUNCTION-LOCAL
$$\frac{\ell \text{ fresh in } \ell s \qquad id \text{ fresh in } ids}{\begin{array}{c}\big((\lambda^{\textbf{local}}\ f\ x.\ e), (\theta +\!\!+ [(op,\ \omega +\!\!+ [\delta])], \ell s, ids)\big) \rightsquigarrow \\ \big((\lambda^{(\textbf{sa}\,\ell, id)} f\ x.\ e), (h, \theta +\!\!+ [(op,\ \omega +\!\!+ [\delta +\!\!+ [(\ell, id)]])], \ell s \uplus \{\ell\}, ids \uplus \{id\})\big)\end{array}}$$

APPLICATION
$$\frac{(\alpha, id) \text{ alive in } \sigma}{\big((\lambda^{(\ell, id)} f\ x.\ e)\ v, \sigma\big) \rightsquigarrow \big(e[(\lambda^{(\ell, id)} f\ x.\ e)/f][v/x], \sigma\big)}$$

Figure 4.4: Selected base steps reduction rules.

stores to local references (Store-Local) happen in place in the stack frame where the reference was allocated. Global heap references follow a standard semantics, *e.g.,* Alloc-Global and Load-Global. We also use references when allocating function closures: In the Function-Global and Function-Local rules, the expression $\lambda^{l}\ f\ x.\ e$ steps to the value $\lambda^{(\alpha,id)} f\ x.\ e$, id is a new identifier in the set of closure identifiers ids in the program state. The superscript (α, id) indicates that the address of the closure for this λ-abstaction (the closure being represented by the closure id id) is α. The address α, which is either a heap address or a stack address depending on the locality mode l, is set to point to the closure identifier. When applying a λ-abstaction, we check in the antecedent of the Application rule whether the closure is still alive. That is, the phrase "(α, id) alive in σ" means that the address α points to id on the heap h in σ, or the address α points to id in any of the stack frames inside the fibers θ in the state σ.

Next we turn our attention to the reduction rules in Figure 4.5 about algebraic effects. The Handler-Install rule installs an effect handler and its corresponding fiber by stepping from the **try**-expression to the corresponding **installed**-expression. Initially, the newly installed fiber in the program state does not have any stack frames. As mentioned earlier in this subsection, when an effect is performed using **do** *op* v, the expression is replaced with **effect** *op* $v\ \bullet$ (see the Do rule). From here, an evaluation context K is created in the **effect** *op* $v\ K$ expression by consuming surrounding neutral context using the Eff rule. If the evaluation context surrounding **effect** *op* $v\ K$ is not a neutral context, it is by definition a **try**-context. When such a context is encountered, and the operation of the **effect** *op* $v\ K$ expression is not the same as the operation in the **try**-context, we simply keep propagating upwards in the program, as shown in the Handler-NEQ rule, until the correct effect handler is found. When the operation matches and the correct effect handler is found, we create a continuation value and substitute it, together with the argument provided when performing the effect, into the handler branch of the effect handler, see Handler-Global-Multi and Handler-Local-One. For instance, in the Handler-Global-Multi reduction rule, where the effect is multi-shot and the continuation closure is stored on the heap, the continuation value

$$\begin{aligned}&\mathbf{kont}^{(\mathbf{ha}\,\ell,id)}(\mathbf{installed}\ (\mathbf{global},\mathbf{many})\ K\ \mathbf{with}\ op\ x\ k.\ e_1 \mid \mathbf{ret}\ x.\ e_2)\\ &\qquad([(op,\omega)] +\!\!+ \theta_2)\end{aligned}$$

is created. This value is made up of three parts: (1) the pair $(\mathbf{ha}\,\ell, id)$ stating that the continuation closure represented by the identifier id is stored in the heap location ℓ (2) an evaluation context consisting of K, from the **effect** *op* $v\ K$ expression, around which the effect handler is reinstalled (reinstalling the effect handler gives *deep-handler* semantics contrary to *shallow*-handlers where the effect handler is not reinstalled) (3) the fibers that the continuation captures, *i.e.,* all the fibers up to and including the fiber installed by effect handler. The

$$\textsc{Do}\quad \big(\textbf{do}\ op\ v, \sigma\big) \rightsquigarrow \big(\textbf{effect}\ op\ v\ \bullet, \sigma\big)$$

$$\textsc{Eff}\quad \big(N[\textbf{effect}\ op\ v\ K], \sigma\big) \rightsquigarrow \big(\textbf{effect}\ op\ v\ N[K], \sigma\big)$$

$$\textsc{Handler-Install}\quad \big(\textbf{try}\ (l,a)\ e\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2, (h, \theta, \ell s, ids)\big) \rightsquigarrow \big(\textbf{installed}\ (l,a)\ e\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2, (h, \theta +\!\!+ [(op,\ [])], \ell s, ids)\big)$$

$$\textsc{Handler-NEQ}\quad \frac{op \neq op'}{\big(\textbf{installed}\ (l,a)\ (\textbf{effect}\ op\ v\ K)\ \textbf{with}\ op'\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2, \sigma\big) \rightsquigarrow \big(\textbf{effect}\ op\ v\ (\textbf{installed}\ (l,a)\ K\ \textbf{with}\ op'\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2), \sigma\big)}$$

$$\textsc{Handler-Local-One}\quad \frac{\ell_1 \text{ fresh in } h \qquad \ell_2 \text{ fresh in } \ell s \qquad id \text{ fresh in } ids \qquad op \text{ not installed in } \theta_2}{\begin{array}{c}\big(\textbf{installed}\ (\textbf{local}, \textbf{once})\ (\textbf{effect}\ op\ v\ K)\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2,\\ (h, \theta_1 +\!\!+ [(op', \omega +\!\!+ [\delta])] +\!\!+ [(op, \omega)] +\!\!+ \theta_2, \ell s, ids)\big) \rightsquigarrow\\ \big(e_1[v/x][(\textbf{cont}^{(\textbf{sa}\,\ell, id)}\ (\textbf{installed}\ (\textbf{local}, \textbf{once})\ K\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2)\\ ([(op, \omega)] +\!\!+ \theta_2)\ \ell_1)/k], (h[\ell_1 := \textbf{false}], \theta_1 +\!\!+ [(op', \omega +\!\!+ [\delta +\!\!+ [(\ell_2, id)]])],\\ \ell s \uplus \{\ell_2\}, ids \uplus \{id\})\big)\end{array}}$$

$$\textsc{Handler-Global-Multi}\quad \frac{\ell \text{ fresh in } h \qquad id \text{ fresh in } ids \qquad op \text{ not installed in } \theta_2}{\begin{array}{c}\big(\textbf{installed}\ (\textbf{global}, \textbf{many})\ (\textbf{effect}\ op\ v\ K)\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2,\\ (h, \theta_1 +\!\!+ [(op, \omega)] +\!\!+ \theta_2, \ell s, ids)\big) \rightsquigarrow\\ \big(e_1[v/x][(\textbf{kont}^{(\textbf{ha}\,\ell, id)}\ (\textbf{installed}\ (\textbf{global}, \textbf{many})\ K\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2)\\ ([(op, \omega)] +\!\!+ \theta_2))/k], (h[\ell := id], \theta_1, \ell s, ids \uplus \{id\})\big)\end{array}}$$

$$\textsc{Handler-Return}\quad \big(\textbf{installed}\ (l,a)\ v\ \textbf{with}\ op\ x\ k.\ e_1 \mid \textbf{ret}\ x.\ e_2, (h, \theta +\!\!+ [(op, \omega)], \ell s, ids)\big) \rightsquigarrow \big(e_2[v/x], (h, \theta, \ell s, ids)\big)$$

$$\textsc{Cont-One}\quad \frac{(\alpha, id) \text{ alive in } (h, \theta_1, \ell s, ids)}{\big((\textbf{cont}^{(\alpha, id)} K\ \theta_2\ \ell)\ v, (h[\ell := \textbf{false}], \theta_1, \ell s, ids)\big) \rightsquigarrow \big(K[v], (h[\ell := \textbf{true}], \theta_1 +\!\!+ \theta_2, \ell s, ids)\big)}$$

$$\textsc{Cont-Multi}\quad \frac{(\ell, id) \text{ alive in } (h, \theta_1, \ell s, ids)}{\big((\textbf{kont}^{(\alpha, id)} K\ \theta_2)\ v, (h, \theta_1, \ell s, ids)\big) \rightsquigarrow \big(K[v], (h, \theta_1 +\!\!+ (rem\_locs\ \theta_2), \ell s, ids)\big)}$$

Figure 4.5: Selected base step reduction rules for effects handlers.

Handler-Local-One rule follows a similar pattern when creating a continuation value for a one-shot effect with a locally allocated closure, except the continuation closure is stored at a stack location ℓ_2, and we allocate a heap location ℓ_1 to enforce that the continuation is not called more than once:

$$\begin{aligned}&\mathbf{cont}^{(\mathbf{sa}\,\ell,id)}(\mathbf{installed}\ (\mathbf{local},\mathbf{once})\ K\ \mathbf{with}\ op\ x\ k.\ e_1 \mid \mathbf{ret}\ x.\ e_2)\\ &\qquad([(op,\omega)] +\!\!+ \theta_2)\ \ell_1.\end{aligned}$$

For brevity, we have omitted the reduction rules for multi-shot effects with local continuation closures and one-shot effects with global continuation closures, the rules we have shown (Handler-Global-Multi and Handler-Local-One) cover all the aspects of the rules. When applying a one-shot continuation in the Cont-One reduction rule, we make sure the continuation has not been used before by checking that the location ℓ points to false, whereafter we flip it to true, and that the continuation closure is still alive like in the Application rule. Afterwards, the continuation is called with a value v by placing v in the context of the continuation value K and pushing the fibers of the continuation value θ_2 onto the stack. In Cont-Multi, when a continuation of a multi-shot effect is called, we use the function *rem_locs* to remove the locations in the fibers θ_2, that is, *rem_locs* θ_2 contains all the same stack frames as θ_2, except the stack frames are empty. Because the stack frames are not removed, but emptied, the program $K[v]$ that the continuation resumes, can allocate into and end the regions it had already started, but it can not use the references that were already allocated in these regions. This models our approach to multi-shot continuations explained in Chapter 4.3.1.

4.4 Program Logic

In this section, we present the Yarrow Logic (YL), an Iris-based program logic built to reason about the operational semantics of Yarrow we presented in Chapter 4.3.2. YL inherits all the reasoning principles from Iris; we assume familiarity with separation logic and use this section to present the features that are unique to YL.

4.4.1 Overview

Modular reasoning about memory in the Yarrow programming language is different from other programming languages, as we have to (1) reclaim resources used to reason about memory in regions that are captured by continuations, and (2) take into account that the configurations of fibers and regions can change between suspending and resuming computations in effect handlers; an effect handler can potentially start and end regions, or install and uninstall other effect handlers, before using its continuation. Our choice of *logical resources* and the *effectfull weakest precondition* are key to how we handle this complexity.

Logical resources We use three types of logical resources. The first two are points-to resources used for the stack and the heap.

$\ell \hookrightarrow v$ heap location ℓ stores value v
(or alternatively a closure identifier, *i.e.,* $\ell \hookrightarrow id$)

$\ell \mapsto v$ stack location ℓ stores value v
(or alternatively a closure identifier, *i.e.,* $\ell \mapsto id$)

As we assume that the memory pointed to by heap references are freed by the garbage collector, YL, like most other Iris-based program logics, does not have to handle revocation of heap points-to resources explicitly; they can simply be dropped in proofs when they are no longer needed; this is always allowed in Iris since it is an *affine* separation logic, *i.e.,* we have $P * Q \vdash P$, for any P and Q. Stack points-to resources used for references allocated in regions are, however, different: when a region ends, or a continuation captures fibers when an effect is performed, we need to reclaim stack points-to resources as it is unsafe to keep using them according to the operational semantics. Therefore, the proofs in our logic use an additional resource, Stack κ, to keep track of the shape of the stack, *i.e.,* the state of the list of fibers that make up the stack. This allows us to always know which stack points-to resources to revoke when regions end or effects are performed. To formally describe the resource Stack κ we need the following additional definitions:

$$\pi \in \text{Domain} \triangleq \text{List } \mathit{Loc} \quad \tau \in \text{Domains} \triangleq \text{List Domain}$$

$$\kappa \in \text{FiberDomains} \triangleq \text{List (Operation} \times \text{Domains)}$$

Stack κ the currentstack is made of out fibers with the domains represented by κ

The Stack κ resource together with stack points-to resources for all the locations in κ gives us the full picture of the fibers that make up the stack, as FiberDomains only differ from physical Fibers except for the values stored in the fibers' locations which can be recovered from the stack points-to resources. As we will see later in this section, proofs of programs that do not use region-based memory management or effect handlers, can forget about this resource.

Effectful weakest precondition To prove safety of programs in YL, we use an effectful weakest precondition ewp $e\ {}_{\rho}\{\Phi\}$, where the effect row ρ represent the effects that are handled in the expression e, and the postcondition Φ is a predicate on the return value of the expression e. The type of the effect row is shown below and uses the notion of a *protocol*. Our protocols are based on a similar definition by De Vilhena and Pottier [2021], but are redesigned to take the fiber domains into account.

$$\Psi \in \text{Protocol} \triangleq \mathit{Val} \to \text{FiberDomains} \to (\mathit{Val} \to \text{FiberDomains} \to \mathit{iProp}) \to \mathit{iProp}$$

$$\rho \in \text{Row} \triangleq \text{Operation} \xrightarrow{\text{fin}} (\text{Affinity} \times \text{Protocol})$$

Protocols work as a contract between the program performing the effect and the effect handler. Our protocols are parameterized by the fiber domains used by the `Stack` resource since both the effect handler and the program performing the effect can use Stack κ for revocation and we need to know the state of κ they are handed by the other end. We delay explaining the details of protocols till we present the reasoning rules for performing and handling effects, as these are the only rules that use them.

The adequacy theorem Proofs in our logic start by assuming the resource Stack ($[(-, [[]])]$) and no points-to resources. This reflects that when a program starts executing, we install an initial fiber with one initial region, such that programs can make local allocations at the top level without having to explicitly start a region. Specifically, the adequacy theorem of YL used to prove safety and functional correctness of programs says: *For all expressions e and postcondition Φ , if* Stack $([(-, [[]])]) \vdash \text{ewp } e \ {}_{\rho}\{\Phi\}$ *is provable, e can always take a step in the operational semantics of Chapter 4.3.2 or it reduces to a value v for which $\Phi(v)$ holds.*

4.4.2 Reasoning Rules

We divide the reasoning rules of the program logic into four categories that we proceed to go over: (1) structural rules (2) rules for memory management (3) rules for performing effects and (4) rules for handling effects. In Chapter 4.4.3, we show how the reasoning rules can be used to verify an example.

Structural rules We display selected structural rules of the program logic in Figure 4.6. The rules reassemble those of other Iris-based separation logics, except we have to take into account the effect row in the effectful weakest precondition. For instance, in the monotonicity rules (EWP-Mono and EWP-Mono-Pers) and the frame rules (EWP-Frame and EWP-Frame-Pers), we use the persistently modality when the effect row ρ in ewp $v \ {}_{\rho}\{\Phi\}$ is not exclusively a row with one-shot effects; ρ lists the effects e can use and if e can perform an effect where the continuation in the effect handler is used multiple times (a multi-shot effect), then it is unsound to rely on exclusively owned resources when modifying the postcondition, intuitively since we are modifying the postcondition every time the continuation is used to resume computation in e. There are three structural rules involving effect rows: EWP-Row1 rows lets the user of the logic weaken the effect row (subset inclusion on effect rows $\rho_1 \subseteq \rho_2$ holds if every binding in ρ_1 occurs in ρ_2), EWP-Row2 is used for weakening a protocol (ordering of protocols is defined as $\Psi_1 \sqsubseteq \Psi_2 \triangleq \square(\forall v\ \kappa\ \Phi.\ \Psi_1\ v\ \kappa\ \Phi \twoheadrightarrow \Psi_2\ v\ \kappa\ \Phi)$) [5], and the order of effects

[5]For a proposition $\square P$, the persistently modality $\square$ enforces that P should be proven without any exclusive owned resources such as a point to resource.

$$\Phi\ v \vdash \text{ewp}\ v\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Val)}$$

$$\mathit{once}\ \rho \twoheadrightarrow (\forall v.\ \Phi\ v \twoheadrightarrow \Psi\ v) * \text{ewp}\ v\ {}_{\rho}\{\Phi\} \vdash \text{ewp}\ v\ {}_{\rho}\{\Psi\} \quad \text{(EWP-Mono)}$$

$$\Box(\forall v.\ \Phi\ v \twoheadrightarrow \Psi\ v) * \text{ewp}\ v\ {}_{\rho}\{\Phi\} \vdash \text{ewp}\ v\ {}_{\rho}\{\Psi\} \quad \text{(EWP-Mono-Pers)}$$

$$\text{ewp}\ e\ \{v.\ \text{ewp}\ N[v]\ {}_{\rho}\{\Phi\}\} \vdash \text{ewp}\ N[e]\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Bind)}$$

$$e_1 \rightsquigarrow_{\text{pure}} e_2 * \triangleright(\text{ewp}\ e_2\ {}_{\rho}\{\Phi\}) \vdash \text{ewp}\ e_1\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Pure)}$$

$$\mathit{once}\ \rho \twoheadrightarrow P \twoheadrightarrow \text{ewp}\ e\ {}_{\rho}\{\Phi\} \vdash \text{ewp}\ e\ {}_{\rho}\{v.\ \Phi\ v * P\} \quad \text{(EWP-Frame)}$$

$$\Box\, P \twoheadrightarrow \text{ewp}\ e\ {}_{\rho}\{\Phi\} \vdash \text{ewp}\ e\ {}_{\rho}\{v.\ \Phi\ v * P\} \quad \text{(EWP-Frame-Pers)}$$

$$\rho' \subseteq \rho \twoheadrightarrow \text{ewp}\ e\ {}_{\rho}\{\Phi\} \vdash \text{ewp}\ e\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Row1)}$$

$$\Psi' \sqsubseteq \Psi \twoheadrightarrow \text{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot\rho}\{\Phi\} \vdash \text{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot\rho}\{\Phi\} \quad \text{(EWP-Row2)}$$

$$\frac{op_1 \neq op_2 * \text{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot(op\ :\ (a,\Psi))\cdot\rho}\{\Phi\}}{\text{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot(op\ :\ (a,\Psi))\cdot\rho}\{\Phi\}} \quad \text{(EWP-Row3)}$$

Figure 4.6: Selected structural program logic rules.

that have different operations can be changed using EWP-Row3 ($\cdot$ is the cons operation on effect rows).

Rules for memory management Selected rules for memory management are shown in Figure 4.7. The rules for managing the heap (EWP-Alloc-Global and EWP-Store-Global) are standard for Iris-based program logics. The rules for managing fibers, and the regions we associate with stack frames in the fibers, centers around the Stack κ resource; EWP-Region inserts a new empty stack frame in the most recently installed fiber. A local allocation (EWP-Alloc-Local) inserts a new location, in the top most stack frame of the most recently installed fiber, and gives the user a stack points-to resource $\ell \mapsto v$. The rules for manipulating local references, *e.g.,* (EWP-Local-Local), work just like their counterpart rules for the heap, *i.e.,* owning a points-to resource for a location ℓ implies that the location belongs to a region that has not ended yet; what makes this possible is how deallocation of stack points-to resources work: when a region ends in the EWP-End rule, one has to provide the points-to resources for all the stack reference allocated in the region, and then, separately, prove (without those stack points-to resources) that the postcondition Φ holds for a stack without the topmost region. Lastly, we have allocation of functions (EWP-Func-Global) and application of functions (EWP-App-Local) which also interferes with memory management as function closures are explicitly allocated in our language. Allocation of function closures is much like allocation

$$\triangleright(\forall \ell.\ \ell \hookrightarrow v \mathrel{-\!\!*} \Phi\ (\mathbf{ha}\,\ell)) \vdash \text{ewp } \mathbf{ref\ global}\ v\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Alloc-Global)}$$

$$\frac{\begin{array}{c}\text{Stack } (\kappa +\!\!+ [(op, \tau +\!\!+ [\pi])]) * \\ \triangleright(\forall \ell.\ \text{Stack } (\kappa +\!\!+ [(op, \tau +\!\!+ [\pi +\!\!+ [(\ell, v)]])]) * \ell \mapsto v \mathrel{-\!\!*} \\ \Phi\ (\mathbf{sa}\,\ell))\end{array}}{\text{ewp } \mathbf{ref\ local}\ v\ {}_{\rho}\{\Phi\}} \quad \text{(EWP-Alloc-Local)}$$

$$\ell \hookrightarrow w * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} \Phi\ ()) \vdash \text{ewp } (\mathbf{ha}\,\ell) \leftarrow v\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Store-Global)}$$

$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} \Phi\ v) \vdash \text{ewp } !(\mathbf{sa}\,\ell)\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Load-Local)}$$

$$\frac{\begin{array}{c}\text{Stack } (\kappa +\!\!+ [(op, \tau)])* \\ \triangleright(\text{Stack } (\kappa +\!\!+ [(op, \tau +\!\!+ [])]) \mathrel{-\!\!*} \text{ewp } \mathbf{end}\ e\ {}_{\rho}\{\Phi\})\end{array}}{\text{ewp } \mathbf{region}\ e\ {}_{\rho}\{\Phi\}} \quad \text{(EWP-Region)}$$

$$\frac{\begin{array}{c}\text{Stack } (\kappa +\!\!+ [(op, \tau +\!\!+ [\pi])]) * \mathop{\Large *}_{(\ell, v) \in \pi} \ell \mapsto v * \\ \triangleright(\text{Stack } (\kappa +\!\!+ [(op, \tau)]) \mathrel{-\!\!*} \Phi\ v)\end{array}}{\text{ewp } \mathbf{end}\ v\ {}_{\rho}\{\Phi\}} \quad \text{(EWP-End)}$$

$$\triangleright(\forall \ell\ id.\ \ell \hookrightarrow id \mathrel{-\!\!*} \Phi\ (\lambda^{(\mathbf{ha}\,\ell, id)} f\ x.\ e)) \vdash \text{ewp } \lambda^{\mathbf{global}}\ f\ x.\ e\ {}_{\rho}\{\Phi\} \quad \text{(EWP-Func-Global)}$$

$$\frac{\ell \mapsto id * \triangleright(\ell \mapsto id \mathrel{-\!\!*} \text{ewp } e[(\lambda^{(\mathbf{sa}\,\ell, id)} f\ x.\ e)/f][v/x]\ {}_{\rho}\{\Phi\})}{\text{ewp } (\lambda^{(\mathbf{sa}\,\ell, id)} f\ x.\ e)\ v\ {}_{\rho}\{\Phi\}} \quad \text{(EWP-App-Local)}$$

Figure 4.7: Selected program logic rules for memory management.

of references: in EWP-Func-Global, the user of the logic obtains a heap points-to resource with a location ℓ pointing to an identifier id, the $(\mathbf{ha}\,\ell, id)$ pair is then used in the λ-abstraction value (allocation of local closures in region happens similarly, except the fibers are updated as in EWP-Alloc-Local). When applying a lambda abstraction, one must show that the function closure is not deallocated by proving ownership of the points-to resource associated with the λ-abstraction value (EWP-App-Local); for globally allocated function closures, this is easy as heap points-to resources are persistent, but stack points-to resource are deallocated when regions end as seen in the EWP-End rule.

Rules for performing effects Suppose we perform an effect with operation op as in the proof goal of the EWP-Do-Once rule in Figure 4.8, *i.e.,* we want to prove ewp $\mathbf{do}\ op\ v\ {}_{(op\ :\ (\mathbf{once}, \Psi)) \cdot \rho}\{\Phi\}$. Here, the effect row tells us that the effect we are performing is a one-shot effect, and that the contract we have with the effect handler is described by the protocol Ψ. First, we have to provide

the stack resource Stack $(\kappa_1 +\!\!+ [(op, \tau)] +\!\!+ \kappa_2)$ together with stack points-to resources for a map m: $\ast_{(\ell,v)\in m}\, \ell \mapsto v$. The predicate *isValid op* τ κ_2 m asserts that κ_2 does not contain a fiber with operation *op*, and the references in m are equal to those allocated in the fibers $([(op, \tau)] +\!\!+ \kappa_2)$. Intuitively, this means we have found the fiber of the effect we are performing, and at the same time, deallocated all the references that the continuation of the effect handler captures in its regions. The last obligation of the EWP-Do-Once rule, is to satisfy the protocol:

$$\Psi\ v\ \kappa_1\ (\lambda\ w\ \kappa_1'.\ \text{Stack}\ (\kappa_1' +\!\!+ [(op, \tau)] +\!\!+ \kappa_2) \mathrel{-\!\!\ast} \bigast_{(\ell,v)\in m} \ell \mapsto v \mathrel{-\!\!\ast} \Phi\ w)$$

We always instantiate a protocol with the argument value v provided when performing the effect, the part of the stack κ_1 that the effect handler is executing with, and a proof obligation that must be proven when resuming computation with a continuation using return value w and a potentially updated stack described by κ_1' (the stack can change if the effect handler starts or end regions and installs or uninstall other effect handlers). The proof obligation says that given the stack resource where the fibers that the continuation captured are back onto the stack Stack $(\kappa_1' +\!\!+ [(op, \tau)] +\!\!+ \kappa_2)$, and ownership of the stack points-to resources for the references in the regions of these fibers $\ast_{(\ell,v)\in m} \ell \mapsto v$, the post-condition of the proof goal we had when performing the effect $\Phi\ w$ must hold. The only difference between the EWP-Do-Once rule for one-shot effects and the EWP-Do-Many rule for multi-shot effects is that when proving the post-condition of the proof goal we had when performing the effect $\Phi\ w$, we can not assume stack points-to resources for the references in the regions of the fibers that the continuation capture.

Rules for handling effects To install an effect handler with allocation of the continuation closure on the heap, we use the EWP-Try rule in Figure 4.8 for both one-shot and multi-shot effects. In this rule, we must first prove that the affinity of the protocol Ψ matches the affinity variable a (we return to how this is done when we present an example protocol in Chapter 4.4.3) and provide the stack resource Stack κ. There is then two proof obligations: (1) prove the handler branch and the return branch are safe to execute; this is captured by the *hdlGlb* predicate whose definition we return to shortly, and (2) prove that the expression e for which the effect is installed is safe to execute; this is captured by the following proof goal:

$$\text{Stack}\ (\kappa +\!\!+ [(op, [])]) \mathrel{-\!\!\ast} \text{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot\rho}\{v.\ \exists\kappa'.\ \Phi_e\ \kappa'\ v \ast \text{Stack}\ (\kappa' +\!\!+ [(op, [])])\}$$

Here we get to assume the stack resource Stack $(\kappa +\!\!+ [(op, [])])$ with a fiber installed for the new effect. Moreover, the new effect is added to the effect row in the effectful weakest precondition with affinity a and protocol Ψ (it is here we see that the protocol is picked as part of the EWP-Try rule and then used

$$\frac{\begin{array}{c}\exists \kappa_1\ \kappa_2\ \tau\ m.\ \mathrm{Stack}\ (\kappa_1 +\!\!+ [(op,\tau)] +\!\!+ \kappa_2) * \\ \ast_{(\ell,v)\in m}\ \ell \mapsto v * \mathit{isValid}\ op\ \tau\ \kappa_2\ m * \\ \triangleright \Psi\ v\ \kappa_1\ (\lambda\ w\ \kappa_1'.\ \mathrm{Stack}\ (\kappa_1' +\!\!+ [(op,\tau)] +\!\!+ \kappa_2) \mathrel{-\!\!*} \\ \ast_{(\ell,v)\in m}\ \ell \mapsto v \mathrel{-\!\!*} \Phi\ w)\end{array}}{\mathrm{ewp}\ \mathbf{do}\ op\ v\ {}_{(op\ :\ (\mathbf{once},\Psi))\cdot\rho}\{\Phi\}} \quad \text{(EWP-Do-Once)}$$

$$\frac{\begin{array}{c}\exists \kappa_1\ \kappa_2\ \tau\ m.\ \mathrm{Stack}\ (\kappa_1 +\!\!+ [(op,\tau)] +\!\!+ \kappa_2) * \\ \ast_{(\ell,v)\in m}\ \ell \mapsto v * \mathit{isValid}\ op\ \tau\ \kappa_2\ m * \\ \triangleright \Psi\ v\ \kappa_1\ (\lambda\ w\ \kappa_1'.\ \mathrm{Stack}\ (\kappa_1' +\!\!+ \mathit{rem\_locs}\ ([(op,\tau)] +\!\!+ \kappa_2)) \\ \mathrel{-\!\!*} \Phi\ w)\end{array}}{\mathrm{ewp}\ \mathbf{do}\ op\ v\ {}_{(op\ :\ (\mathbf{many},\Psi))\cdot\rho}\{\Phi\}} \quad \text{(EWP-Do-Many)}$$

$$\frac{\begin{array}{l}\mathit{protAff}\ a\ \Psi * \mathrm{Stack}\ \kappa * \triangleright\, \mathit{hdlGlb}\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi * \\ \triangleright \big(\mathrm{Stack}\ (\kappa +\!\!+ [(op,[])]) \mathrel{-\!\!*} \\ \quad \mathrm{ewp}\ e\ {}_{(op\ :\ (a,\Psi))\cdot\rho}\{v.\ \exists \kappa'.\ \Phi_e\ \kappa'\ v * \mathrm{Stack}\ (\kappa' +\!\!+ [(op,[])])\}\big)\end{array}}{\mathrm{ewp}\ (\mathbf{try}\ (\mathbf{global}, a)\ e\ \mathbf{with}\ op\ x\ k.\ h \mid \mathbf{ret}\ y.\ r)\ {}_{\rho}\{\Phi\}} \quad \text{(EWP-Try)}$$

Figure 4.8: Selected program logic rules performing and handling effects.

as a contract for performing the effect in e, because the protocol Ψ is added to the effect row in the effectful weakest precondition). As postcondition we must prove a user-defined postcondition Φ_e, which we instantiate with the return value v and the updated fiber domains κ'. Moreover, we must also prove that the current fibers can be described by the stack resource Stack $(\kappa' +\!\!+ [(op,[])])$ which captures that the top most fiber is the one for the effect with operation op, but that all the fibers before that are now κ'. The reason why the fibers installed before the effect with operation op can change from κ to κ', is that inside e, effects in the effect row ρ may be performed, and the handlers for these effects can install other effects and, perhaps more likely, make changes to regions by *e.g.,* allocating local references.

Next, we turn our attention to the *hdlGlb* predicate which captures the correctness of the return branch r and the handler branch h. We show its definition in Figure 4.9. This predicate is divided into two parts, one part for the return branch and one part for the handler branch. Around the two branches, there is a persistently modality conditional on the affinity mode $m\rho$; when $m\rho$ is **once**, we can remove the persistently modality and use exclusively owned resources to prove correctness of the return and handler branche. When the persistently modality is removed, it is required that the existing effect row ρ can not have any multi-shot effects. This is because the program e that we install the effect for can also use effects from ρ. If e uses a multi-shot effect,

$$
\begin{aligned}
&\mathit{hdlGlb}\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi \triangleq m\rho = \mathbf{once} \to \mathit{once}\ \ \rho\ \wedge \Box ?m\rho\ \big(\\
&\quad (*\ \mathit{Return\ branch} : *)\\
&\quad (\forall v\ \kappa.\ \Phi_e\ \kappa\ v \mathrel{-\!\!*} \text{Stack}\ \ \kappa\ \mathrel{-\!\!*} \text{ewp}\ r[v/y]\ {}_\rho\{\Phi\}) \wedge\\
&\quad (*\ \mathit{Handler\ branch} : *)\\
&\quad (\forall v\ k'.\ (\exists \kappa.\ \text{Stack}\ \ \kappa * \Psi\ v\ \kappa\ (\lambda w\ \kappa'.\ \forall \Phi'.\ \text{Stack}\ \ \kappa' \mathrel{-\!\!*}\\
&\qquad\qquad \triangleright\, \mathit{hdlGlb}\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi' \mathrel{-\!\!*} \text{ewp}\ k'\ w\ {}_\rho\{\Phi'\}))\\
&\qquad \mathrel{-\!\!*} \text{ewp}\ h[k'/k][v/x]\ {}_\rho\{\Phi\})\big)
\end{aligned}
$$

Figure 4.9: Definition of the predicate *hdlGlb*.

then execution can enter the handler branch h and return branch r many times without using a continuation (we reason about the use of continuations inside the proof of the handler branch), thus we can not rely on exclusively owned resources when reasoning about h and r.

In the proof of the return branch r, we get to assume the postcondition $\Phi_e\ \kappa\ v$ of the expression e that the effect was installed for, where v is the return value of e and κ corresponds to the fiber domains when the execution of e ended, together with the stack resource $\mathit{Stack}\ \kappa$. In the proof of the handler branch h, k' is the continuation which, together with the argument v that was used to perform the effect, is substituted for the binders k and x. We can assume the stack resource $\mathit{Stack}\ \kappa$, and the protocol instantiated as below:

$$
\begin{aligned}
&\Psi\ v\ \kappa\ (\lambda w\ \kappa'.\ \forall \Phi'.\ \text{Stack}\ \ \kappa' \mathrel{-\!\!*} \triangleright\, \mathit{hdlGlb}\ a\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi' \mathrel{-\!\!*}\\
&\qquad\qquad \text{ewp}\ k'\ w\ {}_\rho\{\Phi'\})
\end{aligned}
$$

Here we see the opposite end of the contract that a protocol Ψ establishes between an effect handler and the program performing the effect (we saw the other end when we discussed the EWP-Do-Once and EWP-DO-Many rules). Note that in the handler branch we get to assume the protocol above, it is not a proof obligation. The predicate on the value w and the fiber configuration κ' that we instantiate the protocol with on this end, says that to get a proof of the effectful weakest precondition for the continuation k' that we can use in the proof of the handler branch, we must provide the stack resource Stack κ' and prove the recursive occurrence of the *hdlGlb* predicate. The reason we have to prove the recursive occurrence of the *hdlGlb* predicate is that the effect handler is reinstalled as a part of the continuation.

In the last few paragraphs, we went over the proof rules for installing an effect handler with the global mode **`global`**. If we want to use the local mode **`local`**, and install the continuation closure in the region that the effect handler uses, we need to use a version of EWP-Try with the only modification that *hdlGlb* is instead *hdlLoc*. We have included the definition of *hdlLoc* in Appendix 4.9; it works like *hdlGlb*, except each time the continuation is used in the handler branch, one must prove that the continuation closure is still

alive by showing ownership of a stack points resource (the resource is provided as part of *hdlLoc*) with the closure identifier that the continuation uses — like when applying a function in the EWP-App-Local rule.

4.4.3 Verification of the Increment First Example

In this subsection, we go over a proof sketch of the Increment First Example in Figure 4.3 with a focus on using the rules for performing and handling effects. Our goal is to prove Stack $([(-, [[]])]) \vdash$ ewp `hdl_ex2 ()` $\langle\rangle\{\mathsf{True}\}$, so we can use the adequacy theorem of YL from Chapter 4.4.1. We skip forward a bit to line 3 in the code just before the **`try`**-construct. At this point in the proof, we have the resources Stack $([(-, [[]; []; [\mathsf{x}]])])$ and $\mathsf{x} \mapsto 0$ reflecting that the regions of `hdl_ex2` and `ex2` are started, and in the region started in `ex2`, we have allocated the local reference `x` on line 2. As we need to reason about an effect handler, we use the EWP-Try rule. Using this rule, we must pick a protocol that captures the interaction between the effect handler and `ex2 x` which is the code the effect is installed for. Throughout this paper, we use a particular type of protocol, namely the ***send-receive protocol***, an extension of the *send-receive protocol* from De Vilhena and Pottier [2021] that also takes into account the fiber domains used with the `Stack` resource:

$$!\vec{x}(v_1, \kappa_1)\{P\}.a?\vec{y}(v_2, \kappa_2)\{Q\} \triangleq \lambda v\ \kappa\ \Phi.\ \exists \vec{x}.\ v = v_1 \land \kappa = \kappa_1 * P * \\ \square?a(\forall \vec{y}.\ Q \mathrel{-\!\!*} \Phi\ v_2\ \kappa_2).$$

When a send-receive protocol on the form above is used, it means that the value argument v must be equal to v_1 and that the fiber domains, when the fibers that the continuation capture are removed from the stack, must be equal to κ_1. The proposition P works as a precondition that the proof of a program performing the effect must prove, and the effect handler can assume when handling the effect. When the effect handler calls the continuation, it must call it with an argument equal to v_2, be in a state where the fiber domains is equal to κ_2 (κ_2 are the fiber domains before the fibers of the continuation are appended to the stack) and prove that Q holds. Thus, P and Q can be seen as precondition and postcondition of **`do`** *op* v. To make the send-receive protocol expressive, it existentially quantifiers over the list of variables $\vec{x}$ in v_1, κ_1, P, v_2, κ_2 and Q, and universally quantifiers over $\vec{y}$ in v_2, κ_2 and Q. The affinity variable a controls whether the send-receive protocol is for a one-shot or multi-shot effect; to control this we use the *persistently modality* $\square$. When the affinity variable a is **`many`**, $\square?a\ P$ is the same as $\square\, P$, whereas if a is **`once`**, the persistently modality is removed and $\square?a\ P$ becomes P. In essence, the persistently modality is conditional on a. The send-receive protocol, together with the composition protocol below that composes two existing protocols, are the only protocols we use in this paper:

$$\Psi_1 + \Psi_2 \triangleq \lambda v\ \kappa\ \Phi.\ \Psi_1\ v\ \kappa\ \Phi \lor \Psi_2\ v\ \kappa\ \Phi$$

Here is the concrete send-receive protocol we use in this proof with the EWP-Try rule:

$$INC \triangleq !\kappa\ \ell_1\ \ell_2\ z\ ((\mathbf{sa}\,\ell_1, \mathbf{sa}\,\ell_2), \kappa)\{\ell_1 \mapsto z\}.\mathbf{once}?((), \kappa)\{\ell_1 \mapsto (z+1)\}$$

The INC protocols says that the effect can be called with a pair of stack addresses $(\mathbf{sa}\,\ell_1, \mathbf{sa}\,\ell_2)$ as argument and any fiber domains κ in the `Stack` resource. For the first entry in the pair, the stack points-to resource $\ell_1 \mapsto z$ is required as precondition. When the effect resumes, it resumes with the unit value and the fiber domains unchanged. The postcondition gives back the stack points-to resource but the integer it points to is now incremented: $\ell_1 \mapsto (z+1)$.

Now that we have picked our protocol, we can proceed to prove the effect handler on line 3 in Figure 4.3 and the code `ex2 x` that the effect is installed for, as required by the EWP-Try rule. When a protocol Ψ is made using send-receive and composition protocols, showing the *protAff* a Ψ obligation in the EWP-Try rule amounts to checking whether the affinity variable used in send-receive protocols matches a. [6] In this example, we are installing the effect handler on line 3 with the **`once`** mode which matches the mode used in the INC protocol.

Proof of `ex2` The `ex2` function is used with the argument `x` for which we have the stack points-to resource $\mathsf{x} \mapsto 0$. Because Stack $([(-, [[]; []; [\mathsf{x}]])])$ is the state of the stack resource prior to using EWP-Try, we can also assume the stack resource Stack $([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [])])$, updated with a new fiber for the effect handler that we just installed. We pick $\Phi_e \triangleq \lambda\kappa\ \_.\ \kappa = [(-, [[]; []; [\mathsf{x}]])] * \mathsf{x} \mapsto 1$ in EWP-Try which makes our proof obligation this:

$$\begin{array}{l}\text{ewp } \texttt{ex2 x}\ {}_{(\texttt{IncFst} : (\mathbf{once},\ INC))} \\ \quad \{\text{Stack } ([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [])]) * \mathsf{x} \mapsto 1\}\end{array}$$

Let us skip ahead in the proof to where the effect is performed. Here, we have the resources Stack $([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [[y]])])$, $\mathsf{x} \mapsto 0$ and $\mathsf{y} \mapsto 1$, which reflects that the region of `ex2` started and the reference `y` is allocated in it. The proof obligation at this point is:

$$\begin{array}{l}\text{ewp } \texttt{do IncFst (x, y)}\ {}_{(\texttt{IncFst} : (\mathbf{once},\ INC))} \\ \quad \{v.\ \text{ewp } \mathbf{end}\ (v\texttt{;assert (!x + !y = 2)})_{(\texttt{IncFst} : (\mathbf{once},\ INC))} \\ \qquad \{\text{Stack } ([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [])]) * \mathsf{x} \mapsto 0\}\}\end{array}$$

Because the effect we want to perform is a one-shot effect, as seen in the effect row of our proof obligation, we use the EWP-Once rule. This rule says that we must provide the stack points-to resources for all locations of the regions in

[6] In the Rocq formalization, we use *upward closures* of protocols as in de Vilhena [2022], but the protocols used in this paper are equivalent to their upward closures when the *protAff* property holds.

the fibers up to and including the fiber for the effect we perform, as these are the fibers captured by the continuation. In our case, this amounts to the fiber (`IncFst`, [[y]]) with one region containing `y` which we provide the stack points-to resource $\mathsf{y} \mapsto 1$ for. The fiber $[(-, [[]; []; [\mathsf{x}]])$ prior to the (`IncFst`, [[y]]) fiber, is what the effect handler gets to use, but the protocol INC makes no assumption about the contents of this fiber. As we are using `(x, y)` as argument to the effect, the protocol INC says that we must provide $\mathsf{x} \mapsto 0$ to the effect handler. The postcondition of the protocol gives us back the stack points-to resource incremented ($\mathsf{x} \mapsto 1$), and the fiber prior to the `IncFst` fiber is left unchanged. When proving the postcondition of our proof obligation, we also get to assume Stack $([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [[\mathsf{y}]])])$; because we were performing a one-shot effect, `y` is still in region of the top most fiber, and we get back the points-to resource $\mathsf{y} \mapsto 1$. With these resources in hand, we can prove the postcondition of the proof obligation we listed above by satisfying the assertion and ending the top most regions with `y` in it, which consumes the $\mathsf{y} \mapsto 1$ resource and leaves us with the Stack $([(-, [[]; []; [\mathsf{x}]]); (\texttt{IncFst}, [])]) * \mathsf{x} \mapsto 1$ resources as needed.

Proof of the effect handler In the proof obligation about the effect handler in the EWP-Try rule, we instantiate the *hdlGlb* predicate which amounts to two proof obligations, one for the handler branch and one for the return branch. The obligation for the return branch is shown below and follows from the EWP-End rule:

$$\text{Stack } [(-, [[]; []; [\mathsf{x}]])] * \mathsf{x} \mapsto 1 \vdash \text{ewp } \textbf{end } (\textbf{end } v) \ {}_{\langle\rangle}\{\mathsf{True}\}$$

The resources $Stack[(-, [[]; []; [x]])]$ and $\mathsf{x} \mapsto 1$ are what `example_2 x` that we installed the effect for return. Thus the above goal completes the proof obligations about the return branch and the code surrounding the effect handler which amounts to ending the two regions started in `hdl_inc_fst` and `hdl_ex2`. For the handler branch, the proof obligation is:

$$\begin{array}{l}\text{Stack } \kappa * \mathsf{x} \mapsto 0 \vdash \\ \text{ewp } \texttt{((x, y).1 <- !(x, y).1 + 1); k ()} \ {}_{\langle\rangle}\{\text{ewp } \textbf{end } (\textbf{end } v) \ {}_{\langle\rangle}\{\mathsf{True}\}\}\end{array}$$

It is the INC protocol that dictates we get to assume $\mathsf{x} \mapsto 0$ and Stack κ for some fiber domains κ. Further, the protocol gives us the specification for calling the continuation in the *hdlGlb* predicate, which says that if we can prove the recursive occurrence of the *hdlGlb* predicate, we get to assume:

$$\text{Stack } \kappa * \mathsf{x} \mapsto 1 \vdash \text{ewp } \texttt{k ()} \ {}_{\langle\rangle}\{\text{ewp } \textbf{end } (\textbf{end } v) \ {}_{\langle\rangle}\{\mathsf{True}\}\}$$

The specification captures that calling the continuation with the effect handler reinstalled (the effect handler is reinstalled as part of `k`), the postcondition for the return branch that we already proved holds (the return branch is used because the effect handler is reinstalled), as long as we can prove the precondition of the continuation Stack $\kappa * \mathsf{x} \mapsto 1 \vdash$ as specified by the INC protocol. The

proof obligation for the handler branch follows from the specification of the continuation above, as the increment of `x` turns $\texttt{x} \mapsto 0$ into $\texttt{x} \mapsto 1$, leaving us with the exact resources and proof goal needed. The only missing obligation is to prove the recursive occurrence of the *hdlGlb* predicate, but this follow from Löb induction ($(\triangleright P \Rightarrow P) \vdash P$) applied at the beginning of the proof of the effect handler.

4.5 Case Study: LIFO data structure

In this section, we show how an effect-handler implementation of a data structure can be made with local allocations. We focus on a LIFO data structure, *i.e.,* a stack, but the approach generalizes to other data structure implementations. The code of the LIFO data structure implementation is shown in Figure 4.10, where `hdl_lifo` is the implementation of the data structure. Our approach to implementing data structures using local allocations in regions is to make all allocations in the region of the effect handler; this also includes the continuations' closure. In the implementation of the `LIFO` effect, we use a local reference `r` that always points-to the element that was last inserted. The LIFO data structure is implemented as a linked-list of local references and has two operations, an `Insert` and a `Remove` operation defined using sum types. To handle the `Insert` operation with argument `x` (`x` is the value we want to insert into the data structure), we load the head of the list from the `r` reference, and then update `r` with an optional of a newly allocated local reference; the reference points to a pair of the value `x` and the previous head of the list `hd`. The protocol for the `LIFO` effect that we define shortly makes sure that we never try to use the `Insert` operation on the empty data structure, hence we never assert `false` on line 9. Thus, when the `Remove` operation is used, we can update `r` by setting it to the next element in the linked-list that makes up the LIFO data structure. Below is the `LIFO` effect protocol that we have proven the implementation against:

$$
\begin{aligned}
&\mathit{LIFO}\ \mathit{op}\ \tau\ \mathit{lifo} \triangleq \\
&\qquad !\kappa\ \pi_1\ x\ xs\ (\texttt{Insert}\ x, \kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_1])])\{\mathit{lifo}\ xs\ \pi_1\}.\textbf{once} \\
&\qquad ?\pi_2\ ((), \kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_2])])\{\mathit{lifo}\ (x :: xs)\ \pi_2\} \\
&\quad + !\kappa\ \pi_1\ x\ xs\ (\texttt{Remove}, \kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_1])])\{\mathit{lifo}\ (x :: xs)\ \pi_1\}.\textbf{once} \\
&\qquad ?\pi_2\ (x, \kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_2])])\{\mathit{lifo}\ xs\ \tau_2\}
\end{aligned}
$$

Central to the specification is a predicate *lifo* $xs\ \pi$ that keeps track of the contents of the data structure using a list xs, and a stack domain π which corresponds to the region of the effect handler.[7] The list xs in the predicate changes as one would expect from using the `Insert` and `Remove` operations. The

[7] In the Rocq formalization, we have a lemma to exchange between *lifo* $xs\ \pi$ and stack points-to resources that correspond to the references in π, these resources can be used to prove the *isValid* predicate in the EWP-DO rules when performing other effects. Similar

stack domain π also changes, during the operation, from π_1 when performing the `LIFO` effect to some other stack domain π_2 when resuming computation. We see that π_1 does in fact correspond to the top most stack frame when performing the effect, as the protocol asserts that the fiber domains in which the effect handler gets to execute are $\kappa +\!\!+ [(op, \tau +\!\!+ [\pi_1])]$ (remember these are the fiber domains after the fibers of continuation has been captured). Likewise, the effect handler's stack frame has changed to π_2 during the execution of the effect handler, as the protocol states that the fiber domains are $\kappa +\!\!+ [(op, \tau +\!\!+ [\pi_1])]$ when resuming computation (this is the state before the fibers that the continuation capture are reinserted). This models that the implementation of the specification can use the region associated with the top most stack frame to make local allocations during the execution of the operations that the effect exposes; the fiber for the operation *op* is some arbitrary fiber where the `LIFO` effect handler is installed.

```
1  let hdl_lifo f = region (
2    let r = ref local None in
3    try (local, once) f () with
4    | effect LIFO arg k ->
5      (match arg with
6       | Insert x -> let hd = !r in r <- (Some (ref local (x, hd))); k ()
7       | Remove ->
8         (match !r with
9          | None -> assert false
10         | Some r -> let p = !r in r <- p.2; k p.1))
11   | ret x -> x)
12
13 let ex3 () = region (
14   do LIFO (Insert 1); do LIFO (Insert 2); do LIFO (Insert 3);
15   assert (do LIFO Remove = 3); assert (do LIFO Remove = 2);
16   assert (do LIFO Remove = 1))
17
18 let hdl_ex3 () = region (hdl_lifo ex3)
```

Figure 4.10: LIFO data structure implementation.

4.6 Case study: Checkpointing

In this section, we use an effect handler to implemented locally allocated checkpoints; checkpoints are continuations that we store in the region where the effect handler is installed together with the closure of the continuation. The implementation of the `Checkpoint` effect is shown in Figure 4.11, and it is inspired by a similar example by Muhcu et al. [2025]. The `Checkpoint` effect has two operations: a `Save` operation to save a checkpoint at the current line in the code, and a `Retry` operation to resume computation from the last saved checkpoint. In the definition of `ex4` on line 8, we see how a checkpoint is saved.

lemmas exist for the definitions used in the protocols of the other case studies that mention stack domains or fiber domains.

The branching on line 10, based on the value of the local reference `r` allocated in the region of `hdl_ex`, goes to the `if`-branch as `r` points to `0` initially. In the `if`-branch, `r` is incremented where after we retry computation from the checkpoint that was saved on line 9 such we never have to assert `false`. Now that `r` is no longer `0`, we go to the else branch where the assertion holds. The implementation of the `Checkpoint` effect on line 3 in Figure 4.11 uses a local reference to store checkpoints in form of continuations. Continuations are locally allocated in the region of the `hdl_checkpoint` function. `Checkpoint` is a multi-shot effect, because when a checkpoint is saved, we store the continuation, such that it can be called later when the retry operation of the effect is performed, but we also use the continuation to resume computation immediately.

Below, we specify the `Checkpoint` effect with a protocol:

$$
\begin{aligned}
&\mathit{CHECKPOINT}\ \mathit{op}\ \tau\ \mathit{cp}\ \mathit{noCp}\ P_1\ P_2 \triangleq \\
&\ !\kappa_1\ \pi_1\ (\mathsf{Save}, \kappa_1 +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_1])])\{(\mathit{noCp}\ \pi_1 \vee \mathit{cp}\ \pi_1) * P_1\ (\kappa_1 +\!\!+ [(\mathit{op}, \tau)])\}.\mathbf{many} \\
&\ ?\kappa_2\ \pi_2\ ((), \kappa_2 +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi_2])])\{\mathit{cp}\ \pi_2 * (P_1\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)]) \vee P_2\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)]))\} \\
&\ + \\
&\ !\kappa\ \pi\ (\mathsf{Retry}, \kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi])])\{\mathit{cp}\ \pi_1 * P_2\ (\kappa +\!\!+ [(\mathit{op}, \tau)])\}.\mathbf{many}?\ ((), [])\{\mathsf{False}\}
\end{aligned}
$$

The protocol uses a number of predicates: *cp* and *noCp* are defined as part of the proof of the effect handler, whereas P_1 and P_2, which we will refer to as user defined, are defined as part of the proof of the program that uses the effect (*c.f.* the proof structure in Chapter 4.4.3). Ownership of $\mathit{cp}\ \pi$ signifies that a checkpoint is installed, and $\mathit{noCp}\ \pi$ that no checkpoint is installed; these predicates are parametrized over the stack frame of the `Checkpoint` effect handler as this contains the region used to store checkpoints. To save a checkpoint, we must provide either $\mathit{noCp}\ \pi_1$ or $\mathit{cp}\ \pi_1$ as precondition together with the user defined predicate $P_1\ (\kappa_1 +\!\!+ [(\mathit{op}, \tau)])$ (the user-defined predicates are parameterized over all stack frames, except the effect handler's, to allow the user to make assertions about stack points-to resources in regions). In the postcondition of the `Save` operation, we can assert that our new checkpoint is installed in form of ownership of the predicate $\mathit{cp}\ \pi_2$ for a new stack frame π_2. We also gain $P_1\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)])$ or $P_2\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)])$. $P_1\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)])$ is returned when computation is resumed just after saving the checkpoint, but we also have to show that we can resume the computation using the continuation of the `Save` operation when done through the `Retry` operation; in the protocol above, the specification for the `Retry` operation states that the predicate P_2 must be provided in the precondition. As computation is never resumed using the continuation of a `Retry` operation, the postcondition is simply false. Instead, computation is resumed using the saved checkpoint and ownership of $P_2\ (\kappa_2 +\!\!+ [(\mathit{op}, \tau)])$ as shown in the postcondition of the `Save` operation. When proving safety of `ex4` in Figure 4.11, P_1 is instantiated with the stack points-to resource of `r` pointing to 0 ($r \mapsto 0$), and P_2 with the resource pointing to 1 ($r \mapsto 1$).

```
1  let hdl_checkpoint f = region (
2    let r = ref local () in
3    try (local, many) f () with
4    | effect Checkpoint arg k ->
5      (match arg with | Save -> r <- k; k () | Retry -> (!r) ())
6    | ret x -> x)
7
8  let ex4 r () = region (
9    do Checkpoint Save;
10   if (!r = 0) then
11     (r <- (!r + 1); do Checkpoint Retry; assert false)
12   else
13     (assert (!r = 1)))
14
15 let hdl_ex4 () = region (
16   let r = ref local 0 in hdl_checkpoint (ex4 r))
```

Figure 4.11: Checkpoint Implementation.

4.7 Case Study: Asynchronous Computation

We have taken the asynchronous computation implementation by De Vilhena and Pottier [2021], which is originally based on an implementation by Dolan et al. [2017], and turned global allocations into local allocations by utilizing the flexibility of the **region**-construct; asynchronous computation is based on recursion and tail-call optimization, but by omitting the creation of a region around the recursive function, we can make local allocations into the region of the enclosing parent function. Thus, in this case study, we break the association of a region per function body. [8]

The asynchronous computation library is implemented using an effect handler, see Figure 4.12. We focus on the parts of the implementation that we have changed to enable local allocations, for a detailed explanation of all the code, we refer to the aforementioned papers. Asynchronous computation provides two operations: `Async` e for asynchronously computing a task represented by the expression e. The `Async` e operation returns a promise p that a user thread can wait for the completion of using the `Await` p operation. The `Await` p operation returns a value, this is the return value of the task that created the promise p. The implementation in Figure 4.12 is based on a function `fulfill`, this function takes a task e and the promise p that the task should fulfill. Promises are implemented as dynamically allocated references pointing to a list of waiting user threads, in our version the references are locally allocated. The region that we use to allocate new promises in, for every invocation of the `Async` e operation, is not the region of the `fullfill` function that creates promises on line 12; the `fullfill` function is recursively called each time an asynchronous computation is made (line 13), thus if we want to utilize tail call optimization, we can not rely on the region of `fullfill`. Luckily, the **region**-construct is

[8] As mentioned in a previous footnote, OxCaml also features special keywords provided whereby the programmer can have finer control over the allocation

flexible in that we can allocate into the region of the parent function, *i.e.,* `run`, by omitting creation of a region in `fullfill`. Promises are not the only things we allocate into the region of `run`: all the non-empty function closures, on line 7, 13 and 22 use this region too.[9]

```
1  let new_promise () = ref local (Waiting (list_nil ()))
2
3  let next q = if queue_empty q then () else (queue_pop q) ()
4
5  let run main = region (
6    let q = queue_create () in
7    let fulfill = (fun local p e =>
8      try (local, once) e () with
9      | effect AC request k ->
10       (match request with
11        | Async e ->
12          let p = new_promise () in
13          queue_push q (fun local () => k p); fullfill p e
14        | Await p ->
15          (match !p with
16           | Done v -> k v
17           | Waiting ks -> p <- (Waiting (list_cons k ks)); next q))
18     | ret x ->
19       (match !p with
20        | Done v -> assert false
21        | Waiting ks ->
22          list_iter (fun local k => queue_push q (fun local () => k v)) ks;
            p <- (Done v); next q)
23   ) in
24   let p = new_promise () in
25   fullfill p main)
```

Figure 4.12: Asynchronous Computation.

Below, we show the protocol for the asynchronous computation effect AC:

$$
\begin{aligned}
\mathit{task}\ e\ P\ \Phi \triangleq \forall \kappa_1.\ P \mathrel{-\!\!*} \mathrm{Stack}\ (\kappa_1 +\!\!+ [(\mathsf{AC}, [])]) \mathrel{-\!\!*} \mathit{sched}\ \kappa_1 \mathrel{-\!\!*} \\
\quad \mathrm{ewp}\ \mathtt{e\ ()}\ {}^{(\mathsf{AC}\ :\ (\textbf{once},\ AC))} \\
\quad \{v.\ \square\,\Phi\ v * \exists \kappa_2, \mathit{sched}\ \kappa_2 * \mathrm{Stack}\ (\kappa_2 +\!\!+ [(\mathsf{AC}, [])])\} \\
AC \triangleq\ \ !e\ \Phi\ \kappa_1\ P\ (\mathtt{Async}\ e, \kappa_1)\{P * \mathit{sched}\ \kappa_1 * \triangleright\, \mathit{task}\ e\ P\ \Phi\}.\textbf{once} \\
\quad ?p\ \kappa_2 (p, \kappa_2)\{\mathit{isPromise}\ p\ \Phi * \mathit{sched}\ \kappa_2\} \\
\quad + !p\ \Phi\ \kappa_1 (\mathtt{Await}\ p, \kappa_1)\{\mathit{isPromise}\ p\ \Phi * \mathit{sched}\ \kappa_1\}.\textbf{once} \\
\quad ?y\ \kappa_2\ (y, \kappa_2)\{\square\,\Phi\ y * \mathit{sched}\ \kappa_2\}
\end{aligned}
$$

The protocol very closely resembles the one used by De Vilhena and Pottier [2021], the differences are best seen in the *task* predicate: each new task gets to use the Stack resource such that user threads can create their own regions, and then we have a predicate *sched* κ. The *sched* κ predicate is abstract to user threads, and defined as part of the proof of the effect handler, we use it to tie

[9] We use the notation "**fun** l `arg => e`" when we want to control where a function closure is allocated using the locality mode l.

the internal state of the effect handler implementation together with allocations in the region of the fibers described by fiber domains κ. The postcondition of a task must show ownership of the stack resource Stack and the *sched* predicate for some fiber domains κ_2 but, importantly, the return value must also satisfy the predicate Φ. The predicate Φ is a persistent predicate that a new task promises to fulfill when the task is created using the `Async` operation. The `Async` operation returns a promise p and another persistent predicate $isPromise\ p\ \Phi$ stating that p promises to fullfill the proof obligation described by Φ. In the specification of the `Await` p operation, we use $isPromise\ p\ \Phi$ as precondition, and when the promise is fulfilled, we get to assume $\Phi\ y$ where y corresponds to the return value of the task fulfilling the promise p.

4.8 Conclusion, Related Work and Future Work

In this paper we introduced Yarrow, an ML-like language with algebraic effects and region-based memory management, and YL, a separation logic for modular reasoning about programs written in Yarrow. We conducted a range of case studies showing that programs using one-shot and multi-shot effect can avoid garbage collection by using region-based memory management. We have already discussed related work closely along the way in the paper, here we discuss some additional related work and point to future work.

Runtimes for effect handlers A common way of implementing effect handlers is using continuation-passing style (CPS) [Hillerström et al., 2020; Leijen, 2017]. This can be done in intermediate representation languages without access to an explicit stack making it more widely applicable to different programming languages. In CPS, continuations are allocated as heap objects that contain enough information to resume computation. This is contrary to the approach by Sivaramakrishnan et al. [2021] that we build upon in this paper where capturing continuations amounts to saving a pointer to a fiber; the closures we associate with continuations are like normal function closure, they do not contain all contents of the stack necessary to resume computation as in CPS. The Flix programming language [Lutze et al., 2023; Madsen, 2022] implements effect handlers on top of the Java Virtual Machine (JVM), and the language has a region construct. The combination of regions and effect handlers is undefined behavior [Madsen, 2026], and their region-based memory management has not been treated in any papers. As mentioned in Chapter 4.3.1, Muhcu et al. [2025] recently proposed a new runtime design for a language with effect handlers (one-shot and multi-shot effects) and stack references based on the Effekt language [Brachthäuser et al., 2020a]. In this design stack references can be used even after performing multi-shot continuations by using an *indirection layer*; stack pointers are accessed through indirection pointers. A stack reference points to a pointer in the indirection layer which again points

to the actual stack. This only works for multi-shot effects as long as two continuations that capture the same stack segment are never used at the same time, as the case is with reentrant continuations. The paper does not come with a program logic or type system for their runtime; we suspect that creating a program logic or type system that prohibits reentrant continuations would be a significant challenge.

For future work, it is interesting to create a prototype runtime implementation for Yarrow and experimentally compare the runtime of programs using effect handlers with and without region-based memory management. The runtime could be complemented with inference of which regions have bounded size and can thus be allocated on the call stack in the style of Birkedal et al. [1996].

Relational logics and semantic typing Many Iris-based logics have been created to establish relational properties about programs such as *contextual equivalence*, see *e.g.,* [Timany et al., 2024c]. Informally, two programs are contextually equivalent if in all contexts they have the same observable behavior. The main application of relational logics and contextual equivalence is to show that an ideal implementation, serving as the specification, is equivalent to an efficient algorithm, serving as the implementation, e.g. as seen in the works of Vindum and Birkedal [2021] or Gäher et al. [2022]. Recently de Vilhena et al. [2026] made a relational logic to establish equivalences between programs with effect handlers (they do not prove contextual equivalence). In the future, we could extend YL to a relational logic. The extension can be used to establish equivalence between complex implementations, optimized w.r.t efficiency by using effect handlers and region-based memory management, and counterparts using simpler programming features. Another direction of future work, is to build a type system for Yarrow using *semantic typing* [Timany et al., 2024c]. Our logic can serve as a foundation in the same way that the original work on effect handlers in separation logic by De Vilhena and Pottier [2021] was used to create a typesystem for effect handlers with affine types [van Rooij and Krebbers, 2025], and a typesystem for effect handlers with dynamic labels [de Vilhena and Pottier, 2023]

4.9 Appendix: Condition for effect handlers with locally allocated continuation closures

$$
\begin{array}{l}
\mathit{hdlLoc}\ a\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi \triangleq \\
\quad m\rho = \mathbf{once} \to \mathit{once}\ \ \rho\ \wedge \\
\quad \Box?m\rho\big((*\ \mathit{Return\ branch}\ *) \\
\qquad (\forall v\ \kappa.\ \Phi_e\ \kappa\ v \mathrel{-\!\!*} \text{Stack}\ \ \kappa\ \ \mathrel{-\!\!*} \text{ewp}\ r[v/y]\ {}_\rho\{\Phi\})\ \wedge \\
\qquad (*\ \mathit{Effect\ branch}\ *) \\
\qquad (\forall v\ k'.\ (\exists \kappa\ \mathit{op}\ \tau\ \pi\ \ell\ \mathit{id}\ P\ \Phi_2\ \rho'\ K. \\
\qquad\qquad \text{Stack}\ \ (\kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi +\!\!+ [\ell]])]) * \ell \mapsto \mathit{id} * \\
\qquad\qquad \Psi\ v\ (\kappa +\!\!+ [(\mathit{op}, \tau +\!\!+ [\pi])])\ (\lambda w_2\ \kappa_2.\ P\ \kappa_2 \mathrel{-\!\!*} \text{ewp}\ K[w_2]\ {}_\rho\{\Phi_2\}) * \\
\qquad\qquad \Box?a(\forall w_1\ \kappa_1\ \Phi_1.\ \ell \mapsto \mathit{id} \mathrel{-\!\!*} \text{Stack}\ \ \kappa_1 \mathrel{-\!\!*} \\
\qquad\qquad\qquad \mathit{hdlLoc}\ a\ \Psi\ \Phi_e\ x\ k\ h\ y\ r\ \rho\ m\rho\ \Phi_1 \mathrel{-\!\!*} \\
\qquad\qquad\qquad \triangleright(\ell \mapsto \mathit{id} \mathrel{-\!\!*} P\ \kappa_1 \mathrel{-\!\!*} \text{ewp}\ K[w_1]\ {}_\rho\{\Phi_2\}) \mathrel{-\!\!*} \\
\qquad\qquad\qquad \text{ewp}\ k'\ w_1\ {}_\rho\{\Phi_1\})) \mathrel{-\!\!*} \\
\qquad\quad \text{ewp}\ h[k'/k][v/x]\ {}_\rho\{\Phi\})\big)
\end{array}
$$

Figure 4.13: Definition of the predicate *hdlLoc*.

Bibliography

A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, USA, 1999.

A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 67–78, 2000. doi: 10.1109/ICDE.2000.839388.

Clément Allain and Gabriel Scherer. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *Proceedings of the ACM on Programming Languages*, 10(POPL):1702–1729, 2026.

Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. Reasoning about weak isolation levels in separation logic. *Proc. ACM Program. Lang.*, 9(ICFP), August 2025. doi: 10.1145/3747515. URL `https://doi.org/10.1145/3747515`.

Aneris-Team, March 2023. URL `https://github.com/logsem/aneris/blob/master/documentation.pdf`.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, page 309–318, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320658. doi: 10.1145/2484239.2484267. URL `https://doi.org/10.1145/2484239.2484267`.

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.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1): 108–123, 2015.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24

(2):1–10, may 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL https://doi.org/10.1145/568271.223785.

Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation log. 2017.

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183, 1996.

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473586. URL https://doi.org/10.1145/3473586.

Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360591. URL https://doi.org/10.1145/3360591.

Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. MonkeyDB: Effectively testing correctness under weak isolation levels. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485546. URL https://doi.org/10.1145/3485546.

Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. Dynamic partial order reduction for checking correctness against transaction isolation levels. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591243. URL https://doi.org/10.1145/3591243.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020a.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type-and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming*, 30:e8, 2020b.

Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, page 67–86, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642288685. doi: 10.1007/978-3-642-28869-2_4. URL https://doi.org/10.1007/978-3-642-28869-2_4.

Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *J. ACM*, 65(2), January 2018. ISSN 0004-5411. doi: 10.1145/3152396. URL https://doi.org/10.1145/3152396.

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015)*, pages 58–71. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015a.

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015b. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-939897-91-0. doi: 10.4230/LIPIcs.CONCUR.2015.58. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2015.58.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359632. URL https://doi.org/10.1145/3341301.3359632.

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: A verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/chajed.

Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/chajed.

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936165. doi: 10.1145/1281100.1281103. URL https://doi.org/10.1145/1281100.1281103.

Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 871–886, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/chang.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming*, 30:e9, 2020.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL https://doi.org/10.1145/2491245.

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349925. doi: 10.1145/3087801.3087802. URL https://doi.org/10.1145/3087801.3087802.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

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.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 715–726. VLDB Endowment, 2006.

Paulo Emílio de Vilhena. *Proof of Programs with Effect Handlers*. Theses, Université Paris Cité, December 2022. URL `https://inria.hal.science/tel-03891381`.

Paulo Emílio De Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL): 1–28, 2021.

Paulo Emílio de Vilhena and François Pottier. A type system for effect handlers and dynamic labels. In *European Symposium on Programming*, pages 225–252. Springer, 2023.

Paulo Emílio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. A relational separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 10(POPL):981–1009, 2026.

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.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, volume 13, 2015.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming*, pages 98–117. Springer, 2017.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19 (11):624–633, November 1976. ISSN 0001-0782. doi: 10.1145/360363.360369. URL `https://doi.org/10.1145/360363.360369`.

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: A separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi: 10.1145/3498689. URL `https://doi.org/10.1145/3498689`.

Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 249–259, New York, NY, USA, 1987. Association for

Computing Machinery. ISBN 0897912365. doi: 10.1145/38713.38742. URL https://doi.org/10.1145/38713.38742.

Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. Data race freedom à la mode. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704859. URL https://doi.org/10.1145/3704859.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2):51–59, jun 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. Distributed causal memory: Modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi: 10.1145/3434323. URL https://doi.org/10.1145/3434323.

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. Verifying reliable network components in a distributed separation logic with dependent separation protocols. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi: 10.1145/3607859. URL https://doi.org/10.1145/3607859.

Jim Gray. Notes on data base operating systems. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors, *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978. doi: 10.1007/3-540-08755-9_9.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37(5):282–293, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512563. URL https://doi.org/10.1145/543552.512563.

Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. IsoVista: Black-box checking database isolation guarantees. *Proceedings of the VLDB Endowment*, 17(12):4325–4328, 2024.

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.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12

(3):463–492, jul 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.

Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, pages 15–27, 2016.

Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30:e5, 2020.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, February 2023. ISSN 2150-8097. doi: 10.14778/3583140.3583145. URL https://doi.org/10.14778/3583140.3583145.

Damien Imbs and Michel Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, 2012.

Iris-Team, July 2026. URL https://gitlab.mpi-sws.org/iris/iris/.

Jepsen. Jepsen analyses, 2025. URL https://jepsen.io/analyses. Accessed 2025-02-21.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, page 275–288, USA, 2002. USENIX Association. ISBN 1880446006.

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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650, 2015. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980.

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, Nara, Japan, September*

18-22, 2016, pages 256–269, 2016. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

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

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.17. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.17.

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: compositional reasoning and inference for weak isolation. *Proceedings of the ACM on Programming Languages*, 2(POPL): 1–34, December 2017. ISSN 2475-1421. doi: 10.1145/3158115. URL http://dx.doi.org/10.1145/3158115.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.

Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. Concurrency control for database theorists. *SIGMOD Rec.*, 51(4):6–17, jan 2023. ISSN 0163-5808. doi: 10.1145/3582302.3582304. URL https://doi.org/10.1145/3582302.3582304.

Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, November 2020. ISSN 2150-8097. doi: 10.14778/3430915.3430918. URL https://doi.org/10.14778/3430915.3430918.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 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 696–723, 2017. doi: 10.1007/978-3-662-54434-1_26. URL https://doi.org/10.1007/978-3-662-54434-1_26.

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 218–231, 2017. doi: 10.1145/3009837.3009877. URL https://doi.org/10.1145/3009837.3009877.

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, pages 336–365, 2020. doi: 10.1007/978-3-030-44914-8_13. URL https://doi.org/10.1007/978-3-030-44914-8_13.

Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer, 1992. doi: 10.1007/3-540-57318-6_25.

Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. 1981. URL https://api.semanticscholar.org/CorpusID:263889622.

Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. Iris-MSWasm: Elucidating and mechanising the security invariants of memory-safe WebAssembly. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):304–332, 2024.

Daan Leijen. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061*, 2014.

Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 486–499, 2017.

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, 2012.

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: Verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022a. doi: 10.1145/3527324. URL https://doi.org/10.1145/3527324.

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, 2022b.

Si Liu, Long Gu, Hengfeng Wei, and David Basin. Plume: Efficient and complete black-box checking of weak isolation levels. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689742. URL https://doi.org/10.1145/3689742.

Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. Oxidizing OCaml with modal memory management. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi: 10.1145/3674642. URL https://doi.org/10.1145/3674642.

Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. With or without you: Programming with effect exclusion. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023. doi: 10.1145/3607846. URL https://doi.org/10.1145/3607846.

Magnus Madsen. The principles of the Flix programming language. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, page 112–127, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450399098. doi: 10.1145/3563835.3567661. URL https://doi.org/10.1145/3563835.3567661.

Magnus Madsen. Confirmed in personal communication with Magnus Madsen the lead developer of the Flix programming language, July 2026.

William Mansky and Ke Du. An Iris instance for verifying CompCert C programs. *Proceedings of the ACM on Programming Languages*, 8(POPL): 148–174, 2024.

Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of Andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434315. URL https://doi.org/10.1145/3434315.

Anders Alnor Mathiasen, Amin Timany, and Lars Birkedal. Verifying isolation levels of database implementations for free using separation logic, 2026a. URL https://arxiv.org/abs/2607.15877.

Anders Alnor Mathiasen, Amin Timany, and Lars Birkedal. Yarrow: Reconciling effect handlers and region-based memory management, 2026b. URL https://arxiv.org/abs/2607.15876.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore OCaml. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.

MicrosoftSQL. MicrosoftSQL documentation, 2025. URL https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server. Accessed 2025-06-02.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL https://doi.org/10.1145/128765.128770.

Lasse Møldrup and Andreas Pavlogiannis. AWDIT: An optimal weak database isolation tester. *Proceedings of the ACM on Programming Languages*, 9 (PLDI):1540–1564, 2025.

Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. Multiple resumptions and local mutable state, directly. *Proc. ACM Program. Lang.*, 9(ICFP), August 2025. doi: 10.1145/3747529. URL https://doi.org/10.1145/3747529.

Abel Nieto, Léon Gondelman, Alban Reynaud, and Lars Birkedal. Modular verification of op-based CRDTs in separation logic. *Proc. ACM Program. Lang.*, (OOPSLA), 2022. Accepted for publication.

Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal. Modular verification of state-based CRDTs in separation logic. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:27, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.22. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.22.

Peter W OHearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors,

2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014, pages 305–319. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

Oracle. Oracle documentation, 2025. URL https://docs.oracle.com/en/database/other-databases/timesten/22.1/introduction/transaction-isolation.html#GUID-7272EB20-51D6-412C-B25D-58986B3668C8. Accessed 2025-06-02.

Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. URL https://doi.org/10.1145/360051.360224.

Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, oct 1979. ISSN 0004-5411. doi: 10.1145/322154.322158. URL https://doi.org/10.1145/322154.322158.

Matthew Parkinson. The next 700 separation logics. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 169–182. Springer, 2010.

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing WebAssembly with effect handlers. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):460–485, 2023.

Lauren Pick, Amanda Xu, Ankush Desai, Sanjit A Seshia, and Aws Albarghouthi. Checking observational correctness of database systems. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1661–1688, 2025.

Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

PostgeSQL. PostgeSQL documentation, 2025. URL https://www.postgresql.org/docs/7.2/xact-read-committed.html. Accessed 2025-06-02.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. On parallel snapshot isolation and release/acquire consistency. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 940–967, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89884-1.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. *On the Semantics of Snapshot Isolation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings*, pages 1–23. 01 2019. ISBN 978-3-030-11244-8. doi: 10.1007/978-3-030-11245-5_1.

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-wasm: Robust and modular verification of webassembly programs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1096–1120, 2023.

J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002a. doi: 10.1109/LICS.2002.1029817.

John C Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th annual IEEE symposium on logic in computer science*, pages 55–74. IEEE, 2002b.

Gerhard Schellhorn, Stefan Bodenmüller, and Wolfgang Reif. Verification of forward simulations with thread-local, step-local proof obligations. *Science of Computer Programming*, 241:103227, 2025.

Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. Formal semantics and program logics for a fragment of OCaml. *Proceedings of the ACM on Programming Languages*, 9(ICFP):128–159, 2025.

Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613172. URL `https://doi.org/10.1145/3600006.3613172`.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 206–221, 2021.

Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. Automated validation of state-based client-centric isolation with TLA+. In *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops: ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14–15, 2020,*

Revised Selected Papers, page 43–57, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-67219-5. doi: 10.1007/978-3-030-67220-1_4. URL https://doi.org/10.1007/978-3-030-67220-1_4.

Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400, 2011.

Jane Street. OxCaml documentation, 2026. URL https://oxcaml.org/documentation/.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL*, 2(POPL):64:1–64:28, 2018. doi: 10.1145/3158152.

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proc. ACM Program. Lang.*, 8(POPL):241–272, 2024a. doi: 10.1145/3632851. URL https://doi.org/10.1145/3632851.

Amin Timany, Armaël Guéneau, and Lars Birkedal. The logical essence of well-bracketed control flow. *Proceedings of the ACM on Programming Languages*, 8(POPL):575–603, 2024b.

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *Journal of the ACM*, 71(6):1–75, 2024c.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML kit (for version 4). Technical report, Technical report, IT University of Copenhagen, 2001.

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

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, page 691–707, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660243. URL https://doi.org/10.1145/2660193.2660243.

Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 867–884, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509532. URL https://doi.org/10.1145/2509136.2509532.

Orpheas van Rooij and Robbert Krebbers. Affect: An affine type and effect system. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704841. URL https://doi.org/10.1145/3704841.

Simon Friis Vindum and Lars Birkedal. Contextual refinement of the Michael-Scott queue (proof pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 76–90, 2021.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913280. doi: 10.1145/99370.99404. URL https://doi.org/10.1145/99370.99404.

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. *Proc. ACM Program. Lang.*, 4 (POPL), December 2019. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *Proceedings of the 2022 international conference on management of data*, pages 1478–1492, 2022.

Shale Xiong. *Parametric Operational Semantics for Consistency Models*. Phd thesis, Imperial College London, 2020.

Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. Viper: A fast snapshot isolation checker. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 654–671, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3567492. URL https://doi.org/10.1145/3552326.3567492.

Zhanhao Zhao, Hexiang Pan, Gang Chen, Xiaoyong Du, Wei Lu, and Beng Chin Ooi. VeriTxn: Verifiable transactions for cloud-native databases with storage disaggregation. *Proceedings of the ACM on Management of Data*, 1(4):1–27, 2023.