AUTOMATIC REASONING TECHNIQUES FOR NON-SERIALIZABLE DATA-INTENSIVE APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Gowtham Kaki

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Suresh Jagannathan, Chair

Department of Computer Science

Dr. Xiangyu Zhang

Department of Computer Science

Dr. Tiark Rompf

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Approved by:

Dr. Voicu S Popescu

Head of the Department Graduate Program

To Pavani, my source of strength and wisdom.

In the pursuit of unity, elegance, and happiness ...

ACKNOWLEDGMENTS

I will be forever grateful to my advisor Suresh Jagannathan, who took a chance on me seven years ago when I had nothing to show except for my enthusiasm for functional programming. Right from the beginning, Suresh has been receptive of my ideas, regardless of how vague and ill-conceived they were, and would patiently work with me over multiple iterations of reifying them into research papers. I have immensely benefited from his constant encouragement to think bold and different, to question fundamental assumptions, and to fearlessly pursue unconventional approaches to their logical conclusion. Working with him over the past seven years has taught me how to think, changed my perception towards research and life in general, and helped me grow as a scientist and a human being.

I am thankful to KC Sivaramakrishnan, my long-time collaborator, mentor, and friend, for his constant encouragement and timely feedback. It is KC who introduced me to the fascinating world of distributed data stores, weak consistency and isolation – the focus of my current work, and also the topic of this thesis. I have hugely benefited from the numerous discussions we had over the years on these topics, several of them resulting in new ideas that are now published. As my senior in the research group, KC has set a fine example through the high standards of his work ethic for me to emulate

During my internships at Microsoft Research India, I have had the good fortune of collaborating with Ganesan Ramalingam, who, through his own example, reinforced in me such virtues as clarity of thought, intellectual humility, and sense of craftsmanship in doing research. I have also enjoyed my collaborations with several other immensely talented people, including Kartik Nagar, Mahsa Nazafzadeh, Kiarash Rahmani, Kapil Earanky, and Samodya Abeysiriwardane, each of whom taught me a new perspective in problem solving.

I am fortunate to have been part of a vibrant research community in Programming Languages at Purdue. Gustavo Petri, He Zhu, KC Sivaramakrishnan, and Xuankang Lin helped me bring the Purdue PL (PurPL) reading group into existence in Spring 2014. PurPL has since matured into an umbrella organization for all PL groups at Purdue, thanks to the meticulous work of Tiark Rompf, Roopsha Samantha, Ben Delaware, Milind Kulkarni, and their students. I have personally benefited from the sense of community and belongingness that PurPL fosters among the PL graduate students at Purdue.

I am thankful for friendship and encouragement that I received from several of my friends at Purdue, including Abhilash Jindal, GV Srikanth, Priya Murria, Vandith Pamuru, Sivakami Suresh, Habiba Farrukh, Devendra Verma, Raghu Vamsi, Jithin Joseph, Ravi Pramod, Suvidha Kancherla, and Praseem Banzal. They have made my long stay at Purdue enjoyable, and gave me memories that are worth cherishing.

I am incredibly lucky to have made lasting friendships during my undergraduate years at BITS, Pilani, which continued to support and sustain me through my Ph.D. Over the last seven years, it is to these friends that I have often turned to during the tough times when I felt an acute need for solace and acceptance. For their unadulterated friendship and unconditional acceptance, I am forever indebted to my friends Bharat and Swathi, Karthik and Vandana, Kartik and Mounica, Kashyap and Nikitha, Krishna and Subbu, Sandeep and Sowjanya, Sasidhar and Prathyusha, Uttam and Sreeja, and Mahesh Lagadapati. They constitute my family beyond my kin.

Lastly, but most importantly, I would like to thank my parents for reposing an immense faith in my abilities and values, and offering me a complete freedom to pursue my own interests without once expressing any misgiving. My greatest sense of gratitude however is towards my best friend and wife Pavani without whose unconditional love, tireless support, and constant encouragement none of the work described in this thesis would have been possible; If I have achieved anything through this thesis, it is as much hers as it is mine.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Kaki, Gowtham PhD, Purdue University, August 2019. Automatic Reasoning Techniques for Non-Serializable Data-Intensive Applications. Major Professor: Suresh Jagannathan.

The performance bottlenecks in modern data-intensive applications have induced database implementors to forsake high-level abstractions and trade-off simplicity and ease of reasoning for performance. Among the first casualties of this trade-off are the well-known ACID guarantees, which simplify the reasoning about concurrent database transactions. ACID semantics have become increasingly obsolete in practice due to serializable isolation – an integral aspect of ACID, being exorbitantly expensive. Databases, including the popular commercial offerings, default to weaker levels of isolation where effects of concurrent transactions are visible to each other. Such weak isolation guarantees, however, are extremely hard to reason about, and have led to serious safety violations in real applications. The problem is further complicated in a distributed setting with asynchronous state replications, where high availability and low latency requirements compel large-scale web applications to embrace weaker forms of consistency (e.g., eventual consistency) besides weak isolation. Given the serious practical implications of safety violations in data-intensive applications, there is a pressing need to extend the state-of-the-art in program verification to reach non-serializable data-intensive applications operating in a weakly-consistent distributed setting.

This thesis sets out to do just that. It introduces new language abstractions, program logics, reasoning methods, and automated verification and synthesis techniques that collectively allow programmers to reason about non-serializable data-intensive applications in the same way as their serializable counterparts. The contributions

made are broadly threefold. Firstly, the thesis introduces a uniform formal model to reason about weakly isolated (non-serializable) transactions on a sequentially consistent (SC) relational database machine. A reasoning method that relates the semantics of weak isolation to the semantics of the database program is presented, and an automation technique, implemented in a tool called ACIDIFIER is also described. The second contribution of this thesis is a relaxation of the machine model from sequential consistency to a specifiable level of weak consistency, and a generalization of the data model from relational to schema-less or key-value. A specification language to express weak consistency semantics at the machine level is described, and a bounded verification technique, implemented in a tool called Q9 is presented that bridges the gap between consistency specifications and program semantics, thus allowing high-level safety properties to be verified under arbitrary consistency levels. The final contribution of the thesis is a programming model inspired by version control systems that guarantees correct-by-construction *replicated data types* (RDTs) for building complex distributed applications with arbitrarily-structured replicated state. A technique based on decomposing inductively-defined data types into *characteristic relations* is presented, which is used to reason about the semantics of the data type under state replication, and eventually derive its correct-by-construction replicated variant automatically. An implementation of the programming model, called QUARK, on top of a content-addressable storage is described, and the practicality of the programming model is demonstrated with help of various case studies.

1 INTRODUCTION

Contemporary applications are *data-intensive*, as opposed to *compute-intensive* [1]; rather than being limited by the available CPU power, the performance bottlenecks in these applications pertain to handling high-volume reads and writes to large amounts of data with complex internal structure. While such applications have gained prominence in recent years — thanks to the proliferation of planet-scale web-services and mobile devices, data-intensive applications, in some form or the other, have been around since the beginning of the formal study of databases itself. Indeed, while the theory of databases has progressively introduced elegant abstractions to reason about database transactions, such as Atomicity, Serializability etc (collectively referred to as ACID guarantees), the performance bottlenecks in database applications have always induced practitioners to break such abstractions and tradeoff safety for performance whenever needed. A prototypical example of this kind is the proposal of *Weak Isolation* by Jim Grey *et al.* in 1976 [2]. Serializable isolation - an integral aspect of ACID guarantees, ensures that any permissible concurrent schedule of transactions yields results equivalent to a serial one in which there is no interleaving of actions from different transactions. As much as it simplifies the reasoning about concurrent transactions, Serializability does not come for free, however — pessimistic concurrency control methods require databases to use expensive mechanisms such as two-phase locking that incur overhead to deal with deadlocks, rollbacks, and re-execution [3, 4]. Similar criticisms apply to optimistic multi-version concurrency control methods that must deal with timestamp and version management [5]. Moreover, while serializability is a *sufficient* condition for correctness, it is often not *necessary* to ensure the overall correctness of an application if many interleavings of transactions turn out to be benign. It is based on these observations that Grey *et al.* proposed multiple "degrees" or "levels" of isolation weaker than serializability that try to recover perfor-

mance by breaking the ACID abstraction, even at the expense of simplicity and ease of reasoning. These weaker variants permit a transaction to witness various effects of newly committed, or even concurrently running, transactions while it executes, thereby weakening serializability's strong isolation guarantees. The ANSI SQL 92 standard defines three such weak isolation levels which are now implemented in many relational and NoSQL databases. Owing to their performance benefits [6–8], weak isolation levels have seen overwhelming adoption in practice [9], displacing serializability as the de facto isolation level on most ACID relational databases [10].

Indeed, strong isolation is not the only abstraction to have been breached by data-intensive applications in the pursuit of performance. The decade of 2000s saw the proliferation of planet-scale web applications, such as Google and Amazon, which replicate their state and logic across multiple *replicas* within and across data centers. Replication is intended not only to improve application throughput and reduce user-perceived latency, but also to tolerate partial failures without compromising overall service availability. Traditionally programmers have relied on *strong consistency* guarantees such as linearizability [11] in order to build correct applications. While serializability is a property of transactions, i.e., a group of operations over multiple objects, linearizability is a real-time guarantee about single operations on single objects. Linearizability requires that the final state of an object be a result of performing the operations on the object sequentially, where each operation appears to take effect sometime between its invocation and conclusion. While linearizability is an easily stated property, it masks the reality underlying large-scale distributed systems with respect to non-uniform latency, availability, and network partitions [12,13]. In particular, strong consistency *à la* linearizability is incompatible with high availability and network partition tolerance. Modern web services, which aim to provide an "always-on" experience, overwhelmingly favor availability and partition tolerance over strong consistency, resulting in the introduction of several *weak consistency* models such as eventual consistency, causal consistency, session guarantees, and timeline

consistency [14] which are now widely implemented in the off-the-shelf data stores that support geo-replication [15, 16].

An unfortunate but unsurprising fallout of breaking open the strong consistency and isolation abstractions is that applications now admit concurrency *anomalies* resulting in behaviors that are difficult to comprehend. The detrimental effect that admission of weak isolation anomalies has on the integrity of database applications has been well-studied both in the experimental and real-world settings [9, 17]. There have also been a few notable casualties among commercial enterprises due to weak isolation-related bugs [18–20]. The scenario with weak consistency is similar [21, 22]. While enforcing serializability (resp. linearizability) for all transactions (resp. operations) would be sufficient to avoid these errors and anomalies, it would likely be an overly conservative strategy; indeed, 75% of the application invariants studied in [9] were shown to be preserved under some form of weak isolation. When to use weak isolation and weak consistency, and in what form, is therefore a prominent question facing the programmers of data-intensive applications universally. The problem is exacerbated by the fact that there do not exist uniform formal specifications of various forms of weak consistency and isolation in a way that is suitable to reason about application correctness. For instance, Grey *et al.* define multiple degrees of weak isolation in terms of the database implementation details such as the nature and duration of locks in each case, which have no direct bearing on the application semantics. The ANSI SQL 92 standard defines four levels of isolation (including serializability) in terms of various undesirable *phenomena* (*e.g., dirty reads* - reading data written by an uncommitted transaction) each is required to prevent. This formulation requires programmers to be prescient about the possible ways various undesirable phenomena might manifest in their applications, and in each case determine if the phenomenon can be allowed without violating application invariants, which is almost impossible in practice given the non-existent tool support. Likewise, several proposals introducing weak consistency variants, such as [23], define them in terms of the relationships between low-level read and write operations, which are far below the high-level data

abstractions exposed to the applications. Given this unsatisfactory state of affairs, there is a pressing need to (a). Formalize the weak consistency and weak isolation properties uniformly at a high-level abstraction closer to the one exposed by the data store to the application, (b). Define proof systems that relate consistency and isolation specifications to the semantics of the application to enable rigorous reasoning and formal proofs of correctness of the latter, and (c). Develop proof automation methods and efficient implementation techniques to apply the aforementioned reasoning at scale. This thesis covers considerable ground towards these ends.

Note that consistency and isolation are properties of executions defined at the abstraction of objects and operations, and are as such independent of the data representation and its high-level interpretation. While data stores offering weakly-isolated transactions and linearizable operations tend to support relational data model suitable for deep join-intensive querying, replicated data stores that eschew strong consistency lean towards the variants of simpler key-value abstraction that preempt expensive joins across geo-distributed replicas. In either case, an application is exposed a uniform data abstraction (relational or key-value) on top of which it can structure its semantically-relevant high-level state. The underlying data structures used to support the abstraction are therefore irrelevant. While many commercially-deployed applications have successfully adopted this paradigm, there is a small but emerging class of distributed applications that break open the data representation abstraction for performance and software engineering reasons. Such applications, e.g., collaborative text editors and distributed ledgers, structure their (replicated) state in terms of bespoke inductively-defined data types, such as document trees and blockchains, whose data representation invariants are closely tied to the application logic. Under a geo-replicated setting, latency and availability requirements make weak consistency inevitable, which, in this case, might break the application invariants related to data representation. For instance, naively merging two concurrent edits at the same location of a shared document may result in a counterintuitive result, or even a violation of document tree integrity (e.g., concurrently inserted `span` and `div` elements in an

HTML document could be incorrectly merged by nesting the `div` inside the `span`, which violates HTML tree integrity). Thus, to engineer a replicated version of an inductive data type, one has to carefully reason about the data type's behavior under replication, identify possible conflicts that result in data representation invariant violations, and extend the type definition with the semantics of reconciling such replication conflicts in a meaningful way. For non-trivial inductive data types, this process is intellectually challenging enough to become the focus of a few research papers [24,25]. If the process of building distributed application around replicated data types were to become any easier, there is an immediate need for automated reasoning techniques that assist, or even automate, the process of engineering replicated data types out of inductively-defined sequential data types. This thesis makes contributions towards this end too.

1.1 Contributions

The various contributions made by this thesis are briefly summarized below.

1.1.1 Compositional Reasoning and Inference for Weak Isolation

The first major contribution of this thesis is a program logic for weakly-isolated transactions along with automated verification support to allow developers to verify the soundness of their applications, without having to resort to low-level operational reasoning as they are forced to do currently. The thesis presents a declarative characterization of weak isolation, and describes a set of syntax-directed compositional proof rules that enable the construction of correctness proofs for transactional programs in the presence of weak isolation. Realizing that the proof burden imposed by these rules may discourage applications programmers from using the proof system, the thesis also develops an inference procedure that automatically verifies the weakest isolation level for a transaction while ensuring its invariants are maintained. The key to inference is a novel formulation of relational database state (represented as sets

of tuples) as a monad, and in which database computations are interpreted as state transformers over these sets. This interpretation leads to an encoding of database computations amenable for verification by off-the-shelf SMT solvers. The approach is realized as an extended static analysis pass called ACIDifier in the context of a DSL embedded in OCaml. An evaluation study featuring standard database benchmarks is presented.

1.1.2 Bounded Verification under Weak Consistency

The second major contribution of this thesis is a bounded verification and inference technique that automatically detects weak consistency anomalies in applications with replicated state, and infers the weakest strengthening of consistency level required to preempt each anomaly. A programming framework embedded in OCaml is presented that lets applications define a well-typed replicated state on top of off-the-shelf key-value stores, and compose effectful computations around it. The framework is equipped with a symbolic execution engine called Q9 that systematically explores the state space of an application executing on top of an eventually consistent data store, under an *unrestricted* consistency model but with a *finite* concurrency bound. Q9 uncovers anomalies (i.e., invariant violations) that manifest as finite counterexamples, and automatically generates repairs for such anomalies by selectively strengthening consistency guarantees for specific operations. Crucial to this step is a novel formulation of Consistency guarantees as constraints over axiomatic executions, which is also presented. An evaluation study Q9 over implementations of well-known benchmarks is presented, which also includes a discussion on subtle anomalies uncovered in such implementations.

1.1.3 Principled Derivation of Mergeable Replicated Data Types

The third and final major contribution of this thesis is a novel programming model inspired by version control systems (e.g., Git) that lets one build distributed applica-

tions around *mergeable* replicated data types *derived* from their inductively-defined sequential counterparts using first principles. The derivation relies on *invertible relational specifications* of an inductively-defined data type as a mechanism to capture salient aspects of the data type relevant to how its different instances can be safely merged in a replicated environment without violating data representation invariants. Importantly, because these specifications only address a data type's (static) structural properties, their formulation does not require exposing low-level system-level details concerning asynchrony, replication, visibility, etc. As a consequence, the version control-based programming framework enables the correct-by-construction synthesis of rich merge functions over arbitrarily complex (i.e., composable) data types. The framework, called Quark is implemented as a shallow extension of OCaml on top of a content-addressable storage abstraction. An evaluation study featuring multiple replicated data types, and applications composed of such data types, including the replicated variants of standard database benchmarks, is presented.

1.2 Roadmap

The rest of the thesis is organized as follows. The next three chapters (2, 3, and 4) present a detailed technical development and evaluation studies pertaining to the aforementioned three major contributions (ACIDifier, Q9, and Quark) respectively. The last section of each of the three chapters discusses the body of work that is closely related to the work presented in the chapter. The last chapter, Chapter 5, presents the conclusions of this thesis, along with a discussion on possible directions that the future work can take.

2 COMPOSITIONAL REASONING AND INFERENCE FOR WEAK ISOLATION

In this chapter, we[1] present a high-level formalism capable of uniformly specifying various weak isolation guarantees found in the database literature and practice. The formalism is high-level in the sense that it captures the semantics of weak isolation in relation to the operational semantics of a transactional program, as against the low-level trace-based characterization found in the literature. The advantage of such high-level characterization is that it lets us extend the existing concurrent program logics with an awareness of weak isolation, and use it as a basis to develop a proof system capable of verifying high-level invariants, and even full-functional correctness, of data-intensive applications composed of database transactions. We complement the proof system with an inference and proof automation technique that is described in the later sections of this chapter. The approach is implemented in a static analysis tool called ACIDifier in the context of a DSL embedded in OCaml that treats SQL-based relational database operations (e.g., inserts, selects, deletes, updates, etc.) as computations over an abstract database state.

Before we delve into technical development, we shall first motivate the problem of verifying weakly-isolated transactions through an example.

2.1 Motivation

Fig. 2.1 shows a simplified version of the TPC-C `new_order` transaction written in our OCaml DSL. The DSL is manages an abstract database state that can be manipulated via a well-defined `SQL` interface. Arbitrary database computations can be built

[1]This work was done in collaboration with Kartik Nagar, Mahsa Nazafzadeh, and Suresh Jagannathan.

```
let new_order (d_id, c_id, item_reqs) = atomically_do @@ fun () ->
  let dist = SQL.select1 District (fun d -> d.d_id = d_id) in
  let o_id = dist.d_next_o_id in
  begin
    SQL.update(* UPDATE *) District
        (* SET *)(fun d -> {d with d_next_o_id = d.d_next_o_id + 1})
        (* WHERE *)(fun d -> d.d_id = d_id );
    SQL.insert(* INSERT INTO *) Order (* VALUES *)
          {o_id=o_id; o_d_id=d_id; o_c_id=c_id;
           o_ol_cnt=S.size item_reqs};
    foreach item_reqs @@ fun item_req ->
      let stk = SQL.select1(* SELECT * FROM *) Stock
            (* WHERE *)(fun s -> s.s_i_id = item_req.ol_i_id &&
                                 s.s_d_id = d_id)(* LIMIT 1 *) in
      let s_qty' = if stk.s_qty >= item_req.ol_qty + 10
                   then stk.s_qty - item_req.ol_qty
                   else stk.s_qty - item_req.ol_qty + 91 in
      SQL.update Stock (fun s -> {s with s_qty = s_qty'})
                       (fun s -> s.s_i_id = item_req.ol_i_id);
      SQL.insert Order_line {ol_o_id=o_id; ol_d_id=d_id;
                             ol_i_id=item_req.ol_i_id;
                             ol_qty=item_req.ol_qty}
  end
```

Figure 2.1.: TPC-C `new_order` transaction

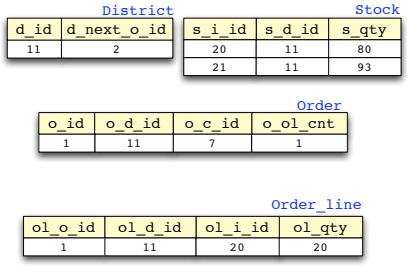
around this interface, which can then be run as transactions using the `atomically_do` combinator provided by the DSL. TPC-C is a widely-used and well-studied Online Transaction Processing (OLTP) benchmark that models an order-processing system for a wholesale parts supply business. The business logic is captured in 5 database transactions that operate on 9 tables; `new_order` is one such transaction that uses `District`, `Order`, `New_order`, `Stock`, and `Order_line` tables. The transaction acts on the behalf of a customer, whose id is `c_id`, to place a new order for a given

set of items (`item_reqs`), to be served by a warehouse under the district identified by `d_id`. Fig. 2.2 illustrates the relationship among these different tables.

The transaction manages order placement by invoking appropriate SQL functionality, captured by various calls to functions defined by the `SQL` module. All `SQL` operators supported by the module take a table name (a nullary constructor) as their first argument. The higher-order `SQL.select1` function accepts a boolean function that describes the selection criteria, and returns any record that meets the criteria (it models the SQL query `SELECT ... LIMIT 1`). `SQL.update` also accepts a boolean function (its 3^{rd} argument) to select the records to be updated. Its 2^{nd} argument is a function that maps each selected record to a new (updated) record. `SQL.insert` inserts a given record into the specified table in the database.

The `new_order` transaction inserts a new `Order` record, whose id is the sequence number of the next order under the given district (`d_id`). The sequence number is stored in the corresponding `District` record, and updated each time a new order is added to the system. Since each order may request multiple items (`item_reqs`), an `Order_line` record is created for each requested item to relate the order with the item. Each item has a corresponding record in the `Stock` table, which keeps track of the quantity of the item left in stock (`s_qty`). The quantity is updated by the transaction to reflect the processing of new orders (if the stock quantity falls below 10, it is automatically replenished by 91).

TPC-C defines multiple invariants, called *consistency conditions*, over the state of the application in the database. One such consistency condition is the requirement that for a given order `o`, the *order-line-count* field (`o.o_ol_cnt`) should reflect the number of order lines under the order; this is the number of `Order_line` records whose `ol_o_id` field is the same as `o.o_id`. In a sequential execution, it is easy to see how this condition is preserved. A new `Order` record is added with its `o_id` distinct from existing order ids, and its `o_ol_cnt` is set to be equal to the size of the `item_reqs` set. The `foreach` loop runs once for each `item_req`, adding a new `Order_line` record for each requested item, with its `ol_o_id` field set to `o_id`. Thus, at the end



District

d_id	d_next_o_id
11	2

Stock

s_i_id	s_d_id	s_qty
20	11	80
21	11	93

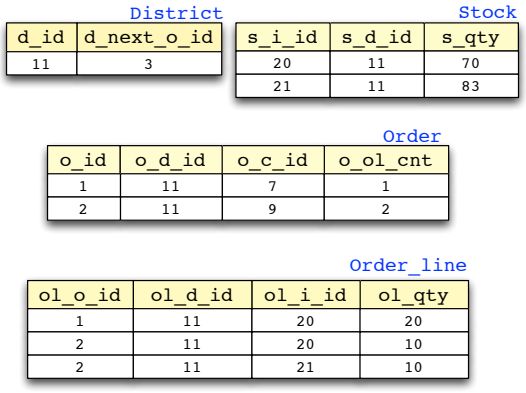
Order

o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20

(a) A valid TPC-C database. The only existing order belongs to the district with `d_id`=11. Its id (`o_id`) is one less than the district's `d_next_o_id`, and its order count (`o_ol_cnt`) is equal to the number of order line records whose `ol_o_id` is equal to the order's id.



District

d_id	d_next_o_id
11	3

Stock

s_i_id	s_d_id	s_qty
20	11	70
21	11	83

Order

o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1
2	11	9	2

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20
2	11	20	10
2	11	21	10

(b) The database in Fig. 2.2a after correctly executing a `new_order` transaction. A new order record is added whose `o_id` is equal to the `d_next_o_id` from Fig. 2.2a. The district's `d_next_o_id` is incremented. The order's `o_ol_cnt` is 2, reflecting the actual number of order line records whose `ol_o_id` is equal to the order's id (2).

Figure 2.2.: Database schema of TPC-C's order management system. The naming convention indicates primary keys and foreign keys. For e.g., `ol_id` is the primary key column of the order line table, whereas `ol_o_id` is a foreign key that refers to the `o_id` column of the order table.

of the loop, the number of `Order_line` records in the database (i.e., the number of records whose `ol_o_id` field is equal to `o_id`) is guaranteed to be equal to the size of the `item_reqs` set, which in turn is equal to the `Order` record's `o_ol_cnt` field; these constraints ensure that the transaction's consistency condition is preserved.

Because the aforementioned reasoning is reasonably simple to perform manually, verifying the soundness of TPC-C's consistency conditions would appear to be feasible. Serializability aids the tractability of verification by preventing any interference among concurrently executing transactions while the `new_order` transaction

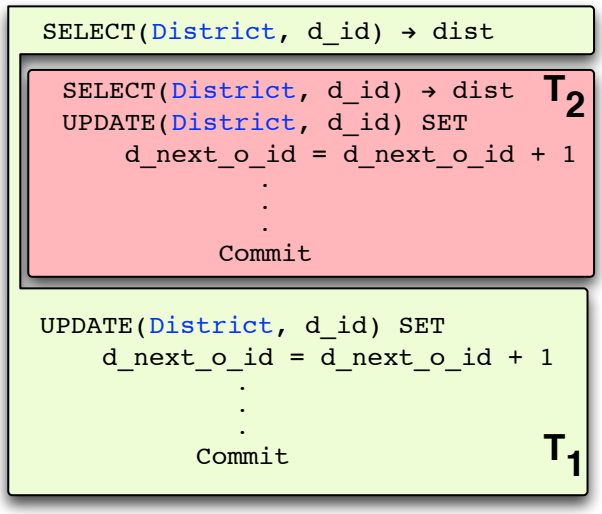


Figure 2.3.: An RC execution involving two instances (T_1 and T_2) of the `new_order` transaction depicted in Fig. 2.1. Both instances read the `d_id District` record concurrently, because neither transaction is committed when the reads are executed. The subsequent operations are effectively sequentialized, since T_2 commits before T_1. Nonetheless, both transactions read the same value for `d_next_o_id` resulting in them adding `Order` records with the same ids, which in turn triggers a violation of TPC-C's consistency condition.

executes, essentially yielding serial behaviors. Under weak isolation[2], however, interferences of various kinds are permitted, leading to executions superficially similar to executions permitted by concurrent (racy) programs [26, 27]. To illustrate, consider the behavior of the `new_order` transaction when executed under a *Read Committed* (RC) isolation level, the default isolation level in 8 of the 18 databases studied in [10]. An executing RC transaction is isolated from *dirty writes*, i.e., writes of uncommitted transactions, but is allowed to witness the writes of concurrent transactions as soon as they are committed. Thus, with two concurrent instances of the `new_order` transaction (call them T_1 and T_2), both concurrently placing new orders for different customers under the same district (`d_id`), RC isolation allows the execution shown in Fig. 2.3.

[2]Weak isolation does not violate atomicity as long as the witnessed effects are those of committed transactions

The figure depicts an execution as a series of SQL operations. In the execution, the `new_order` instance T_1 (green) reads the `d_next_o_id` field of the district record for `d_id`, but before it increments the field, another `new_order` instance T_2 (red) begins its execution and commits. Note that T_2 reads the same `d_next_o_id` value as T_1, and inserts new `Order` and `Order_line` records with their `o_id` and `ol_o_id` fields (resp.) equal to `d_next_o_id`. T_2 also increments the `d_next_o_id` field, which T_1 has already acccessed. This is allowed because reads typically do not obtain a mutually exclusive lock on most databases. After T_2's commit, T_1 resumes execution and adds new `Order` and `Order_line` fields with the same order id as T_1. Thus, at the end of the execution, `Order_line` records inserted by T_1 and T_2 all bear the same order id. There are also two `Order` records with the same district id (`d_id`) and order id, none of whose `o_ol_cnt` reflects the actual number of `Order_line` records inserted with that order id. This clearly violates TPC-C's consistency condition.

This example does not exhibit any of the anomalies that *characterize* RC isolation [28][3]. For instance, there are no *lost writes* since both concurrent transactions' writes are present in the final state of the database. Program analyses that aim to determine appropriate isolation by checking for possible manifestations of RC-induced anomalies would fail to identify grounds for promoting the isolation level of `new_order` to something stronger. Yet, if we take the semantics of the application into account, it is quite clear that RC is not an appropriate isolation level for `new_order`.

While reasoning in terms of anomalies is cumbersome and inadequate, reasoning about weak isolation in terms of traces [29,30] on memory read and write actions can complicate high-level reasoning. A possible alternative would be to utilize concurrent program verification methods where the implementation details of weak isolation are interleaved within the program, yielding a (more-or-less) conventional concurrent program. But, considering the size and complexity of real-world transaction systems, this strategy is unlikely to scale.

[3]Berenson *et al.* characterize isolation levels in terms of the anomalies they *admit*. For example, RC is characterized by *lost writes* because it admits the anomaly.

In this chapter, we adopt a different approach that *lifts* isolation semantics (*not* their implementations) to the application layer, providing a principled framework to simultaneously reason about application invariants and isolation properties. To illustrate this idea informally, consider how we might verify that `new_order` is sound when executed under *Snapshot Isolation* (SI), a stronger isolation level than RC. Snapshot isolation allows transactions to be executed against a private snapshot of the database, thus admitting concurrency, but it also requires that there not be any write-write conflicts (i.e., such a conflict occurs if concurrently executing transactions modify the same record) among concurrent transactions when they commit. Write-write conflicts can be eliminated in various ways, e.g., through conflict detection followed by a rollback, or through exclusive locks, or a combination of both. For instance, one possible implementation of SI, close to the one used by PostgreSQL [31], executes a transaction against its private snapshot of the database, but obtains exclusive locks on the actual records in the database before performing writes. A write is performed only if the record that is to be written has not already been updated by a concurrent transaction. Conflicts are resolved by abort and roll back.

As this discussion hints, implementations of SI on real databases such as PostgreSQL are highly complicated, often running into thousands of lines of code. Nonetheless, the semantics of SI, in terms of how it effects transitions on the database state, can be captured in a fairly simple model. First, effects induced by one transaction (call it T) are not visible to another concurrently executing one during T's execution. Thus, from T's perspective, the global state does not change during its execution. More formally, for every operation performed by T, the global state T witnesses before (Δ) and after (Δ') executing the operation is the same ($\Delta' = \Delta$). After T finishes execution, it commits its changes to the actual database, which may have already incorporated the effects of concurrent transactions. In executions where T successfully commits, concurrent transactions are guaranteed to not be in write-write conflict with T. Thus, if Δ is the global state that T witnessed when it finished execution (the snapshot state), and Δ' is the state to which T commits, then the difference between

Δ and Δ' should not result in a write-write conflict with T. To concretize this notion, let the database state be a map from database locations to values, and let δ denote a transaction-local log that maps the locations being written to their updated values. The absence of write-write conflicts between T and the diff between Δ and Δ' can be expressed as: $\forall x \in dom(\delta)$, $\Delta'(x) = \Delta(x)$. In other words, the semantics of SI can be captured as an axiomatization over transitions of the database state ($\Delta \longrightarrow \Delta'$) during a transaction's (T) lifetime:

- While T executes, $\Delta' = \Delta$.
- After T finishes execution, but before it commits its local state δ, $\forall(x \in dom(\delta))$. $\Delta'(x) = \Delta(x)$.

This simple characterization of SI isolation allows us to verify the consistency conditions associated with the `new_order` transaction. First, since the database does not change ($\Delta' = \Delta$) during execution of the transaction's body, we can reason about `new_order` as though it executed in complete isolation until its commit point, leading to a verification process similar to what would have been applied when reasoning sequentially. When `new_order` finishes execution, however, but before it commits, the SI axiomatization shown above requires us to consider global state transitions $\Delta \longrightarrow \Delta'$ that do not include changes to the records (δ) written by `new_order`, i.e., $\forall(x \in dom(\delta))$. $\Delta'(x) = \Delta(x)$. The axiomatization precludes any execution in which there are concurrent updates to shared table fields (e.g., `d_next_o_id` on the same `District` table), but does not prohibit interferences that write to different tables, or write to different records in the same table. We need to reason about the safety of such interferences with respect to `new_order`'s consistency invariants to verify `new_order`.

We approach the verification problem by first observing that a relational database is a significantly simpler abstraction than shared memory. Its primary data structure is a table, with no primitive support for pointers, linked data structures, or aliasing. Although a database essentially abstracts a mutable state, this state is managed

through a well-defined fixed number of interfaces (SQL statements), each tagged with a logical formula describing what records are accessed and updated.

This observation leads us away from thinking of a collection of database transactions as a simple variant of a concurrent imperative program. Instead, we see value in viewing them as essentially functional computations that manage database state abstractly, mirroring the structure of our DSL. By doing so, we can formulate the semantics of database operations as state transformers that explicitly relate an operation's pre- and post-states, defining the semantics of the corresponding transformer algorithmically, just like classical predicate transformer semantics (e.g., weakest precondition or strongest post-condition). In our case, a transformer interprets a SQL statement in the set domain, modeling the database as a set of records, and a SQL statement as a function over this set. Among other things, one benefit of this approach is that low-level loops can now be substituted with higher-order combinators that automatically lift the state transformer of its higher-order argument, i.e., the loop body, to the state transformer of the combined expression, i.e., the loop. We illustrate this intuition on a simple example.

```
foreach item_reqs @@ fun item_req ->
  SQL.update Stock (fun s -> {s with s_qty = k1})
                   (fun s -> s.s_i_id = item_req.ol_i_id);
  SQL.insert Order_line {ol_o_id=k2; ol_d_id=k3;
                         ol_i_id=item_req.ol_i_id; ol_qty=item_req.
                            ol_qty}
```

Figure 2.4.: Foreach loop from Fig. 2.1

Fig. 2.4 shows a (simplified) snippet of code taken from Fig. 2.1. Some irrelevant expressions have been replaced with constants (`k1`, `k2`, and `k3`). The body of the loop executes a SQL update followed by an insert. Recall that a transaction reads from the global database (Δ), and writes to a transaction-local database (δ) before committing these updates. An update statement filters the records that match the

search criteria from Δ and computes the updated records that are to be added to the local database. Thus, the state transformer for the update statement (call it T_U) is the following function on sets[4]:

$$\begin{aligned}\lambda(\delta, \Delta).\ \delta \cup \Delta \gg= (\lambda\, \mathtt{s}\ .\mathsf{if}\ \mathsf{table}(\,\mathtt{s}\,) = \mathtt{Stock} \wedge \mathtt{s.s\_i\_id} = \mathtt{item\_req.ol\_i\_id} \\ \mathsf{then}\ \{\langle\, \mathtt{s\_i\_id} = \mathtt{s.s\_i\_id};\ \mathtt{s\_d\_id} = \mathtt{s.s\_d\_id};\ \mathtt{s\_qty} = \mathtt{k1}\,\rangle\} \\ \mathsf{else}\ \emptyset)\end{aligned}$$

Here, the set bind operator extracts record elements ($\mathtt{s}$) from the database, checks the precondition of the update action, and if satisfied, constructs a new set containing a single record that is identical to $\mathtt{s}$ except that it binds field $\mathtt{s\_qty}$ to value $\mathtt{k_1}$. This new set is added (via set union) to the existing local database state δ.[5]

The transformer ($\mathsf{T}_I(\delta, \Delta)$) for the subsequent `insert` statement can be similarly constructed:

$$\begin{aligned}\lambda(\delta, \Delta).\ \delta \cup \{\langle\, \mathtt{ol\_o\_id} = \mathtt{k2};\ \mathtt{ol\_d\_id} = \mathtt{k3};\ \mathtt{ol\_i\_id} = \mathtt{item\_req.ol\_i\_id}; \\ \mathtt{ol\_qty} = \mathtt{item\_req.ol\_qty}\,\rangle\}\end{aligned}$$

Observe that both transformers are of the form $\mathsf{T}(\delta, \Delta) = \delta \cup \mathsf{F}(\Delta)$, where F is a function that returns the set of records added to the transaction-local database (δ). Let F_U and F_I be the corresponding functions for T_U and T_I shown above. The state transformation induced by the loop body in Fig. 2.1 can be expressed as the following composition of F_U and F_I:

$$\lambda(\delta, \Delta).\ \delta \cup \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta)$$

The transformer for the loop itself can now be computed to be:

$$\lambda(\delta, \Delta).\ \delta \cup \mathtt{item\_reqs} \gg= (\lambda\, \mathtt{item\_req}\,.\ \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta))$$

Observe that the structure of the transformer mirrors the structure of the program itself. In particular, SQL statements become set operations, and the `foreach` combinator becomes set monad's bind ($\gg=$) combinator. As we demonstrate, the advantage of inferring such transformers is that we can now make use of a semantics-preserving

[4]Bind ($\gg=$) has higher precedence than union ($\cup$). Angle braces ($\langle \ldots \rangle$) are used to denote records.
[5]For now, assume that the record being added is not already present in δ.

translation from the domain of sets equipped with $\gg=$ to a decidable fragment of first-order logic, allowing us to leverage SMT solvers for automated proofs without having to infer potentially complex thread-local invariants or intermediate assertions. Sec. 2.4 describes this translation. In the exposition thus far, we assumed Δ remains invariant, which is clearly not the case when we admit concurrency. Necessary concurrency extensions of the state transformer semantics to deal with interference is also covered in Sec. 2.4. Before presenting the transformer semantics, we first focus our attention in the following two sections on the theoretical foundations for weak isolation, upon which this semantics is based.

2.2 $\mathcal{T}$: Syntax and Semantics

Fig. 2.5 shows the syntax and small-step semantics of $\mathcal{T}$, a core language that we use to formalize our intuitions about reasoning under weak isolation. Variables (x), integer and boolean constants (k), records (r) of named constants, sets (s) of such records, arithmetic and boolean expressions ($e_1 \odot e_2$), and record expressions ($\langle \overline{f} = \overline{e} \rangle$) constitute the syntactic class of expressions (e). Commands (c) include `SKIP`, conditional statements, `LET` constructs to bind names, `FOREACH` loops, SQL statements, their sequential composition ($c_1; c_2$), transactions ($\texttt{TXN}_i \langle \mathbb{I} \rangle \{c\}$) and their parallel composition ($c_1 \,||\, c_2$). Each transaction is assumed to have a unique identifier i, and executes at the top-level; our semantics does not support nested transactions. The $\mathbb{I}$ in the `TXN` block syntax is the transaction's isolation specification, whose purpose is explained below. Certain terms that only appear at run-time are also present in c. These include a `txn` block tagged with sets (δ and Δ) of records representing local and global database state, and a runtime `foreach` expression that keeps track of the set (s_1) of items already iterated, and the set (s_2) of items yet to be iterated. Note that the surface-level syntax of the `FOREACH` command shown here is slightly different from the one used in previous sections; its higher-order function has two arguments, y and z, which are invoked (during the reduction) with the set of already-

Syntax

$$x, y \in \texttt{Variables} \quad f \in \texttt{Field Names} \quad i, j \in \mathbb{N} \quad \odot \in \{+, -, \leq, \geq, =\}$$

$$k \in \mathbb{Z} \cup \mathbb{B} \quad r \in \langle \overline{f} = \overline{k} \rangle$$

$$\begin{array}{llll}
\delta, \Delta, s & \in & \texttt{State} & \coloneqq \mathcal{P}\left(\langle \overline{f} = \overline{k} \rangle\right) \\
\mathbb{I}_e, \mathbb{I}_c & \in & \texttt{IsolationSpec} & \coloneqq (\delta, \Delta, \Delta') \to \mathbb{P} \\
v & \in & \texttt{Values} & \coloneqq k \mid r \mid s \\
e & \in & \texttt{Expressions} & \coloneqq v \mid x \mid x.f \mid \langle \overline{f} = \overline{e} \rangle \mid e_1 \odot e_2 \\
c & \in & \texttt{Commands} & \coloneqq \texttt{LET } x = e \texttt{ IN } c \mid \texttt{IF } e \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \mid c_1; c_2 \\
 & & & \quad \mid \texttt{INSERT } x \mid \texttt{DELETE } \lambda x.e \\
 & & & \quad \mid \texttt{LET } y = \texttt{SELECT } \lambda x.e \texttt{ IN } c \\
 & & & \quad \mid \texttt{UPDATE } \lambda x.e_1 \ \lambda x.e_2 \mid \texttt{FOREACH } x \texttt{ DO } \lambda y.\lambda z.c \\
 & & & \quad \mid \mathsf{foreach}\langle s_1 \rangle \ s_2 \ \mathsf{do} \ \lambda x.\lambda y.e \mid \texttt{TXN}_i\langle \mathbb{I} \rangle \{c\} \\
 & & & \quad \mid \mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle \{c\} \mid c_1 || c_2 \mid \texttt{SKIP} \\
\mathcal{E} & \in & \texttt{Eval Ctx} & ::= \bullet \mid \bullet \,||c_2 \mid c_1 || \bullet \mid \bullet; c_2 \mid \mathsf{txn}_i\langle \mathbb{I}, \delta, \Delta \rangle \{\bullet\}
\end{array}$$

Figure 2.5.: $\mathcal{T}$: Syntax

iterated items, and the current item, respectively. This form of `FOREACH` lends itself to inductive reasoning that will be useful for verification (Sec. 2.3). Our language ensures that all effectful actions are encapsulated within database commands, and that all shared state among processes are only manipulated via transactions and its supported operations. In particular, we do not consider programs in which objects resident on e.g., the OCaml heap are concurrently manipulated by OCaml expressions as well as database actions.

Figs. 2.6 and 2.7 define a small-step operational semantics for this language in terms of an abstract machine that executes a command, and updates either a

Local Reduction $\boxed{\Delta \vdash ([c]_i, \delta) \longrightarrow ([c']_i, \delta')}$

E-INSERT

$$\frac{r.\texttt{id} \notin \mathsf{dom}(\delta \cup \Delta) \quad r' = \langle r \texttt{ with txn} = i; \texttt{del} = \texttt{false} \rangle}{\Delta \vdash ([\texttt{INSERT } r]_i, \delta) \longrightarrow ([\texttt{ SKIP }]_i, \delta \cup \{r'\})}$$

E-SELECT

$$\frac{s = \{r \in \Delta \mid \mathsf{eval}([r/x]e) = \texttt{ true }\} \quad c' = [s/y]c}{\Delta \vdash ([\texttt{ LET } y = \texttt{SELECT } \lambda x.e \texttt{ IN } c]_i, \delta) \longrightarrow ([c']_i, \delta)}$$

E-DELETE

$$\frac{\begin{array}{c}\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset \\ s = \{r' \mid \exists (r \in \Delta).\ \mathsf{eval}([r/x]e) = \texttt{ true } \wedge r' = \langle r \texttt{ with del} = \texttt{true}; \texttt{txn} = i \rangle\}\end{array}}{\Delta \vdash ([\texttt{DELETE } \lambda x.e]_i, \delta) \longrightarrow ([\texttt{ SKIP }]_i, \delta \cup s)}$$

E-UPDATE

$$\frac{\begin{array}{c}\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset \\ s = \{r' \mid \exists (r \in \Delta).\ \mathsf{eval}([r/x]e_2) = \texttt{ true } \wedge \\ \qquad r' = \langle [r/x]e_1 \texttt{ with id} = r.\texttt{id}; \texttt{txn} = i; \texttt{del} = r.\texttt{del} \rangle\}\end{array}}{\Delta \vdash ([\texttt{UPDATE } \lambda x.e_1\ \lambda x.e_2]_i, \delta) \longrightarrow ([\texttt{ SKIP }]_i, \delta \cup s)}$$

E-FOREACH1 $\Delta \vdash ([\texttt{FOREACH } s \texttt{ DO } \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([\mathsf{foreach}\langle\emptyset\rangle\ s\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta)$

E-FOREACH2 $\Delta \vdash ([\mathsf{foreach}\langle s_1 \rangle\ \{r\} \uplus s_2\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([[r/z][s_1/y]c;$
$\mathsf{foreach}\langle s_1 \cup \{r\}\rangle\ s_2\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta)$

E-FOREACH3 $\Delta \vdash ([\mathsf{foreach}\langle s \rangle\ \emptyset\ \mathsf{do}\ \lambda y.\lambda z.c]_i, \delta) \longrightarrow ([\texttt{ SKIP }]_i, \delta)$

Figure 2.6.: $\mathcal{T}$: Transaction-local reduction

transaction-local (δ), or global (Δ) database, both of which are modeled as a set of records of a pre-defined type, i.e., they all belong to a single table. The generalization

Top-Level Reduction $\boxed{(c, \Delta) \longrightarrow (c', \Delta')}$

E-Txn-Start

$$\frac{}{(\,\texttt{TXN}_{\,i}\langle\mathbb{I}\rangle\{c\}, \Delta) \longrightarrow (\mathsf{txn}_i\langle\mathbb{I}, \emptyset, \Delta\rangle\{c\}, \Delta)}$$

E-Txn

$$\frac{\mathbb{I}_e\ (\delta, \Delta, \Delta') \quad \Delta \vdash ([c]_i, \delta) \longrightarrow ([c']_i, \delta')}{(\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}, \Delta') \longrightarrow (\mathsf{txn}_i\langle\mathbb{I}, \delta', \Delta'\rangle\{c'\}, \Delta')}$$

E-Commit

$$\frac{\mathbb{I}_c\ (\delta, \Delta, \Delta')}{(\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{\,\texttt{SKIP}\,\}, \Delta') \longrightarrow (\,\texttt{SKIP}\,, \delta \rhd \Delta')}$$

Figure 2.7.: $\mathcal{T}$: Top-level reduction

to multiple tables is straightforward, e.g., by having the machine manipulate a set of sets, one for each table. The semantics assumes that records in Δ can be uniquely identified via their `id` field, and enforces this property wherever necessary. Certain hidden fields are treated specially by the operational semantics, and are hidden from the surface language. These include a `txn` field that tracks the identifier of the transaction that last updated the record, and a `del` field that flags deleted records in δ. For a set S of records, we define $\mathsf{dom}(S)$ as the set of unique ids of all records in S. Thus $|\mathsf{dom}(\Delta)| = |\Delta|$. During its execution, a transaction may write to multiple records in Δ. Atomicity dictates that such writes should not be visible in Δ until the transaction commits. We therefore associate each transaction with a local database (δ) that stores such uncommitted records[6]. Uncommitted records include deleted records, whose `del` field is set to `true`. When the transaction commits, its local database is atomically *flushed* to the global database, committing these heretofore uncommitted records. The flush operation ($\rhd$) is defined as follows:

$$\forall r.\ r \in (\delta \rhd \Delta) \ \Leftrightarrow\ (r.\texttt{id} \notin \mathsf{dom}(\delta) \wedge r \in \Delta) \vee (r \in \delta \wedge \neg r.\texttt{del})$$

[6]While SQL's `UPDATE` admits writes at the granularity of record fields, most popular databases enforce record-level locking, allowing us to think of "uncommitted writes" as "uncommitted records".

Let $\Delta' = \delta \triangleright \Delta$. A record r belongs to Δ' iff it belongs to Δ and has not been updated in δ, i.e., $r.\texttt{id} \notin \mathsf{dom}(\delta)$, or it belongs to δ, i.e., it is either a new record, or an updated version of an old record, provided the update is not a deletion ($\neg r.\texttt{del}$). Besides the commit, flush also helps a transaction read its own writes. Intuitively, the result of a read operation inside a transaction must be computed on the database resulting from flushing the current local state (δ) to the global state (Δ). The abstract machine of Fig. 2.7, however, does not let a transaction read its own writes. This simplifies the semantics, without losing any generality, since substituting $\delta \triangleright \Delta$ for Δ at select places in the reduction rules effectively allows reads of uncommitted transaction writes to be realized, if so desired.

The small-step semantics is stratified into a transaction-local reduction relation, and a top-level reduction relation. The transaction-local relation ($\Delta \vdash (c, \delta) \longrightarrow (c', \delta')$) defines a small-step reduction for a command inside a transaction, when the database state is Δ; the command c reduces to c', while updating the transaction-local database δ to δ'. The definition assumes a meta-function eval that evaluates closed terms to values. The reduction relation for SQL statements is defined straightforwardly. `INSERT` adds a new record to δ after checking the uniqueness of its id. `DELETE` finds the records in Δ that match the search criteria defined by its boolean function argument, and adds the records to δ after marking them for deletion. `SELECT` bounds the name introduced by `LET` to the set of records from Δ that match the search criteria, and then executes the bound command c. `UPDATE` uses its first function argument to compute the updated version of the records that match the search criteria defined by its second function argument. Updated records are added to δ.

The reduction of `FOREACH` starts by first converting it to its run-time form to keep track of iterated items (s_1), as well as yet-to-be-iterated items (s_2). Iteration involves invoking its function argument with s_1 and the current element x (note: $\uplus$ in $\{x\} \uplus s_2$ denotes a disjoint union). The reduction ends when s_2 becomes empty. The reduction rules for conditionals, `LET` binders, and sequences are standard, and omitted for brevity.

The top-level reduction relation defines the small-step semantics of transactions, and their parallel composition. A transaction comes tagged with an *isolation specification* $\mathbb{I}$, which has two components $\mathbb{I}_e$ and $\mathbb{I}_c$, that dictate the timing and nature of interferences that the transaction can witness, during its execution ($\mathbb{I}_e$), and when it is about to commit ($\mathbb{I}_c$). Formally, $\mathbb{I}_e$ and $\mathbb{I}_c$ are predicates over the (current) transaction-local database state (δ), the state (Δ) of the global database when the transaction last took a step, and the current state (Δ') of the global database. Intuitively, $\Delta' \neq \Delta$ indicates an interference from another concurrent transaction, and the predicates $\mathbb{I}_e$ and $\mathbb{I}_c$ decide if this interference is allowed or not, taking into account the local database state (δ). For instance, as described in §4.1, an SI transaction on PostgreSQL defines $\mathbb{I}$ as follows:

$$\begin{aligned} \mathbb{I}_e\ (\delta, \Delta, \Delta') &= \Delta' = \Delta \\ \mathbb{I}_c\ (\delta, \Delta, \Delta') &= \forall (r \in \delta)(r' \in \Delta).\ r'.\mathtt{id} = r.\mathtt{id} \Rightarrow r' \in \Delta' \end{aligned}$$

This definition dictates that no change to the global database state can be visible to an SI transaction while it executes ($\mathbb{I}_e$), and there should be no concurrent updates to records written by the transaction by other concurrently executing ones ($\mathbb{I}_c$). To simplify the presentation, we use $\mathbb{I}$ instead of $\mathbb{I}_e$ and $\mathbb{I}_c$ when its destructed form is not required.

The reduction of a $\mathtt{TXN}_i\langle\mathbb{I}\rangle\{c\}$ begins by first converting it to its run-time form $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$, where $\delta = \emptyset$, and Δ is the current (global) database. Rule E-Txn reduces $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$ under a database state (Δ'), only if the transaction-body isolation specification ($\mathbb{I}_e$) allows the interference between Δ and Δ'. Rule E-Commit commits the transaction $\mathsf{txn}_i\langle\mathbb{I}, \delta, \Delta\rangle\{c\}$ by flushing its uncommitted records to the database. This is done only if the interference between Δ and Δ' is allowed at the commit point by the isolation specification ($\mathbb{I}_c$). The distinction between $\mathbb{I}_e$ and $\mathbb{I}_c$ allows us to model the snapshot semantics of realistic isolation levels that isolate a transaction from interference during its execution, but expose interferences at the commit point.

Local Context Independence As mentioned previously, our operational semantics does not let a transaction read its own writes. It also does not let a transaction overwrite its own writes, due to the premise $\mathsf{dom}(\delta) \cap \mathsf{dom}(s) = \emptyset$ on the E-DELETE and E-UPDATE rules. We refer to this restriction as *local context independence*. This restriction is easy to relax in the operational semantics and the reasoning framework presented in the next section; our inference procedure described in §2.4, however, has a non-trivial dependence on this assumption. Nonetheless, we have encountered few instances in practice where enforcing local context independence turns out to be a severe restriction. Indeed, all of the transactions we have considered in our benchmarks (e.g., TPC-C) satisfy this assumption.

2.2.1 Isolation Specifications

A distinctive characteristic of our development is that it is parameterized on a weak isolation specification $\mathbb{I}$ that can be instantiated with the declarative characterization of an isolation guarantee or a concurrency control mechanism, regardless of the actual implementation used to realize it. This allows us to model a range of isolation properties that are relevant to the theory and practice of transaction processing systems without appealing to specific implementation artifacts like locks, versions, logs, speculation, etc. A few well-known properties are discussed below:

Unique Ids. As the `new_order` example (§4.1) demonstrates, enforcing global uniqueness of ordered identifiers requires stronger isolation levels than the ones that are default on most databases (e.g., Read Committed). Alternatively, globally unique sequence numbers, regardless of the isolation level, can be requested from a relational database via SQL's `UNIQUE` and `AUTO_INCREMENT` keywords. Our development crucially relies on the uniqueness of record identifiers[7], which are checked locally for

[7]The importance of unique ids is recognized in real-world implementations. For example, MySQL's InnoDB engine automatically adds a 6-byte unique identifier if none exists for a record.

uniqueness by the E-INSERT rule. The global uniqueness of locally unique identifiers can be captured as an isolation property thus:

$$\mathbb{I}_{id}(\delta, \Delta, \Delta') \quad = \quad \forall (r \in \delta).\ r.\texttt{id} \notin \mathsf{dom}(\Delta) \Rightarrow r.\texttt{id} \notin \mathsf{dom}(\Delta')$$

$\mathbb{I}_{id}$ ensures that if the id of a record is globally unique when it is added to a transaction's δ, it remains globally unique until the transaction commits. This would be achieved within our semantic framework by prohibiting the interference from a concurrent transaction that adds the same id. The axiom thus simulates a global counter protected by an exclusive lock without explicitly appealing to an implementation artifact.

Write-Write Conflicts. Databases often employ a combination of concurrency control methods, both optimistic (e.g., speculation and rollback) and pessimistic (e.g., various degrees of locking), to eliminate write-write (ww) conflicts among concurrent transactions. We can specify the absence of such conflicts using our tri-state formulation thus:

$$\mathbb{I}_{ww}(\delta, \Delta, \Delta') \quad = \quad \forall (r' \in \delta)(r \in \Delta).\ r.\texttt{id} = r'.\texttt{id} \Rightarrow r \in \Delta'$$

That is, given a record $r' \in \delta$, if there exists an $r \in \Delta$ with the same id (i.e., r' is an updated version of r), then r must be present unmodified in Δ'. This prevents a concurrent transaction from changing r, thus simulating the behavior of an exclusive lock or a speculative execution that succeeded (Note: a transaction writing to r always changes r because its `txn` field is updated).

Snapshots Almost all major relational databases implement isolation levels that execute transactions against a static snapshot of the database that can be axiomatized thus:

$$\mathbb{I}_{ss}(\delta, \Delta, \Delta') \quad = \quad \Delta' = \Delta$$

Read-Only Transactions. Certain databases implement special privileges for read-only transactions. Read-only behavior can be enforced on a transaction by including the following proposition as part of its isolation invariant:

$$\mathbb{I}_{ro}(\delta, \Delta, \Delta') \quad = \quad \delta = \emptyset$$

In addition to these properties, various specific isolation levels proposed in the database or distributed systems literature, or implemented by commercial vendors can also be specified within this framework:

Read Committed (RC) and Monotonic Atomic View (MAV). RC isolation allows a transaction to witness writes of committed transactions at any point during the transaction's execution. Although it offers only weak isolation guarantees, it nonetheless prevents witnessing *dirty writes* (i.e., writes performed by uncommitted transactions). Monotonic Atomic View (MAV) [7] is an extension to RC that guarantees the continuous visibility of a committed transaction's writes once they become visible in the current transaction. That is, a MAV transaction does not witness *disappearing writes*, which can happen on a weakly consistent machine. Due to the SC nature of our abstract machine (there is always a single global database state Δ; not a vector of states indexed by vector clocks), and our choice to never violate atomicity of a transaction's writes, both RC and MAV are already guaranteed by our semantics. Thus, defining $\mathbb{I}_e$ and $\mathbb{I}_c$ to *true* ensures RC and MAV behavior under our semantics.

Repeatable Read (RR) By definition, multiple reads to a transactional variable in a Repeatable Read transaction are required to return the same value. RR is often implemented (for e.g., in [7, 32]) by executing the transaction against a (conceptual) snapshot of the database, but committing its writes to the actual database. This implementation of RR can be axiomatized as $\mathbb{I}_e = \mathbb{I}_{ss}$ and $\mathbb{I}_c = \mathit{true}$. However, this specification of RR is stronger than the ANSI SQL specification, which requires no more than the invariance of already read records. In particular, ANSI SQL RR allows *phantom reads*, a phenomenon in which a repeated `SELECT` query might return newly inserted records that were not previously returned. This specification is implemented, for e.g., in Microsoft's SQL server, using record-level exclusive read locks, that prevent a record from being modified while it is read by an uncommitted transaction, but which does not prohibit insertion of new records. The ANSI SQL RR specification can be axiomatized in our framework, but it requires a minor extension to our operational semantics to track a transaction's reads. In particular, the records

returned by `SELECT` should be added to the local database δ, but without changing their transaction identifiers (`txn` fields), and flush ($\triangleright$) should only flush the records that bear the current transaction's identifier. With this extension, ANSI SQL RR can be axiomatized thus:

$$\begin{aligned} \mathbb{I}_e(\delta, \Delta, \Delta') &\Leftrightarrow \forall(r \in \delta).r \in \Delta \Rightarrow r \in \Delta' \\ \mathbb{I}_c(\delta, \Delta, \Delta') &\Leftrightarrow true \end{aligned}$$

If a record r belongs to both δ and Δ, then it must be a record written by a different transaction and read by the current transaction (since the current transaction's records are not yet present in Δ). By requiring $r \in \Delta'$, $\mathbb{I}_e$ guarantees the invariance of r, thus the repeatability of the read.

Snapshot Isolation (SI) The concept of executing a transaction against a consistent snapshot of the database was first proposed as Snapshot Isolation in [28]. SI doesn't admit write-write conflicts, and the original proposal, which is implemented in Microsoft SQL Server, required the database to roll-back an SI transaction if conflicts are detected during the commit. This behavior can be axiomatized as $\mathbb{I}_e = \mathbb{I}_{ss}$ (execution against a snapshot), and $\mathbb{I}_c = \mathbb{I}_{ww}$ (avoiding write-write conflicts during the commit). Note that the same axiomatization applies to PostgreSQL's RR, although its implementation (described in Sec. 4.1) differs considerably from the original proposal. Thus, reasoning done for an SI transaction on MS SQL server carries over to PostgreSQL's RR and vice-versa, demonstrating the benefits of reasoning axiomatically about isolation properties.

Serializability (SER) The specification of serializability is straightforward:

$$\begin{aligned} \mathbb{I}_e\ (\delta, \Delta, \Delta') &= \Delta' = \Delta \\ \mathbb{I}_c\ (\delta, \Delta, \Delta') &= \Delta' = \Delta \end{aligned}$$

2.3 The Reasoning Framework

We now describe a proof system that lets us prove the correctness of a $\mathcal{T}$program c w.r.t its high-level consistency conditions I, on an implementation that satisfies

the isolation specifications ($\mathbb{I}$) of its transactions[8]. Our proof system is essentially an adaptation of a rely-guarantee reasoning framework [33] to the setting of weakly isolated database transactions. The primary challenge in the formulation deals with how we relate a transaction's isolation specification ($\mathbb{I}$) to its rely relation (R) that describes the transaction's environment, so that interference is considered only insofar as allowed by the isolation level. Another characteristic of the transaction setting that affects the structure of the proof system is atomicity; we do not permit a transaction's writes to be visible until it commits. In the context of rely-guarantee, this means that the transaction's guarantee (G) should capture the aggregate effect of a transaction, and not its individual writes. While shared memory `atomic` blocks also have the same characteristic, the fact that transactions are weakly-isolated introduces non-trivial complexity. Unlike an `atomic` block, the effect of a transaction is *not* a sequential composition of the effects of its statements because each statement can witness a potentially different version of the state.

2.3.1 The Rely-Guarantee Judgment

Figs. 2.8, 2.9, and 2.10 show an illustrative subset of the rely-guarantee (RG) reasoning rules for $\mathcal{T}$. We define two RG judgments: top-level ($\{I, R\}\ c\ \{G, I\}$), and transaction-local ($\mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\}$). Recall that the standard RG judgment is the quintuple $\{P, R\}\ c\ \{G, Q\}$. Instead of separate P and Q assertions, our top-level judgment uses I as both a pre- and post-condition, because our focus is on verifying that a $\mathcal{T}$ program *preserves* a databases' consistency conditions[9]. A transaction-local RG judgment does not include a guarantee relation because transaction-local effects are not visible outside a transaction. Also, the rely relation ($\mathbb{R}$) of the transaction-local judgment is not the same as the top-level rely relation (R) because it must take

[8]Note the difference between I and $\mathbb{I}$. The former constitute *proof obligations* for the programmer, whereas the latter describes a transaction's *assumptions* about the operational characteristics of the underlying system.

[9]The terms *consistency condition*, *high-level invariant*, and *integrity constraint* are used interchangeably throughout this chapter.

$\boxed{\mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\}}$

RG-SELECT

$$\frac{P(\delta,\Delta) \wedge x = \{r \mid r \in \Delta \wedge [r/y]e\} \Rightarrow P'(\delta,\Delta) \quad \mathbb{R} \vdash \{P'\}\ [c]_i\ \{Q\} \quad \texttt{stable}(\mathbb{R}, P')}{\mathbb{R} \vdash \{P\}\ [\texttt{LET}\ x = \texttt{SELECT}\ \lambda y.e\ \texttt{IN}\ c]_i\ \{Q\}}$$

RG-INSERT

$$\frac{\begin{array}{l}\texttt{stable}(\mathbb{R}, P) \quad \forall \delta, \delta', \Delta, i.\ P(\delta,\Delta) \wedge j \notin \mathsf{dom}(\delta \cup \Delta) \\ \qquad \wedge \delta' = \delta \cup \{\langle x\ \texttt{with id} = j; \texttt{txn} = i; \texttt{del} = \texttt{false}\rangle\} \Rightarrow Q(\delta', \Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\texttt{INSERT}\ x]_i\ \{Q\}}$$

RG-UPDATE

$$\frac{\begin{array}{l}\texttt{stable}(\mathbb{R}, P) \quad \forall \delta, \delta', \Delta.\ P(\delta,\Delta) \wedge \delta' = \delta \cup \{r' \mid \exists (r \in \Delta).[r/x]e_2 \wedge \\ \qquad r' = \langle [r/x]e_1\ \texttt{with id} = r.\texttt{id}; \texttt{txn} = i; \texttt{del} = \texttt{false}\rangle\} \Rightarrow Q(\delta', \Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\texttt{UPDATE}\ \lambda x.e_1\ \lambda x.e_2]_i\ \{Q\}}$$

RG-DELETE

$$\frac{\begin{array}{l}\texttt{stable}(\mathbb{R}, P) \quad \forall \delta, \delta', \Delta.\ P(\delta,\Delta) \wedge \delta' = \delta \cup \{r' \mid \exists (r \in \Delta).\ [r/x]e \\ \qquad \wedge r' = \langle r\ \texttt{with txn} = i; \texttt{del} = \texttt{true}\rangle\} \Rightarrow Q(\delta', \Delta)\end{array}}{\mathbb{R} \vdash \{P\}\ [\texttt{DELETE}\ \lambda x.e]_i\ \{Q\}}$$

Figure 2.8.: $\mathcal{T}$: Transaction-local Rely-Guarantee Judgment Part 1 (SQL Statements)

into account the transaction's isolation specification ($\mathbb{I}$). Intuitively, $\mathbb{R}$ is R modulo $\mathbb{I}$. Recall that a transaction writes to its local database (δ), which is then flushed when the transaction commits. Thus, the guarantee of a transaction depends on the state of its local database at the commit point. The pre- and post-condition assertions (P and Q) in the local judgment facilitate tracking the changes to the transaction-local state, which eventually helps us prove the validity of the transaction's guarantee. Both P and Q are bi-state assertions; they relate transaction-local database state (δ)

RG-Foreach

$$\frac{\texttt{stable}(\mathbb{R}, Q) \quad \texttt{stable}(\mathbb{R}, \psi) \quad \texttt{stable}(\mathbb{R}, P) \quad P \Rightarrow [\emptyset/y]\psi \quad \mathbb{R} \vdash \{\psi \wedge z \in x\}\ [c]_i\ \{Q_c\} \quad Q_c \Rightarrow [y \cup \{z\}/y]\psi \quad [x/y]\psi \Rightarrow Q}{\mathbb{R} \vdash \{P\}\ [\texttt{FOREACH}\ x\ \texttt{DO}\ \lambda y.\lambda z.c]_i\ \{Q\}}$$

RG-Conseq

$$\frac{\{I, R\}\ \texttt{TXN}_i\langle\mathbb{I}\rangle\{c\}\ \{G, I\} \quad \mathbb{I}' \Rightarrow \mathbb{I} \quad R' \subseteq R \quad G \subseteq G' \quad \texttt{stable}(R', \mathbb{I}') \quad \forall \Delta, \Delta'.\ I(\Delta) \wedge G'(\Delta, \Delta') \Rightarrow I(\Delta')}{\{I, R'\}\ \texttt{TXN}_i\langle\mathbb{I}'\rangle\{c\}\ \{G', I\}}$$

Figure 2.9.: $\mathcal{T}$: Transaction-local Rely-Guarantee Judgment Part 2

$\boxed{\{I, R\}\ c\ \{G, I\}}$

RG-Txn

$$\frac{\texttt{stable}(R, \mathbb{I}) \quad \texttt{stable}(R, I) \quad \mathbb{R}_e = R \backslash \mathbb{I}_e \quad \mathbb{R}_c = R \backslash \mathbb{I}_c \quad P(\delta, \Delta) \Leftrightarrow \delta = \emptyset \wedge I(\Delta) \quad \mathbb{R}_e \vdash \{P\}\ c\ \{Q\} \quad \texttt{stable}(\mathbb{R}_c, Q) \quad \forall \delta, \Delta.\ Q(\delta, \Delta) \Rightarrow G(\Delta, \delta \rhd \Delta) \quad \forall \Delta, \Delta'.\ I(\Delta) \wedge G(\Delta, \Delta') \Rightarrow I(\Delta')}{\{I, R\}\ \texttt{TXN}_i\langle\mathbb{I}\rangle\{c\}\ \{G, I\}}$$

Figure 2.10.: $\mathcal{T}$: Top-level Rely-Guarantee Judgment

to the global database state (Δ). Thus, the transaction-local judgment effectively tracks how transaction-local and global states change in relation to each other.

Stability A central feature of a rely-guarantee judgment is a stability condition that requires the validity of an assertion ϕ to be unaffected by interference from other concurrently executing transactions, i.e., the rely relation R. In conventional RG, stability is defined as follows, where σ and σ' denote states:

$$\texttt{stable}(R, \phi) \quad \Leftrightarrow \quad \forall \sigma, \sigma'.\ \phi(\sigma) \wedge R(\sigma, \sigma') \Rightarrow \phi(\sigma')$$

Due to the presence of local and global database states, and the availability of an isolation specification, we use multiple definitions of stability in Fig. 2.10, but they all convey the same intuition as above. In our setting, we only need to prove the stability of an assertion (ϕ) against those environment steps which lead to a global

database state on which the transaction itself can take its next step according to its isolation specification ($\mathbb{I}$).

$$\texttt{stable}(R, \phi) \quad \Leftrightarrow \quad \forall \delta, \Delta, \Delta'. \phi(\delta, \Delta) \wedge R^*(\Delta, \Delta') \wedge \mathbb{I}(\delta, \Delta, \Delta') \Rightarrow \phi(\delta, \Delta')$$

A characteristic of RG reasoning is that stability of an assertion is always proven w.r.t to R, and not R^*, although interference may include multiple environment steps, and R only captures a single step. This is nonetheless sound due to inductive reasoning: if ϕ is preserved by every step of R, then ϕ is preserved by R^*, and vice-versa. However, such reasoning does not extend naturally to isolation-constrained interference because R^* modulo $\mathbb{I}$ is not same as $\mathbb{R}^*$; the former is a transitive relation constrained by $\mathbb{I}$, whereas the latter is the transitive closure of a relation constrained by $\mathbb{I}$. This means, unfortunately, that we cannot directly replace R^* by R in the above condition.

To obtain an equivalent form in our setting, we require an additional condition on the isolation specification, which we call the *stability condition on* $\mathbb{I}$. The condition requires $\mathbb{I}$ to admit the interference of multiple R steps (i.e., $R^*(\Delta, \Delta'')$, for two database states Δ and Δ''), only if it also admits interference of each R step along the way. Formally:

$$\texttt{stable}(R, \mathbb{I}) \quad \Leftrightarrow \quad \forall \delta, \Delta, \Delta', \Delta''.\ \mathbb{I}(\delta, \Delta, \Delta'') \wedge R(\Delta', \Delta'') \Rightarrow \mathbb{I}(\delta, \Delta, \Delta') \wedge \mathbb{I}(\delta, \Delta', \Delta'')$$

It can be easily verified that the above stability condition is satisfied by the isolation axioms from Sec. 2.2.1. For instance, $\mathbb{I}_{ss}$, the snapshot axiom, is stable because if a the state is unmodified between Δ and Δ'', then it is clearly unmodified between Δ and Δ', and also between Δ' and Δ'', where Δ' is an intermediary state. Modifying and restoring the state Δ is not possible because each new commit bears a new transaction id different from the transaction ids (`txn` fields) present in Δ.

The stability condition on $\mathbb{I}$ guarantees that an interference from R^* is admissible only if the interference due to each individual R step is admissible. In other words, it makes isolation-constrained R^* relation equal to the transitive closure of the isolation-constrained R relation. We call R constrained by isolation $\mathbb{I}$ as R modulo $\mathbb{I}$ ($R\backslash\mathbb{I}$; written equivalently as $\mathbb{R}$), which is the following ternary relation:

$$(R\backslash\mathbb{I})(\delta, \Delta, \Delta') \quad \Leftrightarrow \quad R(\Delta, \Delta') \wedge \mathbb{I}(\delta, \Delta, \Delta')$$

It is now enough to prove the stability of an RG assertion ϕ w.r.t $R\backslash\mathbb{I}$:

$$\texttt{stable}((R\backslash\mathbb{I}), \phi) \quad \Leftrightarrow \quad \forall \delta, \Delta, \Delta'.\ \phi(\delta, \Delta) \wedge (R\backslash\mathbb{I})(\delta, \Delta, \Delta') \Rightarrow \phi(\delta, \Delta')$$

This condition often significantly simplifies the form of $R\backslash\mathbb{I}$ irrespective of R. For example, when a transaction is executed against a snapshot of the database (i.e. $\mathbb{I}_{ss}$), $R\backslash\mathbb{I}_{ss}$ will be the identity function, since any non-trivial interference will violate the $\Delta' = \Delta$ condition imposed by $\mathbb{I}_{ss}$.

Rules RG-Txn is the top-level rule that lets us prove a transaction preserves the high-level invariant I when executed under the required isolation as specified by $\mathbb{I}$. It depends on a transaction-local judgment to verify the body (c) of a transaction with id i. The precondition P of c must follow from the fact that the transaction-local database (δ) is initially empty, and the global database satisfies the high-level invariant I. The rely relation ($\mathbb{R}_e$) is obtained from the global rely relation R and the isolation specification $\mathbb{I}_e$ as explained above. Recall that $\mathbb{I}_e$ constrains the global effects visible to the transaction while it is executing but has not yet committed, and P and Q of the transaction-local RG judgment are binary assertions; they relate local and global database states. The local judgment rules require one or both of them to be stable with respect to the constrained rely relation $\mathbb{R}_e$.

For the guarantee G of a transaction to be valid, it must follow from the post-condition Q of the body, provided that Q is stable w.r.t the commit-time interference captured by $\mathbb{R}_c$. $\mathbb{R}_c$, like $\mathbb{R}_e$, is computed as a rely relation modulo isolation, except that commit-time isolation ($\mathbb{I}_c$) is considered. The validity of G is captured by the following implication:

$$\forall \delta, \Delta.\ Q(\delta, \Delta) \Rightarrow G(\Delta, \delta \rhd \Delta)$$

In other words, if Q relates the transaction-local database state (δ) to the state of the global database (Δ) before a transaction commits, then G must relate the states of the global database before and after the commit. The act of commit is captured by the flush action ($\delta \rhd \Delta$). Once we establish the validity of G as a faithful representative of the transaction, we can verify that the transaction preserves the high-level invariant I by checking the stability of I w.r.t G, i.e., $\forall \Delta, \Delta'.\ I(\Delta) \wedge G(\Delta, \Delta') \Rightarrow I(\Delta')$.

The RG-Conseq rule lets us safely weaken the guarantee G, and strengthen the rely R of a transaction. Importantly, it also allows its isolation specification $\mathbb{I}$ to be strengthened (both $\mathbb{I}_e$ and $\mathbb{I}_c$). This means that a transaction proven correct under a weaker isolation level is also correct under a stronger level. Parametricity over the isolation specification $\mathbb{I}$, combined with the ability to strengthen $\mathbb{I}$ as needed, admits a flexible proof strategy to prove database programs correct. For example, programmers can declare isolation requirements of their choice through $\mathbb{I}$, and then prove programs correct assuming the guarantees hold. The soundness of strengthening $\mathbb{I}$ ensures that a program can be safely executed on any system that offers isolation guarantees at least as strong as those assumed.

Salient rules of transaction-local RG judgments are shown in Fig. 2.10. These rules (RG-Update, RG-Select, RG-Delete, and RG-Insert) reflect the structure of the corresponding reduction rule from Fig. 2.7. The rule RG-Foreach defines the RG judgment for a `FOREACH` loop. As is characteristic of loops, the reasoning is pivoted on a loop invariant ψ that needs to be stable w.r.t $\mathbb{R}$. ψ must be implied by P, the pre-condition of `FOREACH`, when no elements have been iterated, i.e, when $y = \emptyset$. The body of the loop can assume the loop invariant, and the fact that z is an element from the set x being iterated, to prove its post-condition Q_c. The operational semantics ensures that z is added to y at the end of the iteration, hence Q_c must imply $[y \cup \{z\}/y]\psi$. When the loop has finished execution, y, the set of iterated items, is the entire set x. Thus $[x/y]\psi$ is true at the end of the loop, from which the post-condition Q must follow. As with the other rules, Q needs to be stable. The rules for conditionals, sequencing etc., are standard, and hence elided.

2.3.2 Semantics and Soundness

We now formalize the semantics of the RG judgments defined in Fig. 2.10, and state their soundness guarantees.

Definition 2.3.1 (Interleaved step and multi-step relations) *Interleaved step relations interleave global and transaction-local reductions with interference as captured by the corresponding rely relations. They are defined thus:*

$$
\begin{array}{lcl}
(c,\Delta) \longrightarrow_R (c',\Delta') & \triangleq & (c,\Delta) \longrightarrow (c',\Delta') \vee (c' = c \wedge R(\Delta,\Delta'))\ [\texttt{global}] \\
([c]_i,\delta,\Delta) \longrightarrow_R ([c']_i,\delta',\Delta') & \triangleq & \Delta \vdash ([c]_i,\delta) \longrightarrow ([c']_i,\delta') \wedge \Delta' = \Delta \\
 & & \vee (c' = c \wedge \delta' = \delta \wedge \mathbb{R}(\delta,\Delta,\Delta'))\ [\texttt{transaction-local}]
\end{array}
$$

An interleaved multi-step relation ($\longrightarrow_R^*$) is the reflexive transitive closure of the interleaved step relation.

Definition 2.3.2 (Semantics of RG judgments) *The semantics of the global and transaction-local RG judgments are defined thus:*

$$
\begin{array}{lcl}
\mathbb{R} \vdash \{P\}\ [c]_i\ \{Q\} & \triangleq & \forall \delta,\delta',\Delta,\Delta'.\ P(\delta,\Delta) \wedge ([c]_i,\delta,\Delta) \longrightarrow_R^* ([\texttt{SKIP}]_i,\delta',\Delta') \Rightarrow Q(\delta',\Delta') \\
\{I,R\}\ c\ \{G,I\} & \triangleq & \forall \Delta.\ I(\Delta) \Rightarrow (\forall \Delta'.\ (c,\Delta) \longrightarrow_R^* (\texttt{SKIP},\Delta') \Rightarrow I(\Delta')) \\
 & & \wedge \mathsf{TxnGuaranteed}(R,G,c,\Delta)
\end{array}
$$

The TxnGuaranteed *predicate used in the semantics of the global RG judgment is defined below:*

$$
\begin{array}{lcl}
\mathsf{TxnGuaranteed}(R,G,c,\Delta) & \triangleq & \forall c', c''\Delta', \Delta''.(c,\Delta) \longrightarrow_R^* (c',\Delta') \wedge (c',\Delta') \longrightarrow (c'',\Delta'') \\
 & & \Rightarrow G(\Delta',\Delta'')
\end{array}
$$

Thus, if $\{I,R\}\ c\ \{G,I\}$ is a valid RG judgment, then (a) every interleaved multi-step reduction of c preserves the database integrity constraint (consistency condition) I, and (b) the effect that every transaction in c has on the database state is captured by G. We can now assert the soundness of the RG judgments in Fig. 2.10 as follows[10]:

Theorem 2.3.1 (Soundness) *The rely-guarantee judgments defined by the rules in Fig. 2.10 are sound with respect to the semantics of Definition 2.3.2.*

[10]Full proofs for the major theorems and lemmas defined in this chapter are available from [34].

PROOF SKETCH. The most important rule is the top-level rule RG-TXN, which proves that a transaction c which begins its execution in global database state satisfying I and encountering interference R while executing under isolation specification $\mathbb{I}$ finishes its execution in a database state also satisfying I, and also guarantees that its commit step satisfies G. The rule uses the transaction-local RG judgment $\mathbb{R}_e \vdash \{P\}\ c\ \{Q\}$. By E-TXN-START, the local and global database states at the start of a transaction satisfy P, and the only challenge is that environment steps in an execution covered by $\mathbb{R}_e \vdash \{P\}\ c\ \{Q\}$ are in $\mathbb{R}_e$, while the top-level judgment requires environment steps in R. We show that it is enough to consider only those environment steps in $\mathbb{R}_e$. First, we use an inductive argument and stability of $\mathbb{I}_e$ ($\texttt{stable}(R, \mathbb{I}_e)$) to show that any execution in which the transaction completes all its steps must always preserve the isolation specification $\mathbb{I}_e$ after every environment step. Intuitively, this is because once $\mathbb{I}_e$ gets broken after some environment step, it will continue to remain broken and the transaction would not be able to proceed (according to E-TXN). Since $\mathbb{R}_e$ contains exactly those environment steps which preserve $\mathbb{I}_e$, the local-level RG judgment can be soundly used, which guarantees that after the transaction finishes its execution, its local state δ and global state Δ will satisfy the assertion Q. Environment steps between the last step of the transaction and its commit step can modify the global state, and hence we also require Q to be stable against R. Again, we use an inductive argument, the stability of $\mathbb{I}_c$, and the fact that the transaction must execute its commit step to show that all environment steps must preserve $\mathbb{I}_c$, and hence it is enough to require $\texttt{stable}(\mathbb{R}_c, Q)$. Q guarantees that the commit step is in G, and G in turn guarantees that after execution, the global database state will obey the invariant I.

2.4 Inference

The rely-guarantee framework presented in the previous section facilitates modular proofs for weakly-isolated transactions, but imposes a non-trivial annotation burden.

$$x, y, \delta, \Delta \in \texttt{Variables} \quad \varphi \in \mathbb{P}^0 \quad \phi \in \mathbb{P}^1$$

$$s \;\coloneqq\; x \mid \delta \mid \Delta \mid \{x \,|\, \varphi\} \mid \mathsf{exists}(\Delta, \phi, s) \mid s_1 \gg\!= \lambda x.s_2 \mid \mathsf{if}\ \varphi\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid s_1 \cup s_2$$

Figure 2.11.: Syntax of the set language $\mathcal{S}$

In particular, it requires each statement (c) of the transaction to be annotated with a stable pre- (P) and post-condition (Q), and loops to be annotated with stable inductive invariants (ψ). While weakest pre-condition style predicate transformers can help in inferring intermediate assertions for regular statements, loop invariant inference remains challenging, even for the simple form of loops considered here. As an alternative, we present an inference algorithm based on state transformers that alleviates this burden. The idea is to infer the logical effect that each statement has on the transaction-local database state δ (i.e., how it transforms δ), and compose multiple such effects together to describe the effect of the transaction as a whole. Importantly, this approach generalizes to loops, where the effect of a loop can be inferred as a well-defined function of the effect of its body, thanks to certain pleasant properties enjoyed by the database programs modeled by our core language. Interpreting database semantics as functional transformations on sets (described in terms of their logical effects) enables an inference mechanism that can leverage off-the-shelf SMT solvers for automated verification.

At the core of our approach is a simple language ($\mathcal{S}$) to express set transformations (see Fig. 2.11). The language admits set expressions that include variables (x), literals of the form $\{x \,|\, \varphi\}$ where φ is a propositional (quantifier-free) formula on x, a restricted form of existential quantification that binds a set Δ satisfying proposition ϕ in a set expression s, a monadic composition of two set expressions (s_1 and s_2) composed using a bind ($\gg\!=$) operation, a conditional set expression where the condition is a propositional formula, and a union of two set expressions. Symbols δ and Δ are also variables in $\mathcal{S}$, but are used to denote local and database states

(also represented as sets), respectively. Constant sets can be written using set literal expressions. For example, the set $\{1, 2\}$ can be written as $\{x \mid x = 1 \vee x = 2\}$. The language is carefully chosen to be expressive enough to capture the semantics of $\mathcal{T}$ statements (as well as SQL operations more generally), yet simple enough to have a semantics-preserving translation amenable for automated verification.

Fig. 2.12 shows the syntax-directed state transformer inference rules for $\mathcal{T}$ commands inside a transaction $\texttt{TXN}_i$. The rules compute, for each command c, a (meta) function F that returns a set of records as an expression in $\mathcal{S}$, given a global database Δ. Intuitively, $\mathsf{F}(\Delta)$ abstracts the set of records added to the local database δ as a result of executing c under Δ (i.e., $\Delta \vdash ([c]_i, \delta) \longrightarrow_R^* ([\,\texttt{SKIP}\,]_i, \delta \cup \mathsf{F}(\Delta))$)[11]. Note that the function F we call state transformer here is actually the *effect* part of the state transformer introduced in Sec. 4.1, which is a function T of form $\lambda(\delta, \Delta).\ \delta \cup \mathsf{F}(\Delta)$. Nonetheless, for simplicity, we will continue to refer to F as state transformer. Since the execution is subject to isolation-constrained interference, the inference judgment depends on the isolation-constrained rely relation $\mathbb{R}$, which is used to enforce the stability of the state transformer F. Recall that $\mathbb{R}$ is a tri-state rely relation over δ, Δ and Δ', that admits an interference from Δ and Δ' depending on the local database state δ. Thus, the stability of the state transformer F of c with respect to $\mathbb{R}$ needs to take into account the (possible) prior state of the local database δ, which depends on the context (sequence of previous commands) of c, and computed by the corresponding state transformer $\mathsf{F}_{\mathsf{ctxt}}$. Thus, the semantics of the state transformer can be understood in terms of the RG judgment as following (formalized as Theorem 2.4.1 in Sec. 2.4.1):

$$\mathbb{R} \vdash \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)\}\ [c]_i\ \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)\}$$

In the above RG judgment, let P denote the pre-condition $\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$, and let Q denote the post-condition $\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$. The stability condition on the state transformer F can be derived from the stability condition on Q. Observe that for Q to be stable, $\mathsf{F}_{\mathsf{ctxt}}(\Delta') \cup \mathsf{F}(\Delta')$ must be equal to $\mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$, where

[11]Recall that the operational semantics treats deletion of records as the addition of the deleted record with its `del` field set to true in the local store.

$\boxed{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}$

$$\frac{}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{INSERT}\ x \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ [\![\mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \{r \mid r = \{\langle x\ \texttt{with del} = \texttt{false};\ \texttt{txn} = i\rangle\}]]\!]_{\langle\mathbb{R},I\rangle}}$$

$$\frac{G = \lambda r.\ \mathsf{if}\ [r/x]e_2\ \mathsf{then}\ \{r' \mid r' = \langle [r/x]e_1\ \texttt{with id} = r.\texttt{id};\ \texttt{del} = r.\texttt{del};\ \texttt{txn} = i\rangle\}\ \mathsf{else}\ \emptyset}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{UPDATE}\ \lambda x.e_1\ \lambda x.e_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ [\![\mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg\!= G]]\!]_{\langle\mathbb{R},I\rangle}}$$

$$\frac{G = \lambda r.\ \mathsf{if}\ [r/x]e\ \mathsf{then}\ \{r' \mid r' = \langle r\ \texttt{with del} = \texttt{true};\ \texttt{txn} = i\rangle\}\ \mathsf{else}\ \emptyset}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{DELETE}\ \lambda x.e \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ [\![\mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg\!= G]]\!]_{\langle\mathbb{R},I\rangle}}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{LET}\ x = e\ \texttt{IN}\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ \lambda(\Delta).\ [e/x]\,\mathsf{F}(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F} \qquad G = \lambda r.\ \mathsf{if}\ [r/x]e\ \mathsf{then}\ \{r' \mid r' = r\}\ \mathsf{else}\ \emptyset \quad \mathsf{F}' = [\![\mathsf{F}_{\mathsf{ctxt}}[\lambda(\Delta).\ \Delta \gg\!= G]]\!]_{\langle\mathbb{R},I\rangle}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{LET}\ y = \texttt{SELECT}\ \lambda x.e\ \texttt{IN}\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ \lambda(\Delta).\ [\mathsf{F}'(\Delta)/y]\,\mathsf{F}(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_1 \quad \mathsf{F}_{\mathsf{ctxt}} \vdash c_2 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_2}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{IF}\ e\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ \lambda(\Delta).\ \mathsf{if}\ e\ \mathsf{then}\ \mathsf{F}_1(\Delta)\ \mathsf{else}\ \mathsf{F}_2(\Delta)}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_1 \quad \mathsf{F}_{\mathsf{ctxt}} \cup \mathsf{F}_1 \vdash c_2 \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}_2}{\mathsf{F}_{\mathsf{ctxt}} \vdash c_1; c_2 \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ \mathsf{F}_1 \cup \mathsf{F}_2}$$

$$\frac{\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}}{\mathsf{F}_{\mathsf{ctxt}} \vdash \texttt{FOREACH}\ x\ \texttt{DO}\ \lambda y.\lambda z.\ c \ \Longrightarrow_{\langle i,\mathbb{R},I\rangle}\ \lambda(\Delta).\ x \gg\!= (\lambda z.\ \mathsf{F}(\Delta))}$$

Figure 2.12.: $\mathcal{T}$: State transformer semantics.

Δ and Δ' are related by R (ignore $\mathbb{I}$ for the moment). Assuming that P is stable, $\mathsf{F}_{\mathsf{ctxt}}(\Delta') = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$ is already given, leaving $\mathsf{F}(\Delta') = \mathsf{F}(\Delta)$ to be enforced. Thus, the stability of F in in the context of $\mathsf{F}_{\mathsf{ctxt}}$ (written $\mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]$) is defined as following:

$$\texttt{stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]) \quad \Leftrightarrow \quad \forall \Delta, \Delta', \overline{\nu}.\ \mathbb{R}(\mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta), \Delta, \Delta') \Rightarrow \mathsf{F}(\Delta) = \mathsf{F}(\Delta')$$

where $\overline{\nu}$ are the variables that occur free in F; this is possible because of how the inference rules are structured. The equality in $\mathcal{S}$ translates to equivalence in first-order logic, as we describe later. In the inference rules, stability is enforced constructively by a meta-function $[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$, which accepts a transformer F (in its context $\mathsf{F}_{\mathsf{ctxt}}$) and returns a new transformer that is guaranteed to be stable under $\mathbb{R}$. $[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$ achieves the stability guarantee by abstracting away the bound global state (Δ) in an unstable F to an existentially bound Δ' as described below:

$$\begin{array}{rcll} [\![\mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]]\!]_{\langle \mathbb{R}, I \rangle} & = & \mathsf{F} & \texttt{if stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}}[\mathsf{F}]). \\ & = & \lambda(\Delta).\ \mathsf{exists}(\Delta', I(\Delta'), \mathsf{F}(\Delta')) & \texttt{otherwise.} \quad \Delta' \texttt{ is a fresh name.} \end{array}$$

Observe that when F is not stable, $[\![\mathsf{F}]\!]_{\langle \mathbb{R}, I \rangle}$ returns a transformer F' that simply ignores its Δ argument in favor of a generic Δ', making F' trivially stable. It is safe to assume $I(\Delta')$ because all verified transactions must preserve the invariant, and hence only valid database states will ever be witnessed. From the perspective of RG reasoning, $[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$ effectively weakens the post-condition of a statement, as done by the RG-Conseq rule for transaction-bound commands. The weakening semantics chosen by $[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$, while being simple, is nonetheless useful because of the $I(\Delta')$ assumption imposed on an existentially bound Δ'. The example in Fig. 2.13 demonstrates. Here, an `add_interest` transaction adds a positive interest (determined by `pc`) to the balance of a bank account, which is required to be non-negative ($I(\Delta) \Leftrightarrow \forall (r \in \Delta).\ r.\texttt{bal} \geq 0$). The transaction starts by issuing a `select1` query, whose transformer F is essentially a singleton set containing a record r whose id is `acc_id` (i.e., $\mathsf{F}(\Delta) = \{\, r \mid r \in \Delta \wedge r.\texttt{id} = \texttt{acc\_id} \,\}$). However, F is unstable because $\mathsf{F}(\Delta')$ may not be the same set as $\mathsf{F}(\Delta)$ when $\Delta' \neq \Delta$. A record $r \in \Delta$ whose `id` = `acc_id` may have its balance updated by a concurrent `withdraw` or `deposit`

```
let add_interest acc_id pc = atomically_do @@ fun () ->
  let a = SQL.select1 BankAccount (fun acc -> acc.id = acc_id) in
  let y = a.bal + pc*a.bal in
  SQL.update BankAccount (fun acc -> {acc with bal = acc.bal + y})
                         (fun acc -> acc.id = acc_id)
```

Figure 2.13.: A transaction that deposits an interest to a bank account.

transaction in Δ', making the record in Δ' different from the record in Δ. Hence the stability check fails. Fortunately, the weakening operator ($[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$) allows us to weaken the effect to $\mathsf{exists}(\Delta, I(\Delta), \{r \mid r \in \Delta \wedge r.\,\mathtt{id} = \mathtt{acc\_id}\})$, which effectively asserts that the `select1` query returns a record with `id` = `acc_id` from *some* database state that satisfies the non-negative balance invariant I. This weakened assertion is nonetheless enough to deduce that $\mathtt{a.bal} \geq 0$, and subsequently prove that $\mathtt{a.bal} + \mathtt{pc} * \mathtt{a.bal} \geq 0$, allowing us to verify the `add_interest` transaction.

The state transformer rules, like the earlier RG rules, closely follow the corresponding reduction rules in Fig. 2.7, except that their language of expression is $\mathcal{S}$. For instance, while the reduction rule for `UPDATE` declaratively specifies the set of updated records, the state transformer rule uses $\mathcal{S}$'s bind operation to *compute* the set. Other SQL rules do likewise. The rules for `LET` binders, conditionals, and sequences compose the effects inferred for their subcommands. Thus, the effect of a sequence of commands $c_1; c_2$ is the union of effects F_1 and F_2 of c_1 and c_2, respectively, except that F_2 is computed in a context that includes F_1 (we write $\mathsf{F}_1 \cup \mathsf{F}_2$ as a shorthand for $\lambda(\Delta).\ \mathsf{F}_1(\Delta) \cup \mathsf{F}_2(\Delta)$). The inference rule for `FOREACH` takes advantage of the $\mathcal{S}$'s bind operator to lift the effect inferred for the loop body to the level of the loop. Since records added to δ in each iteration of `FOREACH` are independent of the previous iteration (recall that we make a local context independence assumption about database programs; Sec. 2.2), sequential composition of the effects of different iterations is the same as their parallel composition. Since the loop body is executed

once per each $z \in x$, the effect of the the loop is a union of effects (F) for all $z \in x$, all applied to the same state (Δ). That is, $\mathsf{F}_{loop}(\Delta) = \bigcup_{z \in x} \mathsf{F}_{body}(\Delta)$. From the definition of the set monad's bind operator, $\mathsf{F}_{loop}(\Delta) = x \gg\!= (\lambda z.\ F_{body}(\Delta))$, which mirrors the definition of the rule.

2.4.1 Soundness of Inference

We now formally state the correspondence between the inference rules given above and the RG judgment of §2.3:

Theorem 2.4.1 *For all i,R,I,c,$\mathsf{F}_{\mathsf{ctxt}}$, F, if $\mathtt{stable}(\mathbb{R}, I)$, $\mathtt{stable}(\mathbb{R}, \mathsf{F}_{\mathsf{ctxt}})$ and $\mathsf{F}_{\mathsf{ctxt}} \vdash c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}$, then:*

$$\mathbb{R} \vdash \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \wedge I(\Delta)\}\ [c]_i\ \{\lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)\}$$

Proof Sketch. The proof follows by structural induction on c. Let $P = \lambda(\delta, \Delta).\ \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \wedge I(\Delta)$ and $Q = \lambda(\delta, \Delta).\delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup \mathsf{F}(\Delta)$. The base cases correspond to `INSERT`, `UPDATE` and `DELETE` statements, where the proof is straightforward. The proofs for `SELECT`, sequencing, and conditionals use the inductive hypothesis to infer the RG-judgments present in the premises of their corresponding RG-rules. The interesting case is the `FOREACH` statement, for which we use the loop invariant $\psi(\delta, \Delta) \Leftrightarrow \delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta) \cup (y \gg\!= (\lambda z.\ \mathsf{F}(\Delta)))$, (where assuming that c is the body of the loop, $c \Longrightarrow_{\langle i,\mathbb{R},I\rangle} \mathsf{F}$) to prove all the premises of RG-Foreach. Using the same notation as the rule RG-Foreach, y refers to the records already processed in previous iterations of the loop, while z refers to the record being processed in the current iteration. At the beginning of the loop $[\phi/y]\psi(\delta, \Delta)$ just reduces to $\delta = \mathsf{F}_{\mathsf{ctxt}}(\Delta)$ which is implied by the pre-condition P. From the inductive hypothesis, we can infer that each iteration corresponds to the application of F. Since all iterations are assumed to be independent of each other, and z is bound to a record in x for each iteration, we conclude that at the end of every iteration, the loop invariant $[y \cup \{z\}/y]\psi$ will be satisfied.

$$
\begin{array}{lcll}
[\![\delta \mid \Delta \mid \ldots]\!]_{\langle\overline{\nu}\rangle} & = & (\top, \lambda(\overline{v}, r).\, r \in \delta) \mid (\top, \lambda(\overline{v}, r).\, r \in \Delta) \mid \ldots & |\overline{v}| = |\overline{\nu}| \\
[\![\{x \,|\, \varphi\}]\!]_{\langle\overline{\nu}\rangle} & = & (\top, \lambda(\overline{v}, r).\, [r/x]\varphi) & |\overline{v}| = |\overline{\nu}| \\
[\![\mathsf{if}\ \varphi\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2,\ \lambda(\overline{v}, r).\ \texttt{if}\ \varphi\ \texttt{then}\ \mathsf{G}_1(\overline{v}, r) & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \qquad \texttt{else}\ \mathsf{G}_2(\overline{v}, r) & (\phi_2, \mathsf{G}_2) = [\![s_2]\!]_{\langle\overline{\nu}\rangle} \\
[\![s_1 \cup s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2, & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \quad \lambda(\overline{v}, r).\, \mathsf{G}_1(\overline{v}, r) \vee \mathsf{G}_2(\overline{v}, r)) & (\phi_2, \mathsf{G}_2) = [\![s_2]\!]_{\langle\overline{\nu}\rangle} \\
[\![s_1 \gg\!= \lambda x.s_2]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_1 \wedge \phi_2 \wedge \forall\overline{\nu}.\forall a.\forall b.\ \pi_1(\overline{\nu}) \Leftrightarrow & \mathsf{fresh}(\pi_1)\ \ \mathsf{fresh}(\pi_2)\ \ \mathsf{fresh}(g) \\
 & & \quad \mathsf{G}_1(\overline{\nu}, a) \wedge \mathsf{G}_2(\overline{\nu}, a, b) \Rightarrow g(\overline{\nu}, b) & (\phi_1, \mathsf{G}_1) = [\![s_1]\!]_{\langle\overline{\nu}\rangle} \\
 & & \quad \wedge\forall\overline{\nu}.\forall b.\exists a.\ \pi_2(\overline{\nu}) \Leftrightarrow & (\phi_2, \mathsf{G}_2) = [\![[a/x]s_2]\!]_{\langle\overline{\nu},a\rangle} \\
 & & \quad g(\overline{\nu}, b) \Rightarrow \mathsf{G}_1(\overline{\nu}, a) \wedge \mathsf{G}_2(\overline{\nu}, a, b), & \mathsf{fresh}(a)\ \ \mathsf{fresh}(b) \\
 & & \lambda(\overline{v}, r).\, \pi_1(\overline{v}) \wedge \pi_2(\overline{v}) \wedge g(\overline{v}, r)) & |\overline{v}| = |\overline{\nu}| \\
[\![\mathsf{exists}(\Delta, \phi, s)]\!]_{\langle\overline{\nu}\rangle} & = & (\phi_s \wedge \forall\overline{\nu}.\forall a.\forall b.\ f(\overline{\nu}, a) \wedge f(\overline{\nu}, b) \Rightarrow a = b & \mathsf{fresh}(a)\ \ \mathsf{fresh}(b) \\
 & & \quad \wedge\forall\overline{\nu}.\exists a.\ f(\overline{\nu}, a) & \mathsf{fresh}(f) \\
 & & \quad \wedge\forall\overline{\nu}.\forall a.\forall b.\ \pi(\overline{\nu}) \Leftrightarrow f(\overline{\nu}, a) \wedge [a/\Delta]\phi & \mathsf{fresh}(\pi)\ \ \mathsf{fresh}(g) \\
 & & \quad \wedge g(\overline{\nu}, b) = \mathsf{G}_s(\overline{\nu}, b), & (\phi_s, \mathsf{G}_s) = [\![[a/\Delta]s]\!]_{\langle\overline{\nu}\rangle} \\
 & & \lambda(\overline{v}, r).\, \pi(\overline{v}) \wedge g(\overline{v}, r)) & |\overline{v}| = |\overline{\nu}|
\end{array}
$$

Figure 2.14.: Encoding $\mathcal{S}$ in first-order logic

2.4.2 From $\mathcal{S}$ to the First-Order Logic

Theorem 2.4.1 lets us replace the local judgment of the RG-Txn rule (Fig. 2.10) by a state transformer inference judgment. The soundness of a transaction's guarantee

can now be established w.r.t the effect F of the body. The RG-Txn rule so updated is shown below ($\mathsf{F}_\emptyset = \lambda(\Delta).\ \emptyset$ denotes an empty context):

$$\frac{\texttt{stable}(R,\mathbb{I}) \quad \texttt{stable}(R,I) \quad \mathbb{R}_e = R\backslash\mathbb{I}_e \quad \mathbb{R}_c = R\backslash\mathbb{I}_c \quad \mathsf{F}_\emptyset \vdash c \Longrightarrow_{\langle i,\mathbb{R}_e,I\rangle} \mathsf{F} \qquad \texttt{stable}(\mathbb{R}_c, \mathsf{F}_\emptyset[\mathsf{F}]) \quad \forall\Delta.\ G(\Delta,\mathsf{F}(\Delta)) \quad \forall\Delta,\Delta'.\ I(\Delta)\wedge G(\Delta,\Delta') \Rightarrow I(\Delta')}{\{I,R\}\ \texttt{TXN}_i\langle\mathbb{I}\rangle\{c\}\ \{G,I\}}$$

Automating the application of the RG-Txn rule for a transaction requires automating the multiple implication checks in the premise. While R, G, $\mathbb{I}$ and I are formulas in first-order logic (FOL) with a relatively simple structure, F is an expression in the set language $\mathcal{S}$ (Fig. 2.11) with a possibly complex structure. Fortunately, however, there exists a semantics-preserving translation from $\mathcal{S}$ to a restricted subset of first-order logic (FOL) that lends itself to automatic reasoning.

The algorithm ($[\![\cdot]\!]_{\langle\cdot\rangle}$) shown in Fig. 2.14 translates an $\mathcal{S}$ expression (s) to FOL. The translation is based on encoding a set of element type T as a unary predicate on T. The predicate is represented as a meta function that accepts an $x : T$ and returns a quantifier-free proposition that evaluates to true ($\top$) if and only if x is present in the set. Alternatively, the translation may also encode the set as a predicate in the logic itself, in which case a quantified proposition constraining the predicate is also generated. For instance, consider the set $\{1, 2\}$. The predicate describing the set can be encoded as the function $\lambda\upsilon.\upsilon = 1 \vee \upsilon = 2$, with no further constraints, or it can be encoded as the function $\lambda\upsilon.g(\upsilon)$ with an associated constraint, $\phi \in \mathbb{P}^1 = \forall\nu.\ g(\nu) \Leftrightarrow \nu = 1 \vee \nu = 2$, defining the uninterpreted predicate g. The translation adopts one or the other approach, depending on the need. For uniformity, we consider the encoding of a set as pair (ϕ,G), where G is a meta function, and ϕ is a FOL formula constraining any uninterpreted predicates used in G.

Due to the presence of bind ($\gg\!=$) in $\mathcal{S}$, a set expression s may contain free variables introduced by an enclosing binder. For instance, consider the $\mathcal{S}$ expression $s_1 \gg\!= (\lambda x.\{y \mid y = x+1\})$, where s_1 is an integer set (expression). The subexpression $\{y \mid y = x + 1\}$ (call it s_2) contains x as a free variable. In such cases, the predicate associated with the subexpression should also be indexed by its free variables so that

a unique set exists for each instantiation of the free variables. Thus, the predicate (G) associated with the subexpression from the above example should be $\lambda\upsilon_1.\lambda\upsilon_2.\ \upsilon_2 = \upsilon_1 + 1$, so that the set $\mathsf{G}\ x_1$ is different from the set $\mathsf{G}\ x_2$ for distinct $x_1, x_2 \in s_1$. Intuitively, the bind expression $s_1 \gg\!= (\lambda x.\{y \mid y = x + 1\})$ denotes the set $\bigcup_{x \in s_1} \mathsf{G}\ x$.

The translation algorithm (Fig. 2.14) takes free variables into account. Given a set expression $s \in \mathcal{S}$, whose (possible) free variables are $\overline{\nu}$ in the order of their introduction (top-most binder first), $[\![s]\!]_{\langle\overline{\nu}\rangle}$ returns the encoding of s as (ϕ, G). The meta-function G is a predicate indexed by the (possible) free variables of s, and thus its arity is $|\overline{\nu}| + 1$. Note that $\overline{\nu}$ is only a sequence of variables introduced by the enclosing binders of s, and not all may actually occur free in s. Nonetheless, its predicate G is always indexed by $|\overline{\nu}|$ free variables for uniformity. The translation encodes database state as an uninterpreted sort. Considering that the state is actually a set of records, we define an uninterpreted relation "$\in$" to relate records and states. Thus, a variable set expression Δ denoting a database state is encoded as the predicate $\lambda(\overline{\upsilon}, r).\ r \in \Delta$, where $|\overline{\upsilon}| = |\overline{\nu}|$ (predicates are uncurried for simplicity; $\overline{\upsilon}$ is a comma-separated sequence; $r \notin \mathcal{S}$ is a special variable). The constraints associated with the encoding of a state are trivial (denoted $\top$). The set literal expression $\{x \mid \varphi\}$ is encoded straightforwardly. The conditional set expression is encoded as an if-then-else predicate in FOL, where the predicates on true and false branches are computed from the set subexpressions s_1 and s_2, respectively. The conjunction of constraints ϕ_1 and ϕ_2, from $[\![s_1]\!]_{\langle\overline{\nu}\rangle}$ and $[\![s_2]\!]_{\langle\overline{\nu}\rangle}$ (resp.), is propagated upwards as the constraint of the conditional expression. A set union expression is encoded similarly.

The first-order encoding of a bind expression describes the semantics of the set monad's bind operator in FOL. Let s_1 be a set, and let f be a function that maps each variable in s_1 to a new set. Then, $s_2 = s_1 \gg\!= f$ if and only if for all $y \in s_2$, there exists an $x \in s_1$ such that $y = f(x)$, and forall $x \in s_1$, $f(x) \in s_2$. The encoding essentially adds new constraints to this effect. The translation first encodes s_1 and s_2 to obtain (ϕ_1, G_1) and (ϕ_2, G_2), respectively. Since s_2 is under a new binder that binds x, the free variable sequence of s_2 is $\overline{\nu}, x$. In the interest of hygiene, we substitute a

fresh a for x, making the sequence $\overline{\nu}, a$. The set s is encoded as a new uninterpreted predicate g indexed by s's free variables ($\overline{\nu}$). Since the set denoted by g is the result of the bind $s_1 \gg\!= \lambda x.s_2$, first-order constraints defining the bind operation (as described above) are generated. The constraints relate the predicates G_1 and G_2, representing s_1 and s_2 (resp.), to the uninterpreted predicate g that represents s. The constraints are assigned names (π_1 and π_2) to give them an easy handle.

The first-order encoding of the $\mathsf{exists}(\Delta, \phi, s)$ expression essentially Skolemizes the existential. Skolemizing is the process of substituting an existentially bound x in $\phi_x \in \mathbb{P}^1$ with $f(\overline{\nu})$, where f is a fresh uninterpreted function (called the Skolem function), and $\overline{\nu}$ are the free variables in ϕ_x bound by enclosing universal quantifiers. Due to the decidability restrictions (Sec. 2.4.3), the only uninterpreted functions we admit in our logic are boolean (i.e., predicates/relations). Consequently, we cannot define the Skolem function f directly. Instead, we define it via an uninterpreted relation, by explicitly asserting the function property:

$$(\forall \overline{\nu}.\forall a.\forall b.\ f(\overline{\nu}, a) \wedge f(\overline{\nu}, b) \Rightarrow a = b) \quad \wedge \quad (\forall \overline{\nu}.\exists a.f(\overline{\nu}, a))$$

We then replace the existentially bound Δ with a new universally bound a in ϕ and s, such that $f(\overline{\nu}, a)$ holds, before encoding the existentially bound s.

Example Let us reconsider the TPC-C `new_order` transaction from Sec. 4.1. Recall that the state transformer (T) for the `foreach` loop shown in Fig. 2.4 is (`k1` , `k2` , and `k3` are constants):

$$\lambda(\delta, \Delta).\ \delta \cup \texttt{item\_reqs} \gg\!= (\lambda\, \texttt{item\_req}\,.\ \mathsf{F}_U(\Delta) \cup \mathsf{F}_I(\Delta))$$

where:

$$\begin{aligned}
\mathsf{F}_U \quad = \quad & \lambda(\Delta).\ \Delta \gg\!= (\lambda s.\mathsf{if}\ \mathsf{table}(s) = \texttt{Stock} \wedge s.\texttt{s_i_id} = \texttt{item_req.ol_i_id} \\
& \qquad \mathsf{then}\ \{\langle \texttt{s_i_id} = s.\texttt{s_i_id};\ \texttt{s_d_id} = s.\texttt{s_d_id};\ \texttt{s_qty} = \texttt{k1} \rangle\} \\
& \qquad \mathsf{else}\ \emptyset) \\
\mathsf{F}_I \quad = \quad & \lambda(\Delta).\ \{\langle \texttt{ol_o_id} = \texttt{k2};\ \texttt{ol_d_id} = \texttt{k3};\ \texttt{ol_i_id} = \texttt{item_req.ol_i_id}; \\
& \qquad \texttt{ol_qty} = \texttt{item_req.ol_qty} \rangle\}
\end{aligned}$$

For any Δ, $\mathsf{F}_U(\Delta)$ and $\mathsf{F}_I(\Delta)$ are expressions in $\mathcal{S}$, so can be translated to FOL by the encoding algorithm in Fig. 2.14. Since the iteration variable `item_req` occurs free in these expressions, the appropriate application of the encoding algorithm is $[\![\mathsf{F}_U(\Delta)]\!]_{\langle \texttt{item\_req} \rangle}$ and $[\![\mathsf{F}_I(\Delta)]\!]_{\langle \texttt{item\_req} \rangle}$, which results in (ϕ_U, G_U) and (ϕ_I, G_I), respectively, where ϕ_U, ϕ_I, G_U, G_I are as shown below:

$$
\begin{aligned}
\phi_U \;=\;& \forall\, \texttt{item\_req}.\forall s.\forall s'.\ \pi_1(\,\texttt{item\_req}\,) \Leftrightarrow \\
& \quad (s \in \Delta) \wedge (\ \texttt{if}\ \mathsf{table}(s) = \texttt{Stock} \wedge s.\texttt{s\_i\_id} = \texttt{item\_req.ol\_i\_id} \\
& \qquad\qquad \texttt{then}\ s' = \langle\, \texttt{s\_i\_id} = s.\texttt{s\_i\_id};\ \texttt{s\_d\_id} = s.\texttt{s\_d\_id};\ \texttt{s\_qty} = \texttt{k1}\,\rangle \\
& \qquad\qquad \texttt{else}\ \bot) \Rightarrow g_0(\,\texttt{item\_req}, s') \\
& \wedge \forall\, \texttt{item\_req}.\forall s'.\exists s.\ \pi_2(\,\texttt{item\_req}\,) \Leftrightarrow \\
& \quad g_0(\,\texttt{item\_req}, s') \Rightarrow s \in \Delta \wedge\ \texttt{if}\ \mathsf{table}(s) = \texttt{Stock} \wedge s.\texttt{s\_i\_id} = \texttt{item\_req.ol\_i\_id} \\
& \qquad\qquad \texttt{then}\ s' = \langle\, \texttt{s\_i\_id} = s.\texttt{s\_i\_id};\ \texttt{s\_d\_id} = s.\texttt{s\_d\_id};\ \texttt{s\_qty} = \texttt{k1}\,\rangle \\
& \qquad\qquad \texttt{else}\ \bot \\
\mathsf{G}_U \;=\;& \lambda(\,\texttt{item\_req}, r).\ \pi_1(\,\texttt{item\_req}\,) \wedge \pi_2(\,\texttt{item\_req}\,) \wedge g_0(\,\texttt{item\_req}, r) \\
\phi_I \;=\;& \top \\
\mathsf{G}_I \;=\;& \lambda(\,\texttt{item\_req}, r).\ r = \langle\, \texttt{ol\_o\_id} = \texttt{k2};\ \texttt{ol\_d\_id} = \texttt{k3};\ \texttt{ol\_i\_id} = \texttt{item\_req.ol\_i\_id}; \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{ol\_qty} = \texttt{item\_req.ol\_qty}\,\rangle
\end{aligned}
$$

Since the transformer (T) of the `foreach` loop is not nested does not contain any free iteration variables, the appropriate application of the encoding algorithm is $[\![\mathsf{T}(\delta, \Delta)]\!]_{\langle\emptyset\rangle}$, which results in the $(\phi_I \wedge \phi_U \wedge \phi_1 \wedge \phi_2, \mathsf{G})$, where ϕ_1, ϕ_2, and G are as defined below:

$$
\begin{aligned}
\phi_1 \;=\;& \forall\, \texttt{item\_req}.\forall s.\ \pi_3 \Leftrightarrow \texttt{item\_req} \in \texttt{item\_reqs} \wedge \mathsf{G}_U(\,\texttt{item\_req}, s') \vee \mathsf{G}_I(\,\texttt{item\_req}, s') \Rightarrow g_1(s) \\
\phi_2 \;=\;& \forall s.\exists\, \texttt{item\_req}.\ \pi_4 \Leftrightarrow g_1(s) \Rightarrow \texttt{item\_req} \in \texttt{item\_reqs} \wedge \mathsf{G}_U(\,\texttt{item\_req}, s') \vee \mathsf{G}_I(\,\texttt{item\_req}, s') \\
\mathsf{G} \;=\;& \lambda(r).\ \pi_3 \wedge \pi_4 \wedge g_1(r)
\end{aligned}
$$

2.4.3 Decidability

Observe that the encoding shown in Fig. 2.14 maps to a fragment of FOL that satisfies the following syntactic properties:

- All function symbols, modulo those that are drawn from $\mathbb{P}^0$ and $\mathbb{P}^1$, are uninterpreted and boolean.
- All quantification is first-order; second-order objects, such as sets and functions, are never quantified.
- Quantifiers appear only at the prenex position, i.e., at the beginning of a quantified formula.

The simple syntactic structure of the fragment already makes is amenable for automatic reasoning via an off-the-shelf SMT solver, such as Z3. The decidability of this fragment, however, is more subtle and discussed below.

Consider a set expression s with no free variables (i.e., $\overline{\nu} = \emptyset$, like $\mathsf{T}(\delta, \Delta)$ from the above example). Let $(\phi, G) = [\![s]\!]_{\langle\emptyset\rangle}$. Note that ϕ is a conjunction of (a). ϕ_i's, where each ϕ_i results from encoding a subexpression s_i of s, and (b). a ϕ_s, resulting from encoding s itself (i.e., its top-level expression). From Fig. 2.14, it is clear that ϕ_s is either $\top$ (for the first four cases), or it is a prenex-quantified formula, where quantification is either $\forall^2$, or $\exists$, or $\forall\exists$. Generalizing this observation, for a set expression s with $|\overline{\nu}|$ free variables, ϕ_s, if quantified, is a prenex-quantified formula, where quantification assumes one among the forms of $\forall^{|\overline{\nu}|+2}$, or $\forall^{|\overline{\nu}|}\exists$, or $\forall^{|\overline{\nu}|+1}\exists$. In other words, the number of $\forall$ quantifiers preceding an $\exists$ quantifier is utmost one more than the number of free variables ($\overline{\nu}$) in s. For the convenience of this discussion, let us call $\forall^{|\overline{\nu}|+1}\exists$ as the prenex signature of ϕ_s.

Next, in Fig. 2.14, observe that the (ordered) set $\overline{\nu}$ is extended only in the encoding rule for $\gg\!=$. Since an occurrence of $\gg\!=$ adds a quantifier to $|\overline{\nu}|$, if s is a bind expression nested inside a top-level bind expression (like $\mathsf{F}_U(\Delta)$ from the above example), then the prenex signature of ϕ_s is $\forall^2\exists$. Furthermore, if the subexpressions

of s are neither bind nor exists expressions, then none of the ϕ_i's are quantified, and the prenex signature of $\phi = \bigwedge_i \phi_i \wedge \phi_s$ remains $\forall^2\exists$. A similar observation holds when s is an exists expression nested inside a top-level bind expression. Since exists is generated as a result of stabilizing a SQL command transformer, which is always a non-nested bind expression, the subexpression (s') of exists is a non-nested bind expression. s' is however nested inside a top-level bind expression, hence its prenex signature is $\forall^2\exists$. Since exists does not extend $\overline{\nu}$, the prenex signature of s remains $\forall^2\exists$. When s is an expression other than $\gg=$ or exists, then ϕ_s is not a quantified formula, and its prenex signature is trivially subsumed by $\forall^2\exists$. Thus, for the subset of $\mathcal{S}$, where bind expressions are restricted to one level of nesting, the FOL formulas generated by the encoding have the prenex signature as $\forall^2\exists$.

The fragment of FOL that admits formulas with prenex signatures of the form $\forall^2\exists^*$ is called the Gödel-Kálmar-Schütte (GKS) fragment [35], which is known to be decidable. The language of encoding, however, is a combination of GKS with (a). $\mathbb{P}^0$, the theory from which quantifier-free propositions (φ) that encode object language expressions are drawn, and (b). $\mathbb{P}^1$, the theory from which invariants (I) are drawn. Thus, the encoding of the subset of $\mathcal{S}$ described above is decidable if the combination of $\mathsf{GKS} + \mathbb{P}^0 + \mathbb{P}^1$ is decidable. We write $\mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ to highlight the parameterization of $\mathcal{S}$ on $\mathbb{P}^0$ and $\mathbb{P}^1$. The discussion in the previous paragraph points to the existence of non-trivial subsets in $\mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ that are decidable:

Theorem 2.4.2 *There exist* $\mathcal{S}'[\mathbb{P}^0, \mathbb{P}^1] \subset \mathcal{S}[\mathbb{P}^0, \mathbb{P}^1]$ *such that* $\mathcal{S}'$ *is decidable if* $\mathsf{GKS} + \mathbb{P}^0 + \mathbb{P}^1$ *is decidable.*

One interesting example of such an $\mathcal{S}'$ is the subset described above: $\mathcal{S}$ with bind expressions confined to one level of nesting. We denote this subset as $\mathcal{S}^1[\mathbb{P}^0, \mathbb{P}^1]$, for which we assert decidability:

Corollary 2.4.3 $\mathcal{S}^1[\mathbb{P}^0, \mathbb{P}^1]$ *is decidable if* $\mathsf{GKS} + \mathbb{P}^0 + \mathbb{P}^1$ *is decidable.*

$\mathcal{S}^1$ is a useful subset of $\mathcal{S}$, for it corresponds to $\mathcal{T}$ programs without nested `foreach` loops. Observe that the `new_order` transaction (Fig. 2.1) belongs to this subset.

Indeed, $\mathcal{S}^1$, while being a restricted version of $\mathcal{S}$, is nonetheless expressive enough to cover all the benchmarks we considered in Sec. 2.6.

A useful instantiation of $\mathcal{S}^1$ is $\mathcal{S}[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$, where BV is the theory of bit-vector arithmetic, which is often used to encode the finite-bit integer arithmetic of real programs. Finite-bit integer arithmetic has a finite axiomatization in GKS. For instance, 32-bit integers can be encoded as 2^{32} distinct constants of an uninterpreted sort T, while integer operations like addition and multiplication can be encoded as uninterpreted functions whose properties are enumerated for the entire domain of T. Thus BV is subsumed by GKS. Since the latter is decidable, the combination is decidable:

Theorem 2.4.4 $\mathcal{S}^1[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$ *is decidable.*

This instantiation requires I to be drawn from GKS+BV, which is expressive enough to describe common database integrity constraints, such as referential integrity, non-negativeness of all integer values in a column etc. The isolation specifications presented in §2.2.1 are already simple first-order formulas that can be encoded in GKS. Furthermore, it is also reasonable to expect the guarantee (G) of a transaction to be expressible in the same logic as its inferred F, since F (without the stability check) is essentially a complete characterization of the transaction, while G is only an abstraction. Thus, with $\mathcal{S}^1[\mathsf{BV}, \mathsf{GKS} + \mathsf{BV}]$ as the language of inference, the verification problem for weakly isolated transactions is decidable.

2.5 ACIDifier Implementation

We have implemented our DSL to define transactions as monadic computations in OCaml (modulo some syntactic sugar), and our automatic reasoning framework as an extra frontend pass (called ACIDifier) in the ocamlc 4.03 compiler. The input to ACIDifier is a program in our DSL that describes the schema of the database as a collection of OCaml type definitions, and transactions as OCaml functions, whose top-level expression is an application of the `atomically_do` combinator. For in-

```
type table_name =  District | Order | Order_line | Stock

type district = {d_id: int; d_next_o_id: int}
type order = {o_id: int; o_d_id: int; o_c_id: int; o_ol_cnt: int}
type order_line = {ol_o_id: int; ol_d_id: int; ol_i_id: int; ol_qty:
    int}
type stock = {s_i_id: int; s_d_id:int; s_qty: int}
```

Figure 2.15.: OCaml type definitions corresponding to the TPC-C schema from Fig. 2.2

stance, TPC-C's schema from Fig. 2.2 can be described via the OCaml type definitions shown in Fig. 2.15. ACIDIFIER also requires a specification of the program in the form of a collection of guarantees (G), one per transaction, and an invariant I that is a conjunction of the integrity constraints on the database. An auxiliary DSL that includes the first-order logic (FOL) combinators has been implemented for this purpose. ACIDIFIER's verification pass follows OCaml's type checking pass, hence the concrete artifact of verification is OCaml's typed AST. The tool is already equipped with an axiomatization of PostgreSQL and MySQL's isolation levels expressed in our FOL DSL. Other data stores can be similarly axiomatized. The concrete result of verification is an assignment of an isolation level of the selected data store to each transaction in the program.

At the top-level, ACIDIFIER runs a loop that picks an unverified transaction and progressively strengthens its isolation level until it passes verification. If the selected data store provides a serializable isolation level, and if the program is sequentially correct, then the verification is guaranteed to succeed. Within the loop, ACIDIFIER first computes the various rely relations needed for verification (R, $\mathbb{R}_l$, and $\mathbb{R}_c$). It then traverses the AST of a transaction, applying the inference rules to construct a state transformer, checks its stability, and weakens it ($[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$) if it is not stable. The result of traversing the transaction's AST is therefore a state transformer (F)

that is stable w.r.t $\mathbb{R}_l$, which is also stabilized against $\mathbb{R}_c$ (using $[\![\cdot]\!]_{\langle \mathbb{R}, I \rangle}$), and then checked against the transaction's stated guarantee (G). If the check passes, then the guarantee is verified to check if it preserves the invariant I. The successful result from both checks results in the transaction being certified correct under the current choice of its isolation level. Successful verification of all transactions concludes the top-level execution, returning the inferred isolation levels as its output. ACIDifier uses the Z3 SMT solver as its underlying reasoning engine. Each implication check described above is first encoded in FOL, applying the translation described in §2.4 wherever necessary.

2.5.1 Pragmatics

Real-World Isolation Levels The axiomatization of the isolation levels presented in §2.2.1 leaves out certain nuances of their implementations on real data stores, which need to be taken into account for verification to be effective in practice. We take these into account while linking ACIDifier with store-specific semantics (isolation specifications, etc.). As an example, consider how PostgreSQL implements an `UPDATE` operation. `UPDATE` first selects the records that meet the search criteria from the snapshot against which it is executing (the snapshot is established at the beginning of the transaction if the isolation level is SI, or at the beginning of the `UPDATE` statement if the isolation level is RC). The selected records are then visited in the actual database (if they still exist), write locks are obtained, and the update is performed, provided that each matched record still meets `UPDATE`'s search criteria. If a record no longer meets the search criteria (due to a concurrent update), it is excluded from the update, and the write lock is immediately released. Otherwise, the record remains locked until the transaction commits.

Clearly, this sequence of events is not atomic, unlike the assumption made by our formal model because the implementation admits interference between the updates of individual records that meet the search criteria. Nonetheless, through a series of relatively straightforward deductions, we can show that PostgreSQL's `UPDATE` is in

fact equivalent (in behavior) to a sequential composition of two atomic operations $c_1; c_2$, where c_1 is effectively a `SELECT` operation with the same search criteria as `UPDATE`, and c_2 is a slight variation of the original `UPDATE` that updates a record only if a record with the same id is present in the set of records returned by `SELECT`:

$$\texttt{UPDATE}\ (\lambda x.\ e_1)\ (\lambda x.\ e_2) \quad \longrightarrow \quad \texttt{LET}\ y = \texttt{SELECT}\ (\lambda x.\ e_1)\ \texttt{IN}\ \texttt{UPDATE}\ (\lambda x.\ e_1 \wedge x.\texttt{id} \in \mathsf{dom}(y))\ (\lambda x.\ e_2)$$

The intuition behind this translation is the observation that all interferences possible during the execution of the `UPDATE` can be accommodated between the time the records are selected from the snapshot, and the time they are actually updated. ACIDifier performs this translation if the selected store is PostgreSQL, allowing it to reason about `UPDATE` operations in a way that is faithful to its semantics on PostgreSQL. ACIDifier also admits similar compensatory logic for certain combinations of isolation levels and operations on MySQL.

Set functions SQL's `SELECT` query admits projections of record fields, and also application of auxiliary functions such as `MAX` and `MIN`, e.g., `SELECT MAX(ol_o_id) FROM Order_line WHERE ...`, etc. We admit such extensions as set functions in our DSL (e.g., `project`, `max`, `min`), and axiomatize their behavior. For instance:

$$s_2 = \texttt{project}\ s_1\ (\lambda z.\ e) \quad \Leftrightarrow \quad \forall y.\ y \in s_2 \Leftrightarrow \exists (x \in s_1).\ y = [x/z]e$$

$$x = \texttt{max}\ s \quad \Leftrightarrow \quad x \in s \wedge \forall (y \in s).\ y \leq x$$

There are however certain set functions whose behavior cannot be completely axiomatized in FOL. These include `sum`, `count` etc. For these, we admit imprecise axiomatizations.

Annotation Burden ACIDifier significantly reduces the annotation burden in verifying a weakly isolated transactions by eliminating the need to annotate intermediate assertions and loop invariants. Guarantees (G) and global invariants (I), however, still need to be provided. Alternatively, a weakly isolated transaction T can be verified against a generic serializability condition, eliminating the need for guarantee annotations. In this mode, ACIDifier first infers the transformer F_{SER}

of T without considering any interference, which then becomes its guarantee (G). Doing likewise for every transaction results in a rely relation (R) that includes F_{SER} of every transaction. Verification now proceeds by taking interference into account, and verifying that each transaction still yields the same F as its F_{SER}. The result of this verification is an assignment of (possibly weak) isolation levels to transactions which nonetheless guarantees behavior equivalent to a serializable execution.

2.6 Evaluation

In this section, we present our experience in running ACIDifier on two different applications: *Courseware*: a course registration system described by [36], and TPC-C.

Courseware The Courseware application allows new courses to be added (via an `add_course` transaction), and new students to be registered (via a `register` transaction) into a database. A registered student can enroll (`enroll`) in an existing course, provided that enrollment has not already exceeded the course capacity (`c_capacity`). A course with no enrollments can be canceled (`cancel_course`). Likewise, a student who is not enrolled in any course can be deregistered (`deregister`). Besides `Student` and `Course` tables, there is also an `Enrollment` table to track the many-to-many enrollment relationship between courses and students. The simplified code for the Courseware application with only `enroll` and `deregister` transactions is shown in Fig. 2.16. The application is required to preserve the following invariants on the database:

1. I_1: An enrollment record should always refer to an existing student and an existing course.

2. I_2: The capacity (`c_capacity`) of a course should always be a non-negative quantity.

Both I_1 and I_2 can be violated under weak isolation. I_1 can be violated, for example, when `deregister` runs concurrently with `enroll` , both at RC isolation. While the

```
type table_name = Student | Course | Enrollment
type student = {s_id: id; s_name: string}
type course = {c_id: id; c_name: string; c_capacity: int}
type enrollment = {e_id: id; e_s_id: id; e_c_id: id}

let enroll_txn sid cid =
  let crse = SQL.select1 [Course] (fun c -> c.c_id = cid) in
  let s_c_enrs = SQL.select [Enrollment]
                   (fun e -> e.e_s_id = sid &&
                             e.e_c_id = cid) in
  if crse.c_capacity > 0 && Set.is_empty s_c_enrs then
    (SQL.insert Enrollment {e_id=new_id (); e_s_id=sid;
                            e_c_id=cid};
     SQL.update Course
        (fun c -> {c with c_capacity = c.c_capacity - 1})
        (fun c -> c.c_id = cid)) else ()

let deregister_txn sid =
  let s_enrs = SQL.select [Enrollment]
                 (fun e -> e.e_s_id = sid) in
  if Set.is_empty s_enrs
  then SQL.delete Student (fun s -> s.s_id = sid)
  else ()
```

Figure 2.16.: Courseware Application

former transaction removes the student record after checking that no enrollments for that student exists, the latter transaction concurrently adds an enrollment record after checking the student exists. Both can succeed concurrently, resulting in an invalid state. Invariant I_2 can be violated by two `enroll`s, both reading `c_capacity`=1, and both (atomically) decrementing it, resulting in `c_capacity`=-1. We ran ACIDIFIER on the Courseware application (Fig. 2.16) after annotating transactions with their respective guarantees, and asserting $I = I_1 \wedge I_2$ as the correctness condition. The

guarantees G_e and G_d for `enroll` and `deregister` transactions, respectively, are shown below:

$$
\begin{aligned}
G_e(\Delta, \Delta') \;\Leftrightarrow\; & \Delta'_s = \Delta_s \wedge \exists\, \texttt{cid}. \exists\, \texttt{sid}\,. \\
& \Delta'_c = \Delta_c \gg\!= \lambda c.\ \mathsf{if}\ c.\,\texttt{c\_id} = \texttt{cid} \\
& \qquad \mathsf{then}\ \mathsf{exists}(c',\ c'.\,\texttt{id} = c.\,\texttt{id} \wedge c'.\,\texttt{c\_name} = c.\,\texttt{c\_name} \\
& \qquad\qquad \wedge\ c'.\,\texttt{c\_capacity} \geq 0,\ \{c'\}) \\
& \qquad \mathsf{else}\ \{c\} \\
& \wedge \Delta_e = \Delta'_e \gg\!= \lambda e.\ \mathsf{if}\ e.\,\texttt{e\_c\_id} = \texttt{cid} \wedge e.\,\texttt{e\_s\_id} = \texttt{sid}\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \{e\} \\
G_d(\Delta, \Delta') \;\Leftrightarrow\; & \Delta'_c = \Delta_c \wedge \Delta'_e = \Delta_e \wedge \exists\, \texttt{sid}\,.\ \mathsf{if}\ \forall (e \in \Delta_e).\ e.\,\texttt{e\_s\_id} \neq \texttt{sid} \\
& \qquad \mathsf{then}\ \Delta'_s = \Delta_s \gg\!= \lambda s.\ \mathsf{if}\ s.\,\texttt{id} = \texttt{sid}\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \{s\} \\
& \qquad \mathsf{else}\ \Delta'_s = \Delta_s
\end{aligned}
$$

For the sake of this presentation we split Δ into three disjoint sets of records, Δ_s, Δ_c, and Δ_e, standing for `Student`, `Course`, and `Enrollment` tables, respectively. Observing that the set language $\mathcal{S}$ (Sec. 2.4), besides being useful for automatic verification, also facilitates succinct expression of transaction semantics, we define G_e and G_d in a combination of FOL and $\mathcal{S}$. G_e essentially says that the `enroll` transaction leaves the `Student` table unchanged, while it may update the `c_capacity` field of a `Course` record to a non-negative value (even when it doesn't update, it is the case that $c'.\,\texttt{c\_capacity} \geq 0$, because $c' = c$, and $c \in \Delta_c$, and we know that $I_2(\Delta_c)$). G_e also conveys that `enroll` might insert a new `Enrollment` record by stating that Δ_e, the `Enrollment` table in the pre-state, contains all records e from Δ'_e, the table in the post-state, except when $e.\,\texttt{e\_c\_id}$ and $e.\,\texttt{e\_s\_id}$ match `cid` and `sid`, respectively. The guarantee G_d of `deregister` asserts that the transaction doesn't write to `Course` and `Enrollment` tables. The transaction might however delete a `Student` record bearing an `id`$=$`sid` (formally, $\Delta'_s = \Delta_s \gg\!= \lambda s.\ \mathsf{if}\ s.\,\texttt{id} = \texttt{sid}\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \{s\}$), for some `sid` for which no corresponding `Enrollment` records are present in the pre-state (in other words, $\forall (e \in \Delta_e).\ e.\,\texttt{e\_s\_id} \neq \texttt{sid}$).

Table 2.1.: The discovered isolation levels for TPC-C transactions

	`new_order`	`delivery`	`payment`	`order_status`	`stock_level`
MySQL	SER	SER	RC	RC	RC
PostgreSQL	SI	SI	RC	RC	RC

With help of the guarantees, such as those described above, ACIDIFIER was able to automatically discover the aforementioned anomalous executions, and was subsequently able to infer that the anomalies can be preempted by promoting the isolation level of `enroll` and `deregister` to SER (on both MySQL and PostgreSQL), leaving the isolation levels of remaining transactions at RC. The total time for inference and verification took less than a minute running on a conventional laptop.

TPC-C The simplified schema of the TPC-C benchmark has been described in Sec. 4.1. In addition to the tables shown in Fig. 2.2, the TPC-C schema also has `Warehouse` and `New_order` tables that are relevant for verification. To verify TPC-C, we examined four of the twelve consistency conditions specified by the standard, which we name I_1 to I_4:

1. Consistency condition I_1 requires that the sales bottom line of each warehouse equals the sum of the sales bottom lines of all districts served by the warehouse.

2. Conditions I_2 and I_3 effectively enforce uniqueness of ids assigned to `Order` and `New_order` records, respectively, under a district.

3. Condition I_4 requires that the number of order lines under a district must match the sum of order line counts of all orders under the district.

Similar to the example discussed in Sec. 4.1, there are a number of ways TPC-C's transactions violate the aforementioned invariants under weak isolation. ACIDIFIER was able to discover all such violations when verifying the benchmark against $I = \bigwedge_i I_i$, with guarantees of all three transactions provided. The isolation levels

were subsequently strengthened as shown in Table. 2.1. As before, inference and verification took less than a minute.

To sanity-check the results of ACIDIFIER, we conducted experiments with a high-contention OLTP workload on TPC-C aiming to explore the space of correct isolation levels for different transactions. The workload involves a mix of all five TPC-C transactions executing against a TPC-C database with 10 warehouses. Each warehouse has 10 districts, and each district serves 3000 customers. There are a total of 5 transactions in TPC-C, and given that MySQL and PostgreSQL support 3 isolation levels each, there are a total of $3^5 = 243$ different configurations of isolation levels for TPC-C transactions on MySQL and PostgreSQL. We executed the benchmark with all 243 configurations, and found 171 of them violated at least one of the four invariants we considered. As expected, the isolation levels that ACIDIFIER infers for the TPC-C transactions do not result in invariant violations, either on MySQL or on PostgreSQL, and were determined to be the weakest safe assignments possible.

2.7 Related Work

This section discusses the work that is closely related to the work presented in this chapter.

Specifying weak isolation. Adya [29] specifies several weak isolation levels in terms of *dependency graphs* between transactions, and the kinds of dependencies that are forbidden in each case. The operational nature of Adya's specifications make them suitable for runtime monitoring and anomaly detection [37–39], whereas the declarative nature of the isolation specifications presented in this chapter make them suitable for formal reasoning about program behavior. Cerone *et al.* [30] specify isolation levels with atomic visibility using the vocabulary introduced in [40], but such trace-level specifications are hard to relate to high-level semantics of database programs.

Crooks *et al.* [41] also explore the use of a state-based interpretation of isolation, and present specifications of weak isolation that are not tied to implementation-specific artifacts. However, they do not consider verification (manual or automated) of client programs, and it is not immediately apparent if their specification formalism is amenable for use within a verification toolchain. Warszawski *et al.* [42] present a dynamic analysis for weak isolation that attempts to discover weak isolation anomalies from SQL log files. Their solution, while capable of identifying database attacks due to the use of incorrect isolation levels, does not consider how to verify application correctness, infer proper isolation levels, or formally reason about the relationship between weak-isolation levels and application invariants.

Reasoning under weak isolation. Fekete *et al.* [43] propose a theory to characterize non-serializable executions that arise under SI. They also propose an algorithm that allocates either SI or SER isolation levels to transactions while guaranteeing serializability. Cerone *et al.* [44] improve on Adya's SI specification and use it to derive a static analysis that determines the safety of substituting SI with a weaker variant called *Parallel Snapshot Isolation* [45]. These efforts focus on establishing the equivalence of executions between a pair of isolation levels, without taking application invariants into account. Bernstein *et al.* [46] propose informal semantic conditions to ensure the satisfaction of application invariants under weaker isolation levels. All these techniques are tailor-made for a finite set of well-understood isolation levels (rooted in [28]).

Reasoning under weak consistency. There have been several recent proposals to reason about programs executing under weak consistency [8, 36, 47–50]. All of them assume a system model that offers a choice between a *coordination-free* weak consistency level (*e.g.*, eventual consistency [8, 47–50]) or causal consistency [36, 51]). All these efforts involve proving that atomic and fully isolated operations preserve application invariants when executed under these consistency levels. In contrast, the work presented in this chapter focuses on reasoning in the presence of weakly-

isolated transactions under a strongly consistent data store. Gotsman *et al.* [36] adapt *Parallel Snapshot Isolation* to a transaction-less setting by interpreting it as a consistency level that serializes writes to objects; a dedicated proof rule is developed to help prove prove program invariants hold under this model. By parameterizing the proof system over a gamut of weak isolation specifications, ACIDifier avoids the need to define a separate proof rule for each new isolation level we may encounter.

Inference. Vafeiadis *et al.* [52,53] describe *action inference*, an inference procedure for computing rely and guarantee relations in the context of RGSep [54], an integration of rely-guarantee and separation logic [55] that allows one to precisely reason about local and shared state of a concurrent program. The ideas underlying action inference have been used to prove memory safety, linearizability, shape invariant inference, etc. of fine-grained concurrent data structures. While the motivation of this work is similar (automated inference of intermediate assertions and local invariants), the context of study (transactions vs. shared-memory concurrency), the objects being analyzed (relational database tables vs. concurrent data structures), the properties being verified (integrity constraints over relational tables vs. memory safety, or linearizability of concurrent data structure operations) and the analysis technique used to drive inference (state transformers vs. abstract interpretation) are quite different.

3 BOUNDED VERIFICATION UNDER WEAK CONSISTENCY

This chapter shifts the focus from weak isolation to weak consistency. We[1] present a reasoning technique based on bounded symbolic execution to find weak consistency anomalies violating the integrity of the replicated state in distributed applications. The symbolic execution is complemented with an inference procedure that traverses a finite lattice of consistency levels, and finds the weakest consistency level sufficient preempt an anomaly. Repeating the anomaly detection-repair loop sufficient number of times yeilds bounded guarantees on program correctness.

Our symbolic execution engine (called[2] Q9) operates in the context of a programming framework embedded in OCaml that allows the expression of replicated data types (RDTs) composed of a given library of conflict-free/convergent replicated data types (CRDTs). The framework lets effectful computations be defined over instances of these types. The engine abstracts executions in terms of path conditions and RDT operations, under an axiomatization of a data storage model that only provides weak eventual consistency guarantees on object updates. The engine checks application-specific safety properties on different state configurations induced by considering executions in which the visibility and ordering of RDT operations on different replicas may vary. Q9 tracks operations precisely (up to a bound), thus ensuring that every violation of a safety property is a true violation. Over a collection of benchmark results, including well-studied database applications [56], Q9 was able to correctly identify anomalies that arise because satisfiability of the application's safety properties demand greater coordination and synchronization than manifest explicitly in

[1]This work was done in collaboration with Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan

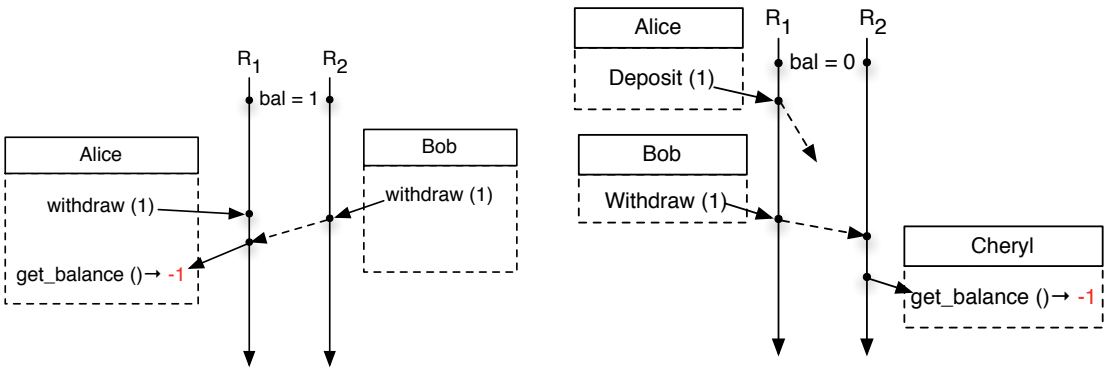
[2]The number 9 in Q9 refers to our initial hypothesis that most replication anomalies manifest under 9 or fewer concurrent operations. The letter 'q' is a symbol resembling 9, hinting at our approach of using symbolic execution to uncover such anomalies.

the application or which is implicitly supported by the storage layer. Counterexamples generated by Q9 are used to automatically strengthen the consistency level (i.e., the degree of global synchronization required) of offending operations. Empirical results support the thesis that anomalies can be detected quickly under relatively small bounds, and repaired easily by selectively strengthening consistency requirements on RDT operations to enforce greater coordination among replicas.

3.1 Replicated State Anomalies: The Motivation for Verification

Consider a simple distributed application that maintains a bank account with replicated state. The representation of a bank account may be in terms of a convergent replicated type (e.g., an integer PN-counter [57] that admits increments and decrements) that guarantees all replicas will *eventually* reflect the same value of the account. However, convergence alone may not be sufficient to preserve application-specific safety properties. For example, suppose we wish to assert that the balance of the account will always be non-negative. Given operations to deposit and withdraw amounts into the account, it is straightforward, in a sequential execution, to ensure this invariant is always preserved, by ensuring that `deposit` only ever adds to the balance, and that `withdraw` always checks if there is a sufficient balance before withdrawing. However, asynchronous replication may lead to anomalous executions that violate the invariant. Two such executions that are illustrative of the anomalies possible under asynchronous replication are shown in Fig. 3.1.

Fig. 3.1a depicts an execution that allows two `withdraw` operations to be applied concurrently at different replicas. Two users, Alice and Bob, assume that there is a sufficient balance in their (joint) account, and issue a withdraw operation for $1 each, to replicas R_1 and R_2, respectively. Each withdraw operation reads the local balance ($1), checks that it is sufficient for the withdraw ($1 $\geq$ $1), and subsequently issues a `Withdraw` effect that will be asynchronously transmitted to the other replica. The effect is essentially a computational message that updates the state of the account on



(a) Anomaly due to concurrency

(b) Anomaly due to out-of-order message delivery

Figure 3.1.: Anomalous executions of a simple bank account application. All operations operate over a single replicated account object.

the replicas which receive it. When both `Withdraw` effects are eventually applied at both the replicas, the balance drops below $0 resulting in an invariant violation, which gets witnessed by Alice when she queries the balance. The anomaly is reminiscent of a classical data race between two writes in a shared memory system, except that writes are not lost or overwritten[3]

However, unintended executions that are unfamiliar to shared memory programmers are also possible in an asynchronous replicated system. Consider the execution shown in Fig. 3.1b involving three users - Alice, Bob, and Cheryl, and two replicas - R_1 and R_2. The initial balance at both the replicas is $0. Alice first submits a `deposit` operation for $1 to R_1. Bob, who subsequently connects to R_1, finds there is sufficient balance to perform a `withdraw` for $1. While effects from both the operations are expected to be delivered to R_2, it is possible that because of transient network conditions, the `Withdraw` effect gets delivered first while the `Deposit` is still in transit. This results in a transient violation of the no-negative-balance invariant, which gets witnessed when Cheryl queries the balance at R_2. The `Deposit` effect will

[3]Note that we cannot rectify this anomaly by simply forcing each replica to check the balance before applying a received `Withdraw` effect as that may cause the account balance on different replicas not to converge.

eventually be delivered to R_2 resulting in the invariant being restored; however since there are no bounds on when this can happen, there are no guarantees on how this violation may affect system behavior. Indeed, it is possible that a temporary violation of safety may lead to cascading errors that compromise application integrity. For instance, a negative (albeit temporary) balance could be witnessed by a minimum balance enforcement module, which may erroneously impose a penalty on the account that remains even after the violation is remedied. As this example demonstrates, asynchronous replication of an application's state can result in anomalous behaviors that could be confounding to understand and repair. It is clearly unreasonable to expect an application programmer to be prescient about the anomalies that might manifest under replication, determine if they indeed lead to invariant violations, and fix the application to avoid them.

Q9's verification engine is based on the observation that concurrency anomalies under replication most often have small representative counterexamples involving few concurrent operations, similar to those shown in Fig. 3.1. Such anomalies can be exposed by exploring the state space of the application with a relatively small bound on the number of concurrent operations. Moreover, by representing the state space using an appropriate formal vocabulary that abstracts away low-level details, such as process crashes and network faults, we can compute an abstract representation of each counterexample that represents not only the counterexample, but an entire *class* of such counterexamples. Systematically eliminating such classes of counterexamples by consistency strengthening leads us to compute the weakest consistency configuration at which an application is free of all discovered anomalies, and hence most likely to be safe.

3.2 The Q9 Programming Framework

The first component of Q9 is a programming framework implemented as a collection of type definitions and libraries in OCaml, intended to operate within a

```
module BankAccount (C:CRInt) : BANKACCT = struct
  type t = C.t
  type m = C.m
  let init = C.init_val
  let get_balance () : int = C.get ()
  let do_deposit (amt:int) : m =  C.add amt
  let do_withdraw (amt:int) : m  =
    if C.get () >= amt then C.add (0-amt)
    else error "Insufficient Balance"
  let inv_non_neg_bal () : bool = CRInt.get () >= 0
end

module CRInt : sig
  type t
  type m = t -> t
  val init_val : t
  val add : int -> m
  val get : unit -> int
end

module Bank (Checking : BANKACCT) (Savings: BANKACCT) = struct
  type t = Checking.t * Savings.t
  type m = t -> t
  let txn_transfer (amt:int) : m =
    let m1 = Checking.do_withdraw amt in
    let m2 = Savings.do_deposit amt in
    fun (x,y) -> (m1 x, m2 y)
end
```

Figure 3.2.: A Bank Account application written in Q9

replicated, eventually consistent, distributed environment; for this purpose, it comes equipped with a library of convergent RDT [57] definitions. The semantics of a CRDT

object guarantee that even when multiple operations are applied to it in different order on different replicas, the object's state at all replicas, after all operations have been delivered and executed (i.e., when the system becomes quiescent), will be the same. Q9 uses CRDT specifications to specify richer RDTs through compositional abstractions, capturing notions of convergent state replication for free without having to define a specialized network/storage layer, or to prove additional semantic properties (e.g., commutativity) for the sake of convergence. In this section, we illustrate the programming model with the help of an example.

Fig. 3.2 shows a simple `BankAccount` application written in Q9 that was informally introduced in the previous section. The application manages a replicated object (a bank account) whose underlying representation is given in terms of a CRDT integer (`C.t`). The signature of a `CRInt` defines two operations: `get` returns the current value of the integer, and `add` adds its argument to the existing value of the integer. Observe that while the signature for `get` is unremarkable, `add`'s type returns a value of type `m`, a function type with signature `C.t -> C.t`. This return type captures the essence of an effectful operation in a replicated setting. The function returned by `add` is intended to be applied to every instance of `C` on every replica in the distributed environment managed by Q9 supplying the value of the integer (`C.t`) at that replica as the argument to this function. On the other hand, `get`'s signature does not appeal to `m` - it is expected to return the value of the integer on the replica to which it is applied, in contrast to `add`. This particular specification of CRDTs allows us to hide the implementation artifacts of replication behind high-level signatures that characterize an operation's local and remote effects.

The `BankAccount` module defines standard banking operations that internally manipulate the `CRInt` CRDT. The operations include `get_balance`, `do_deposit`, and `do_withdraw`. `get_balance` function is standard - like `C.get`, it returns the integer value of the current state on whatever replica it is applied to. Operations `do_deposit` and `do_withdraw`, on the other hand, are effectful. These operations return an *effect*, which is essentially a computation that must eventually be performed

on all replicas. The type of an effect is therefore `t -> t` for which we introduce a type synonym `m`. Thus, the type of an RDT operation[4] over type `t`, that expects an argument of type `a`, and returns a `t` effect (i.e., `m = t -> t`) is as follows:

```
type a t oper = t -> a -> m
```

Note that, by definition, an RDT operation acts on a single instance of an RDT.

As described above, the semantics of an RDT operation follows from the CRDT computation it returns. For instance, a `do_deposit` operation returns a `CRInt.m` computation that when invoked on a replica R will add `amt` to `C`'s integer state (`C.t`) on R. This abstract notion of an *effect* can be concretized as a function that maps a replica state to a new state. As a convention, we use pascal case and uncurried arguments to denote an effect, and snake case and curried arguments for an operation. We also drop the prefix `do_`. Hence, the effect of (`do_deposit amt`) is written `Deposit(amt)`. Its semantics is defined by ascribing it the following denotation:

$$[\![\,\texttt{Deposit(amt)}\,]\!] = \lambda\,\texttt{s}'\,.\,\texttt{s}' + \texttt{amt}$$

In ascribing the above denotation to `Deposit`, we assumed that `CRInt.t` is an `int` and `CRInt.add` is basically integer addition. We can similarly ascribe a denotation to `Withdraw(amt)` effect. Assuming that `withdraw` generates an effect only when the balance check is satisfied on the origin replica, the `Withdraw(amt)` effect can be thought of as a function that simply decrements `amt` from the integer state on all the replicas:

$$[\![\,\texttt{Withdraw(amt)}\,]\!] = \lambda\,\texttt{s}'\,.\,\texttt{s}' - \texttt{amt}$$

The function `get_balance` doesn't generate an effect, and thus has no need for a denotation.

As highlighted by the `oper` type definition above, an RDT operation (`do_...`) is only ever allowed to operate on a single RDT. However, in general, distributed applications will need to compose multiple RDT operations to perform useful computation. To express such compositions, Q9 additionally supports transactions over

[4]By convention, we denote such operations by prefixing their name with `do_`.

replicated objects. In Fig. 3.2, module `Bank` composes two `BankAccount` objects - one (`Checking.t`) denoting a checking account and the other (`Savings.t`) savings. It defines a `txn_transfer` transaction (the prefix `txn_` is a naming convention for transactions) that withdraws from the former and deposits to the latter. Provided that the withdraw on the checking account is successful, it returns a composite effectful computation that when applied on a replica serves to perform the transfer on the instance of `Checking` and `Savings` on that replica. If we think of a transaction as generating an aggregate effect, that effect is a composition of effects of individual RDT-specific operations that constitute the transaction. For instance, if `txn_transfer amt` is thought of as generating a `Transfer(amt)` effect, its denotation is as follows:

$$[\![\,\texttt{Transfer(amt)}\,]\!] = \lambda(\,\texttt{s}_1'\,,\,\texttt{s}_2'\,).\,([\![\,\texttt{Withdraw}\,(\,\texttt{amt}\,)]\!](\,\texttt{s}_1'\,),[\![\,\texttt{Deposit}\,(\,\texttt{amt}\,)]\!](\,\texttt{s}_2'\,))$$

The result of executing `transfer` is therefore the generation of this effect.

Finally, Q9 also allows applications to specify safety properties as boolean functions, via functions whose names are prefixed by `inv_`. These safety properties become the basis for verifying RDT applications.

3.2.1 Explicit Effect Representation

Besides explicating the semantics of RDT operations, the notion of an effect serves a more concrete purpose in Q9. Named effects (e.g., `Withdraw(amt)`) constitute the class of messages that are exchanged between replicas, and act as the pivot for consistency enforcement. More importantly, effects provide a tangible structure to reason about concurrent operations potentially executable on different replicas, an essential requirement for any verification exercise. For these reasons, Q9 translates high-level RDT programs to an intermediate representation (IR) that uses explicit effects. In our running example, the translated version of the `BankAccount` RDT in the IR includes the following definitions

```
type eff = Deposit of int
         | Withdraw of int

let apply_eff (s':int) (e:eff) =
  match e with
    | Deposit(a) -> s' + a
    | Withdraw(a) -> s'-a
```

Note that the `eff` type definition and the `apply_eff` function reify the abstract notions of effects and their semantics in the context of the `BankAccount` application. Operations and transactions can now be defined in terms of these explicit effects:

```
let deposit (amt:int) = Deposit(amt)

let withdraw (amt:int) =
  if s>= amt then Withdraw(amt)
  else error "Insufficient Balance"

let transfer (amt:int) =
  let eff1 = Checking.withdraw amt in
  let eff2 = Savings.deposit amt in
    (eff1,eff2)
```

Observe that, under this formulation, CRDT definitions such as `CRInt` exist only to *transfer* CRDT semantics to their consumers. After compilation to the effect-aware IR, the application's RDTs themselves become CRDTs; e.g., applying (via `apply_eff`) a collection of effects (i.e., values of `eff` type) in any order results in the same `BankAccount` state. Thus, the Q9 programming model serves as a way to engineer arbitrary distributed applications with convergent semantics, while its underlying IR directly manipulates effects in ways consistent with CRDT semantics.

The translation to this IR elaborates each operation (i.e., `do`-prefixed function on an RDT) in to a representation that returns the corresponding effect. The effect takes the place of the RDT computation `m` in the definition of the operation (for e.g., compare `withdraw` given above to its definition in Fig. 3.2). Conversely, the interpretation of the effect of an operation at state s' is obtained by inlining the CRDT computation (`m`), and applying it on s' (for e.g., compare the interpretation of `Withdraw` in `apply_eff` above to the definition of `do_withdraw` in Fig. 3.2). A

```
module Microblog(Tweet: TWEET)(Userline : USERLINE)
                (Timeline : TIMELINE) =
struct
  type t = Tweet.t * Userline.t * Timeline.t

  let txn_new_tweet (uid: user_id) (str: string) =
    let tweet_id = UUID.new () in
    let e1 = Tweet.new tweet_id str in
    let e2 = Userline.add uid tweet_id in
    let fids = User.get_followers uid in
    let e3 = Timeline.add
      (Set.map (fun fid -> (fid,tweet_id)) fids) in
    (e1,e2,e3)
end
```

Figure 3.3.: Microblog application's `txn_new_tweet` transaction

transaction's effect is a composition of effects on multiple RDT objects in the same way as the object it manipulates is a composition of multiple RDT objects. For instance, `txn_transfer` of `Bank` returns a pair of `BankAccount` effects for the type it manipulates (`Bank.t`) is a pair of `BankAccount.t` objects.

Microblog. Fig. 3.3 shows a more complex transaction from a Twitter-like microblogging application in explicit effect representation. The transaction manipulates objects of three different types: `Tweet.t`, `Userline.t`, and `Timeline.t`. Each object can be thought of as a set of records of a similar type, akin to a table in a relational database. For instance, `Tweet.t` represents a set of tweets. The transaction first constructs an effect (`e1`) for adding the new tweet to the collection of tweets, followed by an effect (`e2`) to add the corresponding tweet id to *userline* of the author (identified by `uid: user_id`), and finally an effect (`e3`) to add the tweet id to the *timeline* of every follower of the author. It returns a tuple of these effects to be applied on first, second and third components of `Microblog.t` object, respec-

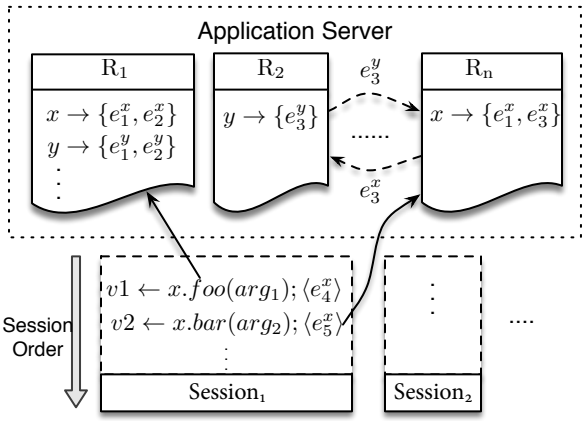


Figure 3.4.: Q9 system model.

tively. We shall revisit this transaction in Sec. 4.5, where we describe the anomalies it exhibits, along with the fixes that Q9 discovers.

3.3 System Model

Figure 3.4 presents a schematic diagram of the system model adopted by Q9. An application's state is composed of multiple objects $(x, y, \ldots)$, each of which is replicated across multiple locations. Each location is called a replica. Each replica maintains an unordered *history* of an object's effects known to that replica. For example, the history of of the object x at replica R_1 (in Fig. 3.4) is the set $\{e_1^x, e_2^x\}$, whereas its history at replica n is the set $\{e_1^x, e_3^x\}$. The difference in histories is due to *asynchronous replication* which allows effects to be propagated lazily (through the network) and delivered asynchronously. Under a reasonable assumption that the network offers eventual delivery guarantees, all generated effects will be present on all replicas eventually. The state of an object at a replica is a function of the object's history; it is computed by applying the effects, in no particular order, to the initial object of the RDT. For instance, if x is a `BankAccount` object, and e_1^x is `Deposit(10)`, and e_2^x is `Withdraw(5)`, then a possible state of x at R_1 is:

$$
\begin{aligned}
x &= [\![\texttt{Deposit(10)}]\!]\,([\![\texttt{Withdraw(5)}]\!]\,(\texttt{BankAccount.init})) \\
&= (\lambda s.\, s + 10)\,((\lambda s.\, s - 5)\,0) \\
&= 5
\end{aligned}
$$

The clients of the application interact with the system by invoking operations (e.g., `deposit`) on objects. The sequence of operations invoked by a particular client on an object is called a *session*, and the operations found in this sequence are said to be in a *session order* with one another. Session order also relates the operations within a transaction. For e.g., in the `txn_transfer` transaction of Fig. 3.2, the `withdraw` operation precedes the `deposit` operation in session order. At any given instant, an application could be serving multiple concurrent clients/sessions. An operation or a transaction invoked by a client is executed at one of the replicas (e.g., operation *foo* executes at R_1 in Fig. 3.4). Due to transient system conditions (e.g., network partitions, load balancing etc), it is possible that the operations of the same session get executed at different replicas. For example, operation *bar* from the same session as *foo* executes at a different replica. When an operation (e.g., *foo*) is executed on an object (x) at a replica (R_1), it is supplied the state of the object (x) computed from its history at the replica (R_1). In this case, we say that the effects in the history (e_1^x and e_2^x) are *visible* to the operation (*foo*).

The system described above is quite general insofar as it makes no assumptions on either the timing or order in which effects are generated and propagated. Indeed, the model abstracts many realworld distributed data stores [15,16,58,59], and is consistent with the models used in a number of research prototypes such as Walter [45], Chapar [51], Antidote [60], and Quelea [61].

$$
\begin{array}{llll}
x, y, f \in \texttt{Variables} & c \in \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} & W_{i\in[1,N]} \in \texttt{Eff Constructors} & \\
B \in \texttt{Base Types} & \coloneqq & \texttt{int} \mid \texttt{bool} \mid B\ \texttt{set} \mid B \to B & \\
T \in \texttt{Types} & \coloneqq & B \mid \texttt{eff} \mid T\ \texttt{set} \mid T \to T & \\
p \in \texttt{Patterns} & \coloneqq & \texttt{true} \mid \texttt{false} \mid W_{i\in[1,N]}(x) \mid \emptyset \mid \{x\} \mid x \cup y & \\
e \in \texttt{Expressions} & \coloneqq & c \mid x \mid \lambda x.e \mid \texttt{fix}\, e \mid e\, e \mid \emptyset \mid \{e\} \mid e \cup e & \\
 & & \mid W_{i\in[1,N]}(e) \mid \texttt{match}\ e\ \texttt{with}\ p \Rightarrow e\ \texttt{else}\ e \mid [e]_T & \\
\pi \in \texttt{Programs} & \coloneqq & [\lambda x.e]_{B\to\texttt{eff}} \mid \pi \,||\, \pi &
\end{array}
$$

Figure 3.5.: λ_R: The core calculus of Q9

3.4 The Q9 Verification Engine

3.4.1 Core Calculus

Fig. 3.5 shows the syntax of the core calculus (λ_R) of Q9 that lets us capture the essence of Q9 programs abstractly. The calculus operates on integer and boolean values, sets, and functions. A special type `eff` for effects is also present, and it is assumed to be a (non-recursive) sum type of N effect constructors - $W_{i\in[1,N]}$, where each constructor has exactly one argument of a base type (B). The syntactic class of expressions includes the usual suspects - lambda abstractions, applications etc., along with set expressions which include an empty set constructor ($\emptyset$), a singleton constructor ($\{e\}$), and a set union constructor ($e_1 \cup e_2$). Applications of effect constructors ($W_{i\in[1,N]}$) are expressions of type `eff`. A `match-with-else` expression lets a value be matched against a pattern (p), and if the match is successful, evaluates the corresponding expression. Any λ_R expression can be annotated with its type ($[e]_T$).

At the top-level, a λ_R program (π) is a parallel composition of functions of type $B \to$ `eff`, where B is a non-effect (base) type. Intuitively, a λ_R program models a $Q9$ application operating over a single RDT maintaining a state of type B (e.g., B

could be `int` in our running `BankAccount` example). Each function represents an invocation of an operation on the RDT; this construction models the generation of operations by a client in a given session; invocations can proceed in parallel ($\pi \,||\, \pi$). When an operation is invoked, it reads the current state at *some* replica, which is a value of type B, and generates a new effect (an `eff`). Since our system model does not mandate that all replicas witness the effect generated by an operation instantaneously, any operation may witness a *subset* (say, S) of effects generated so far by operations that executed previously. The state an operation witnesses is the result of applying the effects in S to an initial state. We write $\iota : B$ to denote the initial state of the RDT being modeled.

Recall that the denotation of an effect is a function that defines what it means to apply that effect (e.g., from Sec. 3.2: $[\![\,\texttt{Deposit(amt)}\,]\!] = \lambda\,\texttt{s}\,.\,\texttt{s} + \texttt{amt}$). The denotation of a set of effects is simply a functional composition of the denotation of its constituents (in the following, $\uplus$ denotes a disjoint union, ϵ stands for an effect, and $\circ$ is the function composition operator as in $f \circ g$):

$$\begin{aligned} [\![\emptyset]\!] &= \lambda x.\, x \\ [\![S \uplus \{\epsilon\}]\!] &= [\![S]\!] \circ [\![\epsilon]\!] \end{aligned}$$

Thus, the result of applying a set S of effects to ι is defined by $([\![S]\!]\,\iota)$.

Our calculus does not support transactions - each invocation ($[\lambda x.e]_{B \to \texttt{eff}}$) operates over a single RDT, and produces a single effect. Supporting transactions is however straightforward - invocations would need to be supplied multiple RDTs to operate over, may produce multiple effects, and each of these effects may have internal (session) ordering guarantees that would need to be preserved. We revisit these issues in Sec. 3.5.

Having defined what it means to apply a set of effects, we can now capture the essence of the operational semantics of a λ_R program in a single rule defined over sets of effects, rather than replicas, or other system-level artifacts:

$$\frac{S \subseteq \mathsf{A} \quad (\lambda x.e)\ ([\![S]\!]\ \iota) \Downarrow\ W_j(v)}{(\mathsf{A},\ [\lambda x.e]_{B \to \texttt{eff}}\ ||\ \pi) \longrightarrow (\mathsf{A} \cup \{W_j(v)\},\ \pi)}\ \text{[E-Oper]}$$

The rule uses the set of effects generated thus far (A) as a proxy for the overall system state, and $[\![S]\!]$, where $S \subseteq \mathsf{A}$ as a proxy for a replica state. Since this covers all possible system configurations and replica states, the semantics is general enough to admit all possible behaviors of a distributed program interacting within asynchronously replicated state.

The E-Oper rule describes a small-step evaluation relation of λ_R programs (π) with the following signature:

$$(\mathsf{A}, \pi) \longrightarrow (\mathsf{A}', \pi')$$

As described above, A is the set of effects generated in the system, and π denotes the program being reduced. E-Oper is the only computation rule of the small-step relation; the rest are congruence rules that let s morph into a form suitable for E-Oper. The antecedent uses a big-step evaluation relation ($\Downarrow$) to interpret expressions. The definition of this relation is standard and elided here.

Safety Recall that Q9 applications define their safety properties as boolean functions. Let $I = [\lambda x.\, e]_{B \to \texttt{bool}}$ denote such an invariant. We say that a certain RDT state $s : B$ satisfies the invariant iff $I(s)$ evaluates to `true` as per the big-step semantics of λ_R. Using I, we can informally capture the safety of the application as follows:

- The initial state ι at every replica satisfies the invariant I, and
- At any given instance, if the state at every replica satisfies the invariant I, and if we execute *any* operation f at some replica R generating an effect ϵ, then applying ϵ at any replica R' results in a state that still satisfies I.

We can transplant this informal characterization into the framework of our calculus thus:

Definition 3.4.1 *An execution of a program π is* safe *with respect to invariant I if and only if $\forall A, A', \iota$, if:*

- $I\,\iota \Downarrow \texttt{true}$
- $\forall S \subseteq A, I([\![S]\!]\,\iota) \Downarrow \texttt{true}$
- $(\mathsf{A}, \pi) \longrightarrow (\mathsf{A}', \pi')$

then $\forall S' \subseteq A', I([\![S']\!]\,\iota) \Downarrow \texttt{true}$.

Note that the definition folds the generation of effect ϵ into the small-step transition $(\mathsf{A}, \pi) \longrightarrow (\mathsf{A}', \pi')$; it is expected that $\mathsf{A}' = \mathsf{A} \cup \{\epsilon\}$. The definition also takes into account the effects that are generated concurrently with ϵ: these are the effects that are present in A, but not in the set $S \subseteq \mathsf{A}$ witnessed by ϵ. The effects in $\mathsf{A} - S$ may have been generated before or been concurrent with ϵ in realtime, but our asynchronous model doesn't make such distinctions: if an effect is not visible to ϵ (i.e., not present on the replica where ϵ was generated), it is a concurrent effect insofar as the operational semantics is concerned.

A subtle yet consequential aspect of the above formal development is that it only loosely constrains the notion of an initial state ι. Def. 3.4.1 defines the safety of a concrete execution starting from *any* state ι that satisfies the invariant I. In particular, ι is not obligated to denote either an empty state (i.e., a state with no effects), or a state reachable from an empty state; instead, it need only be a state that satisfies the invariant I. This rather broad definition of ι nonetheless captures the reality of a database application, which is allowed to start its execution from an arbitrary database state created independently of the application's interface, as long as the state satisfies the application's integrity constraints I. For instance, TPC-C's reference implementation [56] ships with a sample workload generator that executes the application against a database state not reachable from an empty state by a TPC-C execution, but one that satisfies all of the TPC-C's stated integrity constraints. A downside to this relaxed specification of ι however is that it may require I to be strengthened to include *all* valid database states, failing which the application could be judged unsafe by Def. 3.4.1. This problem is revisited in Sec. 3.4.3, and again in Sec. 4.5 in the context of real applications.

The calculus and operational semantics described here succinctly capture the semantics of concurrent programs under asynchronous state replication without having to concretize low-level aspects of the system model such as message communication, process creation, replica organization, etc. Notwithstanding its succinctness, checking safety in the sense described above by naively exploring a concrete state space of executions is clearly infeasible given the large set of behaviors that are possible. We therefore refine our semantics to leverage symbolic reasoning to enable us to characterize and represent many concrete states at once.

3.4.2 Abstract Relations

In this section, we introduce the formal vocabulary that lets Q9 represent anomalies and consistency specifications abstractly. We say effect e_1 is *visible* to another effect e_2 ($\mathsf{vis}(e_1, e_2)$) if the operation that generated e_2 was executed against a state to which e_1 has already been applied. For instance, in Fig. 3.1b, the effect (call it e_1) of Alice's `Deposit` is visible to the effect (call it e_2) of Bob's `Withdraw`. The visibility relation is irreflexive - effects cannot see themselves; asymmetric - if e_1 is visible to e_2, then e_1 necessarily happened before e_2, therefore cannot see e_2; and, non-transitive - if e_1 is visible to e_2, and e_2 is visible to e_3, then e_1 need not necessarily be visible to e_3; Fig. 3.1b captures such a scenario. Another important aspect of visibility is that it only ever relates effects on the same object. This follows from the fact that an operation is only allowed to access the state of a single object, hence can only witness the effects of previous operations on *that* object. Given a relation $\mathsf{sameobj}$ that relates effects on the same object, we have:

$$\forall e_1, e_2.\ \mathsf{vis}(e_1, e_2) \Rightarrow \mathsf{sameobj}(e_1, e_2)$$

The *session-order* relation relates effects of the same session in the order they are generated. For instance, in Fig. 3.4, effects e_4^x and e_5^x are in a session order (written $\mathsf{so}(e_4^x, e_5^x)$). Session order is irreflexive and asymmetric for the same reason as visibility, but it is transitive because the order of operations in a session is a total order.

Having formalized visibility (vis) and session order (so), we can define what it means for an effect e_1 to *happen before* an effect e_2. Clearly, e_1 has happened before e_2 if $\mathsf{vis}(e_1, e_2)$ or $\mathsf{so}(e_1, e_2)$. Moreover, if we already know that an effect e_0 has happened before e_1, and if $\mathsf{vis}(e_1, e_2)$ or $\mathsf{so}(e_1, e_2)$, then it follows that e_0 has happened before e_2. Observe that these are the only two ways that the asynchronous system model we defined in Sec. 3.3 lets us define a *happens-before* ordering of effects. Thus, the happens-before relation (denoted hb) is simply a transitive closure of $\mathsf{vis} \cup \mathsf{so}$:

$$\mathsf{hb} = (\mathsf{vis} \cup \mathsf{so})^+$$

Visibility, session order, and happens-before are the major (binary) relations that let us capture dependencies among effects generated by an application. To aid our exposition, we also define various helper (unary) functions - oper and arg, that let us project various attributes of an effect. Recall that the type of effects in Q9's IR (Sec. 3.2) is a tagged union of effect arguments. Functions oper and arg project the tag and the argument, respectively, of an effect. For instance, $\mathsf{oper}(\texttt{Withdraw}\,(5)) = \texttt{Withdraw}$ and $\mathsf{arg}(\texttt{Withdraw}\,(5)) = 5$.

3.4.3 Symbolic Execution

In this section, we consider a replacement of the concrete evaluation relations ($\rightarrow$, $\Downarrow$) with symbolic counterparts ($\hookrightarrow$, $\downarrow$) to facilitate symbolic reasoning over states and effects. In the process, we also take into account the specific characteristics of asynchronous state replication so as to bound the state space and expedite the process of anomaly detection.

Intuition

Recall that the semantics of λ_R tracks the state of a program's execution as A, the set of effects generated thus far, and the state of a replica as a subset S of A. This construction accounts for any number of replicas, and a liberal network semantics with arbitrary latency and message reordering. (In reality though, there are only a

finite number of replicas), and inter-replica latency is usually comparable to (i.e., a small multiple of) replica-local execution time. As a result, in practice, a non-trivial subset S_b of effects in A can be expected to be already present on all replicas [62].

The corresponding state $b = [\![S_b]\!]$ is therefore a "common prefix" of all the replica states, which is extended at each replica by applying a subset of effects from $S_c = \mathsf{A} - S_b$. The effects in S_c are called *concurrent effects*. A concurrent effect is an effect that is not present on (i.e., not applied to the state of) at least one replica. Each replica contains a subset of concurrent effects, which are applied to the common prefix b to compute the state at that replica. Let k denote the number of concurrent effects. Lower k values represent concrete executions where replicas are more-or-less in sync with each other. Conversely, high k values indicate executions characterized by, e.g., network partitions, process crashes, high network latency etc, that result in divergence between replicas.

Note that representing executions as a pair of a common prefix state b, and a set S_c of concurrent effects is not any less general than the scheme used by operational semantics, which tracks the set A of *all* effects. We can let the former simulate the latter by setting $b = \iota$ (ι is the initial state) and $S_c = \mathsf{A}$. However, the advantage of using the (b, S_c) scheme instead of A is that it lets us perform bounded verification by allowing us to bound the amount of concurrency (i.e., the size k of the set S_c) without having to constrain the pre-state (b). The pre-state is constrained only inasmuch as the application allows it. For instance, in the `BankAccount` application, where the invariant allows the balance to be any non-negative quantity, setting $k = 3$ and $b \geq 0$ lets us explore all executions with an unknown (but non-negative) initial balance, and at most 3 concurrent effects, a setting sufficient to detect the anomalies given in Fig. 3.1. In practice, we find that small values of k are sufficient to discover all anomalies an application may exhibit under arbitrary asynchronous replication.

$\boxed{\Gamma \vdash e \downarrow \nu}$

S-Match-EffSym

$$\frac{\Gamma \vdash e \downarrow \nu_\bullet \quad \nu = \mathsf{oper}(\nu_\bullet) = W_i \qquad \Gamma \wedge \nu \vdash [\mathsf{arg}(\nu_\bullet)/x]\, e_1 \downarrow \nu_1 \quad \Gamma \wedge \neg\nu \vdash e_2 \downarrow \nu_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ W_i(x) \Rightarrow e_1\ \texttt{else}\ e_2 \downarrow \mathsf{if}\ \nu\ \mathsf{then}\ \nu_1\ \mathsf{else}\ \nu_2}$$

S-Match-BoolSym

$$\frac{\Gamma \vdash e \downarrow \nu_\bullet \quad \Gamma \wedge \nu \vdash e_1 \downarrow \nu_1 \quad \Gamma \wedge \neg\nu \vdash e_2 \downarrow \nu_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with true} \Rightarrow e_1\ \texttt{else}\ e_2 \downarrow \nu_\bullet ? \nu_1 : \nu_2}$$

S-Match-SetSym

$$\frac{\Gamma \vdash e \downarrow \nu_\bullet \quad H(\Gamma, [\nu_\bullet, x \cup y, e_1, e_2]) = z}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ x \cup y \Rightarrow e_1\ \texttt{else}\ e_2 \downarrow z}$$

Figure 3.6.: Symbolic evauation rules for λ_R expressions

Formalization

Symbolic execution is formalized in terms of evaluation relations for λ_R expressions ($\downarrow$), and λ_R programs ($\hookrightarrow$). These relations are symbolic counterparts to λ_R's big-step concrete expression evaluation relation ($\Downarrow$), and small-step program evaluation relation ($\longrightarrow$). The class of symbolic values is defined as follows:

$$\nu \ ::= \ c \mid x \mid \lambda x.\, e \mid \nu\, \nu \mid \nu ?\, \nu : \nu \mid \emptyset \mid \{\nu\} \mid \nu \cup \nu \mid W_{i \in [1,N]}(\nu)$$

Constants (integer and boolean) and variables are symbols. An application of a symbolic value to a symbolic value is a symbolic value. For instance, $x + y$ is an application of the built-in function $+$ to x, and the result to y. A guarded symbolic value is a value of the form $\nu_1 ?\, \nu_2 : \nu_3$. An example is $(x > 10)?2 : 3$. Sets of symbolic values are also symbolic values. Finally, application of an effect constructor to a symbolic value results in a symbolic value of type `eff`.

Based on their structure, symbolic values can be divided into two categories: values that are either constants or applications of the constructors (e.g., $W_{i\in[1,N]}$, $\{\cdot\}$ etc.) at the top level, and the rest (e.g., a variable or a guarded value). The values of the former kind are *destructible*, meaning that they can be deconstructed and matched against a pattern in a `match` expression, with execution proceeding as if it were a concrete execution. We let $\nu_{\downarrow}$ denote destructible symbolic values. In contrast, non-destructible values (denoted $\nu_{\bullet}$) require the symbolic execution to explicitly handle the case of such values being matched against patterns. Fig. 3.6 contains a few symbolic execution rules that illustrate the point. The rule S-MATCH-EFFSYM describes how non-destructible symbolic values of type `eff` are handled in a `match` expression. Γ is a conjunction of path constraints, which are simply boolean symbolic values. The scrutinee of `match` is a non-destructive `eff` value ($\nu_{\bullet}$), which is matched against an `eff` constructor. Unable to destruct $\nu_{\bullet}$, the rule evaluates both the branches under appropriate path constraints (involving the application of `oper` special function), and returns a guarded value. The rule `S-Match-BoolSym` does guarded symbolic execution over `match` expressions involving non-destructible boolean values. The rule S-MATCH-SETSYM describes a case where the symbolic execution cannot make progress even by constructing guarded values. Here, a set expression e evaluates a non-destructible symbolic value $\nu_{\bullet}$, which is matched against a union pattern $x \cup y$. Since the execution cannot determine whether ν matches $x \cup y$ or not (ν could be a variable, for example), it has no way to make progress. Attempts to execute both the branches (e_1 and e_2), and return a guarded value may lead to divergence if either of e_1 or e_2 contains a recursive call on either x or y (because each recursive call branches further, which never ends). The symbolic evaluation prevents this by halting the evaluation and returning a fresh symbol with the same type as the `match` expression. It uses a (meta) function H for this purpose. The function H essentially performs memoization; it takes enough arguments to ensure that if the symbolic execution evaluates the same `match` expression again in the same context (Γ), it returns the same symbol (z). To avoid cluttering the rule with technicalities,

we assume that the type binding for z is already present in Γ, and z does not occur free in the `match` expression. The rule S-APP deals

Example. Consider the following version of `apply_eff` , which is slightly modified from Sec. 3.2 to make it conform to the syntax of λ_R (for brevity, we use `D` for `Deposit` , and `W` for `Withdraw`):

$$\lambda(s : \texttt{int}).\lambda(e : \texttt{eff}).\ \texttt{match}\ e\ \texttt{with}\ \texttt{D}(a) \Rightarrow s + a$$
$$\texttt{else match}\ e\ \texttt{with}\ \texttt{W}(a) \Rightarrow s - a\ \texttt{else}\ s$$

The result of symbolically evaluating the body of the function is a guarded symbolic value shown below. We name it ν_{app}, and parameterize it on the free symbols e and s:

$$\nu_{app}(e, s) \quad = \quad \mathsf{if}\ (\mathsf{oper}(e) = \texttt{D})\ \mathsf{then}\ (s + \mathsf{arg}(e))$$
$$\mathsf{else\ if}\ (\mathsf{oper}(e) = \texttt{W})\ \mathsf{then}\ (s - \mathsf{arg}(e))\ \mathsf{else}\ s$$

∎

As mentioned previously, symbolic evaluation explores the state space of the application starting from a symbolic state (b) that satisfies the invariant (I), and assuming that the number of concurrent effects ($|S_c|$) never exceeds a fixed value k. We write S_c^k to explicitly denote a set S_c that has cardinality k. Let us name the k concurrent effects as $E_{i \in [1,k]}$. Thus:

$$S_c^k = \bigcup_{i=1}^{k} \{E_i\}$$

A replica state (s) is computed by applying a subset of S_c^k to b. Let us say an operation f executes against the state s at replica R and generates an effect ϵ. From the definition of vis, it follows that a subset of S_c^k effects at R is visible to ϵ; this subset can be constructed thus:

$$\bigcup_{i=1}^{k} \mathsf{if}\ \mathsf{vis}(E_i, \epsilon)\ \mathsf{then}\ \{E_i\}\ \mathsf{else}\ \emptyset$$

where $E_i \in S_c^k$. That is, the effect E_i is included in the set only if it is visible to ϵ. We call this set a projection of S_c^k on ϵ, and denote it as $S_c^k \rhd \epsilon$. We define what it means to apply such a set of visible effects by defining its denotation as follows:

$$\begin{aligned} [\![\emptyset \rhd \epsilon]\!] &= \lambda s.\, s \\ [\![(S \uplus \{E_i\}) \rhd \epsilon]\!] &= [\![S \rhd \epsilon]\!] \circ \lambda s.\, \mathsf{if}\ \mathsf{vis}(E_i, \epsilon)\ \mathsf{then}\ ([\![E_i]\!]\, s)\ \mathsf{else}\ s \end{aligned}$$

Intuitively, the state at replica R witnessed is $[\![S_c^k \rhd \epsilon]\!]\, b$, where b is the common prefix state.

Having defined what it means to apply a projection, we can now define a symbolic equivalent of the concrete small-step evaluation rule. The rule represents the global state as a tuple of the common prefix and concurrent effect set (b, S_c^k), instead of the set of all effects (A):

$$\frac{(\lambda x.e)\ ([\![S_c^k \rhd \epsilon]\!]\ b) \downarrow \epsilon}{((b, S_c^k),\ [\lambda x.e]_{B \to \texttt{eff}}\ ||\ \pi) \overset{\epsilon}{\hookrightarrow} \pi} \quad \text{[S-Oper]}$$

The conclusion of the rule indicates that the λ_R program $[\lambda x.e]_{B \to \texttt{eff}}\ ||\ \pi$ symbolically reduces to π under the state (b, S_c^k) while generating an effect ϵ. The antecedent requires that the effect ϵ be the result of symbolically executing $\lambda x.e$ against a state that applies a subset of effects visible to ϵ to b.

We now redefine our notion of safety to consider k-bounded symbolic execution:

Definition 3.4.2 (k-safety) *A symbolic execution of a program π bounded by k concurrent effects is k-safe with respect to invariant I if $\forall b, k, S_c^k, \epsilon, \epsilon_f$ s.t:*

- $I([\![S_c^k \rhd \epsilon_f]\!]\, b) \downarrow \texttt{true}$
- $\forall \pi, \pi', ((b, S_c^k), \pi) \overset{\epsilon}{\hookrightarrow} \pi'$

then $I([\![(S_c^k \cup \{\epsilon\}) \rhd \epsilon_f]\!]\, b) \downarrow \texttt{true}$

In the above definition, ϵ is the effect generated by the small-step reduction of the program π, whereas ϵ_f is an effect generated by *some* operation f witnessing the state. The first premise asserts that f initially sees an invariant-satisfying state regardless

of what subset of concurrent effects it witnesses. In the context of a replicated state system, it means that all replicas initially satisfy the invariant. Invariant satisfiability is defined by asserting that the symbolic value resulting from symbolically evaluating the invariant function is equal to `true`. The second premise (interpreted using the S-Oper rule) states that the effect ϵ is the result of symbolically evaluating an operation in π against a state containing a subset of concurrent effects. Concretely, it means that ϵ is an effect generated by executing an operation in π at some replica. Under these premises, proving k-safety of π requires proving that f continues to see an invariant satisfying state even when the set S_c^k of concurrent effects is extended with ϵ. That is, even if f is executed on a replica that includes the newly generated effect ϵ, it still sees an invariant satisfying state.

Note that, since k-bounded verification only explores a limited state space, k-safety does not guarantee the unconditional safety of λ_R programs, but it does guarantee that any counterexample to safety it discovers is a real counterexample, assuming the invariant I is a complete specification of valid program states, i.e., any assignment to the symbolic pre-state b that satisfies I is a valid assignment[5]. We call the counterexample discovered by symbolic execution of program π as a witness to the k-unsafety of π, and a counterexample discovered by the concrete execution as a witness to the unsafety of π. The soundness of bounded verification can now be stated thus:

Theorem 3.4.1 *If a λ_R program is k-unsafe with witness ω, then it unsafe with witness ω, provided that its invariant I is a complete specification of valid program states.*

The proof of the theorem follows from the fact that the symbolic execution computes an underapproximation of the set of behaviors a λ_R program can exhibit. Thus, any execution captured by the symbolic encoding of the program is a valid program execution, including an unsafe execution.

[5]This assumption is already captured by Def. 3.4.1, which defines a valid concrete execution as one starting from any invariant satisfying state. To avoid potential sources of confusion, we explicitly qualify our soundness guarantees with this assumption wherever required.

The soundness guarantee of Theorem 3.4.1 is conditional to the invariant I being a complete specification of valid program states. If on the other hand I is only a partial specification, then symbolic execution may capture executions that do not manifest concretely, thereby leading to false k-safety violations. However, as explained in Sec. 3.4.1, completeness of I is a reasonable assumption to make in the context of database programs, which are often executed against databases populated independently of such programs. In this setting, the only valid assumptions about the database state are the stated integrity constraints (I).

Example

Let us say we would like to verify the 3-safety of a `withdraw(amt)` operation, i.e., safety of `withdraw(amt)` assuming three concurrent effects $S_c^3 = \{E_1, E_2, E_3\}$.

As per Def. 3.4.2 the invariant can be assumed to be valid in any pre-state. That is, for some effect ϵ_f, assume `inv_non_neg_bal` $(\llbracket S_c^3 \rhd \epsilon_f \rrbracket\, b) \downarrow$ `true`. The term $\llbracket S_c^3 \rhd \epsilon_f \rrbracket$ denotes the application of effects visible to ϵ_f via `BankAccount`'s `apply_eff`. Recall (from the previous example) the symbolic value $\nu_{app}(e, s)$, which is the result of symbolically executing `apply_eff` on a symbolic effect e and a symbolic state s. Since $\llbracket S_c^3 \rhd \epsilon_f \rrbracket$ applies an effect $E_i \in S_c^3$ only if $\mathsf{vis}(E_i, \epsilon_f)$, it reduces to the following symbolic value (`let` bindings are used for the sake of clarity):

$$\begin{aligned}
&\texttt{let } s_1 \;=\; \mathsf{if}\ \mathsf{vis}(E_1, \epsilon_f)\ \mathsf{then}\ \nu_{app}(E_1, b)\ \mathsf{else}\ b\ \texttt{in}\\
&\texttt{let } s_2 \;=\; \mathsf{if}\ \mathsf{vis}(E_2, \epsilon_f)\ \mathsf{then}\ \nu_{app}(E_2, s_1)\ \mathsf{else}\ s_1\ \texttt{in}\\
&\qquad\quad \mathsf{if}\ \mathsf{vis}(E_3, \epsilon_f)\ \mathsf{then}\ \nu_{app}(E_3, s_2)\ \mathsf{else}\ s_2
\end{aligned}$$

Let us name the above symbolic value $\nu_s^{pre}(\epsilon_f)$. The parameterization on ϵ_f underscores that ϵ_f could be any effect witnessing the pre-state. Since the invariant is valid initially, $\nu_s^{pre}(\epsilon_f) \geq 0$.

Next, Def. 3.4.2 lets us assert the conditions under which the program π generates the effect ϵ. In other words, it lets us capture the local safety of ϵ. Here, π is simply the `withdraw(amt)` operation, and ϵ is a `Withdraw` effect. The operation generates

the effect only if the balance it reads is not less than `amt`. The symbolic execution captures this condition as a path constraint (i.e., a logical formula whose satisfiability determines the feasibility of the current program path), which is then allowed to be asserted as a premise of k-safety. The balance that `withdraw(amt)` reads is $[\![S_c^3 \rhd \epsilon]\!]\, b$, which expands to $\nu_s^{pre}(\epsilon)$. Thus, the premise that `withdraw(amt)` is locally-safe translates to the assertion $\nu_s^{pre}(\epsilon) \geq$ `amt`.

Having captured the two premises of Def. 3.4.2 as constraints on symbolic values, we are now required to prove that if we include ϵ in the set of concurrent effects, the invariant still evaluates to `true`, i.e., `inv_non_neg_bal` $([\![\{S_c^3 \cup \{\epsilon\}\} \rhd \epsilon_f]\!]\, b) \downarrow$ `true`. The expression $[\![\{S_c^3 \cup \{\epsilon\}\} \rhd \epsilon_f]\!]\, b$ essentially applies ϵ to the result of $[\![S_c^3 \rhd \epsilon_f]\!]$ if and only if $\mathsf{vis}(\epsilon, \epsilon_f)$. Recalling that the result of applying a symbolic `BankAccount` effect e on a symbolic state s is $\nu_{app}(e, s)$, and that $[\![S_c^3 \rhd \epsilon_f]\!] = \nu_s^{pre}(\epsilon_f)$, we deduce that the result of $[\![\{S_c^3 \cup \{\epsilon\}\} \rhd \epsilon_f]\!]\, b$ is the following symbolic value:

$$\mathsf{if}\ \mathsf{vis}(\epsilon, \epsilon_f)\ \mathsf{then}\ \nu_{app}(\epsilon, \nu_s^{pre})(\epsilon_f)\ \mathsf{else}\ \nu_s^{pre}(\epsilon_f)$$

Let us call the above value $\nu_s^{post}(\epsilon_f)$. The proof obligation generated by Def. 3.4.2 is:

$$\nu_s^{pre}(\epsilon_f) \geq 0 \wedge \nu_s^{pre}(\epsilon) \geq \texttt{amt} \Rightarrow \nu_s^{post}(\epsilon_f) \geq 0$$

The validity of the above implication is equivalent to the unsatisfiability of the following conjunction:

$$\nu_s^{pre}(\epsilon_f) \geq 0 \wedge \nu_s^{pre}(\epsilon) \geq \texttt{amt} \wedge \neg(\nu_s^{post}(\epsilon_f) \geq 0)$$

An SMT solver, such as Z3 determines the conjunction to be satisfiable[6], thus proving that `withdraw` is 3-unsafe. The counterexample that Z3 returns involves assigning `amt` $= 1$, and making E_1 a `Deposit(1)` effect that is visible to the current effect $\epsilon =$ `Withdraw(1)`, but not making it visible to the (reference) effect ϵ_f. The counterexample is visualized in Fig. 3.7a, and is an abstract representation of the concrete anomaly described in Fig. 3.1b. Fixing the anomaly (as described in the

[6]In general, the efficacy of bounded verification in Q9 depends on the ability of the solver to reason about the theories required to encode the path constraints of the program and the invariant. Fortunately, there exist decidable theories, such as linear arithmetic, that are useful in practice.

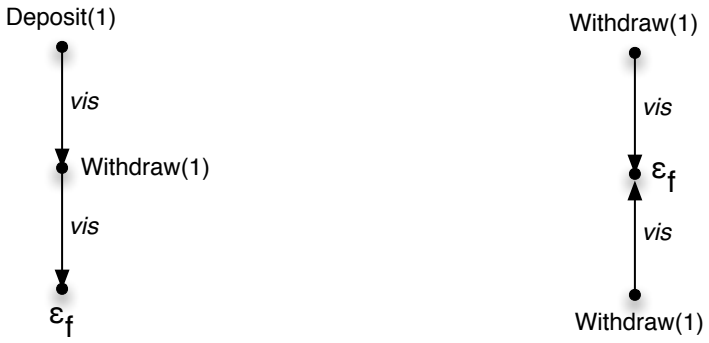


Figure 3.7.: Counterexamples to 3-safety.

next section), and rechecking the satisfiability lets us discover another counterexample, visualized in Fig. 3.7b. This counterexample is an abstraction of the concurrent withdraws anomaly of Fig. 3.1a. Fixing the second anomaly is enough to show that the constraints are unsatisfiable, thus `withdraw` is 3-safe.

3.4.4 Automated Repair

Once a counterexample demonstrating an anomaly has been discovered, Q9 helps to automatically repair the application by appropriately strengthening the consistency of the offending operation. We equip $Q9$ with a set of *consistency levels*, each designed to exempt a few classes of anomalous executions. Realization of these levels incurs a performance penalty in proportion to their strength (stronger consistency levels prohibit more anomalies and incur heavier penalty). The challenge is to determine the weakest consistency model that prohibits the anomaly exhibited by the counterexample. Fortunately, this step can be automated by observing that consistency levels can be captured in the same abstract language as the counterexamples that the symbolic execution discovers.

Consider the counterexample execution depicted in Fig 3.7a. We can formally express behaviors that admit this execution as the following counterexample (call it φ_{cex}):

$$\mathsf{oper}(E_1) = \texttt{Deposit} \wedge \mathsf{oper}(\epsilon) = \texttt{Withdraw} \wedge \mathsf{sameobj}(E_1, \epsilon) \wedge \mathsf{sameobj}(\epsilon, \epsilon_f)$$
$$\wedge \mathsf{vis}(E_1, \epsilon) \wedge \mathsf{vis}(\epsilon, \epsilon_f) \wedge \neg\mathsf{vis}(E_1, \epsilon_f) \wedge \neg\mathsf{vis}(\epsilon_f, E_1)$$

Given this characterization, we are interested in finding the weakest consistency assignment to `withdraw` (the operation that generated ϵ) that would prevent the counterexample. Here, we are aided by the fact that consistency levels can be specified in terms of the anomalies they prohibit. For instance, consider *causal write*, a consistency level that ensures a write is applied at a replica only after all the causally preceding writes have been applied. Thus, if ϵ is a causal write, then any anomalous execution involving three effects a, ϵ, and b, where (i). a causally precedes ϵ, and (ii). ϵ is visible to b, but (iii). a is not visible to b, is prohibited. Causal precedence is effectively captured by the happens-before relation (hb), which is a composition of vis and so (recall: $\mathsf{hb} = (\mathsf{vis} \cup \mathsf{so})^+$). Thus, if ϵ is a causal write, then the following must be true:

$$\forall a, b.\ \neg(\mathsf{hb}(a, \epsilon) \wedge \mathsf{vis}(\epsilon, b) \wedge \neg\mathsf{vis}(a, b))$$

Or, equivalently:

$$\forall a, b.\ \mathsf{hb}(a, \epsilon) \wedge \mathsf{vis}(\epsilon, b) \Rightarrow \mathsf{vis}(a, b)$$

If we name the above proposition φ_{cw}, and can prove that $\varphi_{cex} \wedge \varphi_{cw}$ is `UNSAT`, then it would be sufficient to make `withdraw` a causal write to prevent this anomaly. An off-the-shelf SMT solver like Z3 confirms that this is indeed the case[7]. Similar reasoning can be applied to second counterexample execution in 3.7b to determine that `withdraw` also needs to be totally-ordered w.r.t other `withdraw`s, i.e., it must be a *Total Write*.

The state space of consistency models found in the literature and implemented on various systems can be characterized in terms of a finite partially-ordered lattice [14],

[7] We adopt an approach similar to [63] to encode an overapproximation of hb, a transitive closure relation, in first-order logic

Table 3.1.: Consistency Models

Model Name	Specification	Description
Causal Write (CW)	$\forall a, b.\ \mathsf{hb}(a, \epsilon) \wedge \mathsf{vis}(\epsilon, b) \Rightarrow \mathsf{vis}(a, b)$	A write ϵ is applied only after all the causally preceding writes (a) are applied. [64]
Monotonic Write (MW)	$\forall a, b.\ \mathsf{so}(a, \epsilon) \wedge \mathsf{vis}(\epsilon, b) \Rightarrow \mathsf{vis}(a, b)$	A write ϵ is applied only after all the previous writes (a) from the session are applied [23].
Total-Order Write (TW)	$\forall a.\ \mathsf{oper}(a) = \mathsf{oper}(\epsilon) \wedge a \neq \epsilon \Rightarrow \mathsf{vis}(a, \epsilon) \vee \mathsf{vis}(\epsilon, a)$	All writes of the same operation as ϵ are applied in the same order everywhere.
SC Write (SC)	$\forall a.\ \mathsf{sameobj}(a, \epsilon) \wedge a \neq \epsilon \Rightarrow \mathsf{vis}(a, \epsilon) \vee \mathsf{vis}(\epsilon, a)$	All writes on the same object as ϵ are applied in the same order everywhere.
Atomicity (ATOM)	$\forall a, b, c.\ \mathsf{txn}(\tau, \{a, b\}) \wedge \neg\mathsf{txn}(\tau, \{c\}) \wedge \mathsf{vis}(a, c) \wedge \mathsf{sameobj}(b, c) \Rightarrow \mathsf{vis}(b, c)$	Writes from a transaction τ are applied atomically
Parallel Snapshot Isolation (PSI)	$\mathsf{ATOM} \wedge \forall a, b.\ \mathsf{txn}(\tau, \{a\}) \wedge \neg\mathsf{txn}(\tau, \{b\}) \wedge \mathsf{sameobj}(a, b) \Rightarrow \mathsf{vis}(a, b) \vee \mathsf{vis}(b, a)$	Writes from a transaction τ are made SC, and applied atomically.

where the partial order denotes the relative strength of the models under consideration. It is therefore possible to determine the consistency level of an operation by systematically traversing the lattice and checking whether each consistency model is sufficient to prevent the counterexample. Among the consistency models that are at the same level of the lattice, the order of traversal can be heuristic, perhaps based

on their relative run-time costs on a specific system. Systematic traversal of the consistency lattice is indeed the search strategy adopted by Q9.

The consistency models that Q9 considers in its search are shown in Table 3.1. The relation txn relates a transaction (its name τ) to the set of its constituent effects. Note that Atomicity is a property of a transaction. It is in fact a transaction's baseline consistency level in Q9. A transaction's consistency and isolation properties can also be strengthened in various ways, just like an operation's. One such way is to obtain a (conceptual) write lock on an object each time a write is performed, releasing all locks only when the transaction commits. The resultant consistency level, called Parallel Snapshot Isolation (PSI) [45], is specified in Table 3.1. As shown, it results in the writes outside a PSI transaction τ being totally ordered with τ's writes on the same objects. Other consistency levels found in the literature, e.g., Session Guarantees [23], can be specified in a similar way.

3.5 Transactions

In our exposition thus far, we have focused on operations that generate a single effect. Q9 programming framework also supports transactions that operate on multiple RDT objects, and generate multiple effects. To deal with such transactions, symbolic execution and verification require a few extensions, which we describe below.

First, the concept of state has to be generalized from a single object to a collection of objects. Thus, the common prefix state is $\overline{b}$ instead of b, and b_i denotes the common prefix state for the i'th object. The denotation of an effect ϵ, i.e., $[\![\epsilon]\!]$ is now a function that operates on a sequence of objects, but only updates (i.e., computes an updated value for) the object for which ϵ was generated. Building on these refined notions of state and effect denotation, we now generalize the S-Oper symbolic execution rule to deal with transactions. In λ_R, we formalize transactions simply as functions that

accept multiple arguments, and generate a set σ of effects. The generalized S-OPER is shown below:

$$\frac{S \subseteq S_c^k \quad (\lambda \overline{x}.e)\ (\llbracket S \rrbracket\ \overline{b}) \downarrow \sigma \qquad \forall (\epsilon' \in S), (\epsilon \in \sigma).\mathsf{sameobj}(\epsilon', \epsilon) \Rightarrow \mathsf{vis}(\epsilon', \epsilon)}{((\overline{b}, S_c^k),\ [\lambda \overline{x}.e]_{B \to \texttt{eff}}\ ||\ \pi) \overset{\sigma}{\hookrightarrow} \pi} \quad \text{[S-OPER]}$$

The rule reflects the system model of Q9, where a transaction, is executed at a single replica atomically, leading to all of its effects (ϵ) witnessing the same state that includes a set $S \subseteq S_c^k$ of concurrent effects. The quantified assertion in the premise captures the constraint that all effects $\epsilon \in \sigma$ witness the same set of concurrent effects that are on the same object as ϵ.

Safety is defined w.r.t an invariant function I, which relates multiple objects, i.e., it is a boolean function on multiple objects. The generalized k-safety definition that extends k-safety to transactions is defined thus:

Definition 3.5.1 (k-safety) *A symbolic execution of a program π bounded by k concurrent effects is k-safe with respect to invariant I if $\forall b, k, S, S_c^k, \epsilon, \sigma$ s.t:*

- $S \subseteq S_c^k$
- $I(\llbracket S \rrbracket\ b) \downarrow \texttt{true}$
- $\forall \pi, \pi', ((\overline{b}, S_c^k), \pi) \overset{\sigma}{\hookrightarrow} \pi'$

then $I((\llbracket S \rrbracket \circ \llbracket \sigma \rrbracket)\ b) \downarrow \texttt{true}$

The generalized definition is similar to the previous definition in the sense that it allows us to assume invariant on *any* subset S of the pre-state, and asks us to prove the invariant in the post state when S is extended with *all* the effects (σ) generated by the transaction, thus guaranteeing atomicity. Note that $\llbracket S \rrbracket \circ \llbracket \sigma \rrbracket$ denotes the functional composition of denotations of the sets S and σ. With the updated k-safety definition, anomaly detection and repair can work just as described in the previous section.

3.6 Implementation and Evaluation

The Q9 programming framework is implemented in OCaml as a collection of data and module type definitions, and modules that implement various CRDT semantics, such as counters, sets, maps, and boolean flags [65]. The Q9 symbolic execution engine is implemented as a compiler pass that follows typechecking in the OCaml 4.03 compiler[8]. Its first component is a translator that translates high-level RDT programs with implicit effects to their intermediate representation with explicit effects in preparation for analysis and verification. The second component performs bounded verification, given the k-bound as an input, and works in a tight loop with an SMT solver. The third component handles consistency repair. Verification progresses one operation at a time, followed by one transaction at a time. Each operation/transaction is verified for safety against its current consistency setting, starting with eventual consistency; this baseline reflects the system model described in Sec. 3.3. If verification fails, the verifier obtains a counterexample from the solver, computes its abstract representation, and passes it on to the repair engine, which then traverses a lattice of consistency models as described in Sec. 3.4.4, using the solver to check if a particular model is sufficient to preempt the counterexample. It returns the weakest such model to the verifier, which repeats verification with the new setting. This process continues until the verification of the operation/transaction succeeds, or the top of the consistency lattice has been reached, and no consistency setting was found to be adequate to guarantee safety[9].

The main component of the verifier is a symbolic execution engine that executes the body of an operation/transaction against symbolic inputs. The crux of the symbolic execution algorithm is as described in Sec. 3.4.3, but the engine also includes a number of optimizations aimed at rewriting symbolic values so as to keep their size

[8] `https://github.com/tycon/q9`

[9] This might happen if the consistency lattice given to the analysis is not strong enough. If the lattice describes the consistency levels of a data store, then the failure means that the safety of the program cannot be guaranteed on that store.

roughly proportional to the length of the program traversed. An example of such a rewrite rule is shown below:

$$\mathsf{if}\ (\mathsf{if}\ \nu_1\ \mathsf{then}\ \nu_2\ \mathsf{else}\ \nu_3)\ \mathsf{then}\ \nu_4\ \mathsf{else}\ \nu_5 \longrightarrow \mathsf{if}\ ((\nu_1 \wedge \nu_2) \vee (\neg\nu_1 \wedge \nu_3))\ \mathsf{then}\ \nu_4\ \mathsf{else}\ \nu_5$$

Symbolic execution generates verification conditions (VCs) based on the k-safety definition (Def. 3.4.2). A VC-Encode component encodes these VCs as satisfiability queries in Z3, after asserting the required axioms on special relations such as vis and so (e.g., so is transitive, hb is irreflexive etc). If the query is satisfiable, then a model is obtained and passed on to the verifier, which then uses it for consistency repair as described above.

3.6.1 Verification Experiments

To test the effectiveness of Q9 in detecting and fixing replication anomalies, we ported a range of applications, including several standard database benchmarks, to the Q9 programming model, and verified them under various values of bound (k). The applications are briefly described below:

- **eBanking**: A banking application that extends the running example with additional functionality.
- **Twissandra**: A Twitter-like microblogging application based on a popular Cassandra application with the same name [66].
- **RUBiS**: Rice University Bidding System [67] - an eBay-like auction site.
- **eCart**: An eCommerce application that lets users jointly control a shopping cart.
- **TPC-C**: A database benchmark that emulates a warehouse application.
- **TPC-E**: A database benchmark that emulates a brokerage application.

Table 3.2.: A sample of the anomalies found and fixes discovered by Q9

Oper/Txn	Violated Inv.	Anomalies	Fix
Twissandra			
`txn_new_tweet`	`Timeline` $\xrightarrow{\text{ref}}$ `Tweet`	Write to `Timeline` is applied before the previous write to `Tweet`	MW
`add_username`	Uniqueness of usernames	Concurrent checks for the uniqueness of a username succeed independently, resulting in duplicates.	TW
RUBiS			
`txn_bid_for_item`	`WalletBids` $\xrightarrow{\text{ref}}$ `Bids`	Write to `WalletBids` is applied before the previous write to `Bids`	MW
eCart			
`checkout`	$\forall(a \in \text{stock}).\ \text{qty}(a) \geq 0$	Concurrent checkouts of same items succeed independently resulting in negative stock.	TW
TPC-C			
`txn_new_order`	Per-district order ids are unique and sequential	Concurrent `txn_new_order` transactions read the same `next_oid` from a `District` record, and insert new orders with this id, resulting in orders with duplicate ids.	PSI
TPC-E			
`complete_trade`	`Broker` $\xrightarrow{\text{ref}}$ `COUNT(Trade)`	Update to `Trade` is applied before the previous insert to `Trade`.	CW
`txn_trade_result`	`Broker` $\xrightarrow{\text{ref}}$ `COUNT(Trade)`	Concurrent `trade_result` txns complete the same trade, and independently increment `Broker`' `num_trades`.	PSI

Both TPC-C and TPC-E, which were originally written for testing relational databases [56], were reimplemented to leverage CRDTs to make them amenable for execution in a distributed environment. Specifically, each TPC-C/TPC-E table translates into an RDT. For instance, TPC-C's `Order` table is implemented by an `Order.t` RDT, which internally uses a set CRDT to manage its contents. Every `INSERT, UPDATE` and `DELETE` operation on the table is implemented by a dedicated operation on the RDT. For example, a SQL `INSERT` operation that inserts an order record is implemented by an operation `do_add_order` that adds the order information to the set. The operation is eventually translated into an `AddOrder` effect, and symbolic reasoning is performed on such effect representations.

Each application described above defines one or more invariants that capture its salient safety properties. During verification, we found anomalies that violate a subset of the invariants for each application. Table 3.2 presents an interesting sample of the violations we found. These anomalies can be broadly classified into the following categories:

- **(In)equality invariants**: Invariants on integers involving equalities and inequalities. An example is `bal` ≥ 0 found in the eBanking application.

- **Uniqueness invariants**: Invariants that require a value of a particular type to be unique. An example is TPC-C's requirement that every order under a district to have a unique identifier. Another example is the requirement that user names be unique in Twissandra.

- **One-to-one referential integrity**: Invariants that require references between objects to be valid. That is, if an object of type A refers to another object of type B, then the corresponding B object must be present whenever an A object is present. We denote such one-to-one referential integrity relations as $A \xrightarrow{\text{ref}} B$, whenever A and B are both objects. For example, Twissandra requires references from users' timelines to tweets to be valid.

- **One-to-many referential integrity**: Whenever an object of type A refers to a certain property (f) of (some) objects of type B, then the property must hold of the corresponding B objects whenever an A object is present. We denote such a relation as $A \xrightarrow{\text{ref}} f(B)$, and call it one-to-many referential integrity provided A and B are both object types, and f is a function from B to some base type. Usually, whenever $A \xrightarrow{\text{ref}} f(B)$ there is also an inverse one-to-one relation, i.e., $B \xrightarrow{\text{ref}} A$. An example of one-to-many referential integrity is the `Order` $\xrightarrow{\text{ref}}$ `COUNT(OrderLine)` invariant in TPC-C, which requires an order's `o_ol_count` field to accurately reflect the number of `OrderLine` records referring back to the order. Another example is TPC-C's `Warehouse` $\xrightarrow{\text{ref}}$ `SUM(History)` that requires a warehouse's year-to-date balance to agree with its ledger stored in the `History` table. Similar constraints are also present in TPC-E.

Table 3.2 lists various operations and transactions that violate the invariants of the kind described above. For each violation, the table briefly describes the anomaly that was discovered, and also lists the consistency level suggested to preempt the anomaly (c.f., Table 3.1 for a description of consistency levels). As an example of the kind of repair Q9 was able to perform, consider TPC-C's `txn_new_order` transaction, which adds a new `Order` record with an id (`Order.o_id`) equal to the sequence number of the next order for the corresponding district (`District.next_o_id`). The transaction also increments the district's order sequence number. During the verification of `txn_new_order`, Q9 was able to discover an anomaly that violates TPC-C's safety requirement that every order must have a unique id. The anomaly consists of two concurrent `txn_new_order`s reading the same id of the district's next order (`District.next_o_id`), and inserting duplicate `Order` records with that id. Subsequent to the discovery of anomaly, Q9 was also able to use the counterexample to reason that if `txn_new_order` is executed under Parallel Snapshot Isolation (PSI) consistency model, then the anomaly can be preempted. While Q9 found a violation of uniqueness invariant in TPC-C, through a similar reasoning it found a violation of

Table 3.3.: Verification Statistics

Application	Opers	Txns	Anomalies found	Max k for an anomaly	Max time (s) for an anomaly	Max k verified
eBanking	3	2	3	5	0.28	60
Twissandra	20	10	5	5	6.59	50
RUBiS	17	6	5	5	3.03	50
eCart	10	5	5	6	1.09	60
TPC-C	18	5	6	10	51.79	18
TPC-E	44	10	3	10	113.53	17

one-to-many referential integrity invariant (`Broker` $\xrightarrow{\text{ref}}$ `COUNT` (`Trade`)) in TPC-E. The fix, again, is to strengthen the transaction's consistency level to PSI.

Q9 was also able to perform the reasoning in the opposite direction, i.e., it was able to discover that certain transactions need not be atomic when we made atomicity optional for transactions. Instead, Q9 suggested weaker alternatives to atomicity that are nonetheless safe *in that context*. For instance, consider `txn_new_tweet` transaction in Twissandra, which adds a new tweet to the `Tweet` table, and then adds the corresponding tweet `id` to a subset of objects in the `Timeline` table. Without atomicity (ATOM), the transaction (temporarily) violates the referential integrity between timelines and tweets if the latter write to `Timeline` is applied before the former write to `Tweet` , and the intermediate state becomes visible to an operation. While atomicity is sufficient to restore the safety, it is however not necessary; Q9 discovers that the anomaly can be preempted by executing the transaction under Monotonic Writes consistency model, which is weaker than atomicity, and is cheaper on some systems [23, 68]. Similar deductions were made for `txn_bid_for_item` transaction in RUBiS.

Table 3.3 shows various statistics quantifying the cost and efficacy of bounded verification. The table demonstrates Q9 was able to successfully find a number of anomalies for each application. The fact that anomalies were found in TPC-C and TPC-E might be surprising, considering that these benchmarks were well-studied. Clearly, as our experiments demonstrate, migration of concurrent applications to replicated environments is error-prone without tool support of the kind that Q9 provides.

The main takeaway from Table 3.3 is that all the anomalies were found within a small k bound, the maximum being 10 for TPC-C and TPC-E. The time that Q9 took to discover an anomaly is also reasonable, with the worst case being around 2 minutes for an anomaly in TPC-E. To test the limits of bounded verification through symbolic execution, we ran Q9 overnight (6-8 hours) on select (typically the most complex) transactions from each application and noted the maximum k for which it was able to verify the transaction (for TPC-C and TPC-E, we were able to verify all transactions). The maximum k thus found is listed against each application in Table 3.3. As shown, we were able to verify k-safety of some applications to k values that are significantly higher than the k values at which anomalies were discovered. Taken together, these statistics vindicate Q9's approach of using symbolic execution-driven bounded verification to discover anomalies in real distributed applications.

3.6.2 Validation Experiments

Since Q9 does bounded verification, we validate the consistency assignments discovered by Q9 by testing the applications on a distributed database. Our goal is two-fold. First, we would like to check if the consistency assignments discovered by Q9 through bounded verification are indeed sufficient to avert anomalies in the general case. And second, we would like to ascertain that any weaker consistency assignment invariably leads to the anomalies discovered by Q9 during verification; i.e., there are no false positives.

Our experimental setup consists of a distributed database equipped with RDT operations, with support for various consistency levels for operations and transactions. The distributed database itself is implemented as a shim layer on top of Cassandra [15] in the same vein as [61, 69]. We instantiated 2 replicas within the same data center with a inter-replica latency of 5ms, and 16 clients in total performing transactions. In order to tease out the anomalies that arise due to the asynchronous nature of the distributed database, we induced the shim layer to drop 50% of the effects transmitted over the network between the replicas. The replicas perform retransmission of the dropped effects until all the effects are received everywhere. Consequently, every replica receives every effect eventually, but the effects may be applied out of order.

We evaluated the TPC-C benchmark, where each client simulates the workflow for purchasing by performing a series of `NEW-ORDER`, `PAYMENT` and `DELIVERY` transactions. We call one such sequence of three transactions as a purchase. First, we ran the TPC-C workload with Q9 recommended consistency levels. We observed no anomalies for 1000 purchases per client. On the other hand, running the `NEW-ORDER` transaction at a level weaker than PSI consistency level (i.e., with only atomicity (ATOM)) led to anomalies; there were multiple orders with the same id, thus violating the safety requirement of TPC-C. The results demonstrate that Q9 is effective at finding appropriate consistency configuration: no anomalies were observed at the recommended consistency configuration, while any weaker configuration leads to manifestation of anomalies.

3.7 Related Work

There is a large body of work focused on the safety aspect of distributed applications with weakly-consistent replicated state.

CRDTs [65] define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. This property makes them especially attractive as a basis for dealing with replication in highly-

available distributed systems. However, reasoning over CRDTs can be difficult, and the nuances of their implementations relate poorly to understanding if and how they might preserve high-level application invariants.

Burckhardt *et al.* [70] presents an operational model of a replicated data store that is based on the abstract system model presented in [71]. As with the system model from Sec. 3.3, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. The verification strategy given in [71] is based on a replication-aware simulation argument that does not have an obvious automation pathway. Gotsman *et al.* [36] develop a rely-guarantee methodology and proof rule that can establish whether a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure preservation of a given data integrity invariant. The previous chapter presents a similar framework for reasoning about weak isolation [8, 9], which is different from weak consistency in the sense that it relates to groups of operations manipulating multiple objects. Ivy [72] is a tool for verifying the correctness of distributed protocols as sophisticated as Paxos [73].

These efforts require support from developers who must define deep specifications within a mechanized theorem prover [51, 74], state and prove various kinds of local and global assertions within the context of a program logic [36], and/or fix counterexamples that prevent inductive generalization [72]. Given such input, these approaches are capable of addressing important verification challenges in realistic distributed systems. The work presented here contrasts significantly from these other efforts because it demands no additional effort from the programmer other than a specification of a safety property. While Q9 cannot provide the same level of guarantees that full verification can, empirical evidence suggests that it nonetheless provides a high degree of utility, effectively serving as a principled anomaly detection tool for geo-replicated distributed programs, with minimal overhead demanded of the developer.

Context-bounded model-checking [75, 76], a bounded verification technique comparable to ours but for shared memory, critically assumes SC semantics, making it ineffective in discovering any of the anomalies discussed in Table 3.2.

Some of the challenges faced in reasoning about replicated data types are reminiscent of issues that arise in reasoning about weak memory systems. However, the differences between weak memory and weak consistency are sufficiently significant that reasoning techniques possible in the former are difficult to transparently migrate to the latter. In particular, weak memory models usually guarantee coherence (total ordering) of writes to a single location, a property not feasible under weak consistency since it requires global coordination [8]; reads in a single thread witness a monotonically progressing state under weak memory, but the same is not guaranteed under weak consistency; and, formalizations of the former reason over memory operations (reads and writes), whereas our formalization reasons at the level of abstract atomic effects (e.g., `Deposit`). This generalization allows us to scale the reasoning beyond litmus tests [77, 78] to real programs. Finally, repair mechanisms for weak memory are defined in terms of fences - low-level architecture-dependent artifacts that "flush" the local state. In contrast, the repair mechanisms for weak consistency are defined in terms of fine-grained high-level consistency models (Table 3.1) expressed in terms of causality, ordering, and visibility relations over groups of related objects. Collectively, these differences make reasonsing techniques proposed for weak (shared) memory ineffective in reasoning about weak (distributed) consistency, and mandate new formalizations of the kind proposed in this chapter.

Representative examples of testing and checking frameworks for distributed systems include MaceMC [79] a model-checker that discovers liveness bugs in distributed programs, and [80], a random testing tool that checks partition tolerance of NoSQL distributed database systems with varying consistency levels to enable high-availability. Q9 differs from these systems in significant ways: among other things, MaceMC does not consider safety issues related to replication, while Jepsen is purely a dynamic

analysis that does not leverage semantic properties of the application in searching for faulty executions.

Finally, there has been a vast body of work produced over the years that explore the use of symbolic execution as a means for more effective testing and bug-finding [81]. Surprisingly, we are unaware of any effort in this space that examines the applicability of symbolic execution to the problem of anomaly detection for highly-available geo-replicated distributed applications, a class of programs that are becoming increasingly important and pervasive.

4 DERIVATION OF MERGEABLE REPLICATED DATA TYPES

The Q9 programming framework presented in the previous chapter lets developers build distributed applications around a library of carefully-engineered convergent replicated data types (CRDTs). The focus there is on detecting and repairing consistency anomalies that manifest as high-level invariant violations in such applications. In this paper, we[1] propose a fundamentally different approach to programming with replicated state that enables the *automatic* derivation of correct distributed (replicated) variants of ordinary data types. Key to our approach is the use of *invertible relational specifications* of an inductive data type definition. These specifications capture salient aspects of the data type that are independent of its execution under any system model, thus greatly reducing the cognitive overhead of having to explicitly reason about low-level operational issues related to replication, asynchrony, visibility, etc that have been on the centerstage in the previous chapter. The specifications, however, provides sufficient guidance on structural properties maintained by the type (e.g., element ordering) critical to how we might correctly *merge* multiple instances in a replicated setting.

The approach presented in this chapter is based on a model of replication centered around *versioned states* and explicit *merges*. In particular, we model replicated state in terms of concurrently evolving *versions* of a data type that trace their origin to a common ancestor version. We assume implementations synchronize pairs of replicas by merging concurrent versions into a single convergent version that captures salient characteristics of its parents. The merge operation is further aided by context information provided by the *lowest common ancestor* (LCA) version of the merging versions.

[1]This work was done in collaboration with Samodya Abeysiriwardane, Swarn Priya, KC Sivaramakrishan, and Suresh Jagannathan.

```
module Counter: COUNTER =
struct
  type t = int
  let zero = 0
  let add x v = v + x
  let sub  x v = v - x
  let mult x v = x * v
  let read v = v
end
```

Figure 4.1.: A `Counter` data type in OCaml

Because the exact semantics of merging depends on the type and structure of replicated state, data types define merge semantics via a three-way merge function that merges pairs of concurrent versions in the context of their LCA version. The version control model of replication, therefore, allows any ordinary data type equipped with a three-way merge function to become a distributed data type. The full expressivity of merge functions can be exploited to define bespoke distributed semantics for data types that need not necessarily mirror their sequential behavior (i.e., distributed objects that are not linearizable or serializable), but which are nonetheless well-defined (i.e., convergent) and have clear utility.

Our focus in this paper is on deriving such correct merge functions automatically over arbitrarily complex (i.e, composable) data type definitions, and in the process, ascribe to them a meaningful and useful distributed semantics. By doing so, we eliminate the need to reason about low-level operational or axiomatic details of replication when transforming sequential data types to their replicated equivalents.

The approach presented in this chapter towards deriving data type-specific merge functions is informed by two fundamental observations about replicated data type state and its type. First, we note that it is possible to define an intuitive notion of a merge operation on concurrent versions of an abstract object state *regardless* of

its type. We illustrate this notion in the context of a simple integer counter, whose OCaml implementation is shown in Fig. 4.1. Suppose we wish to replicate the state of the counter across multiple machines, each of which is allowed to perform concurrent conflicting updates to its local instance. As long as clients just use the counter's `add` and `sub` operations, conflicts are benign - since integer addition and subtraction commute, `add` and `sub` operations can be asynchronously propagated and applied in any order on all replicas, with the resulting final state guaranteed to be the result of a linearization of all concurrently generated operations[2]. However, since integer multiplication does not commute with addition and subtraction, we cannot simply apply `mult` on various replicas asynchronously, and expect the state to converge. Global synchronization for every multiplication is certainly helpful, but is typically too expensive to be practical [7, 82] at scale. Under such circumstances, it is not readily apparent if we can define replicated counters that support multiplication and yet still have a well-defined semantics that guarantees all replicas will converge to the same counter state.

Fortunately, a state- and merge-centric view of replication lets us arrive at such a semantics naturally. In the current example, we view the replicated counter state as progressing linearly in terms of versions on different replica. Synchronization between replicas merges their respective (latest) versions into a new version in the context of their lowest common ancestor (LCA) version. We can define the merge operation by focusing on the *difference* between the LCA version and the state on each replica. Fig. 4.2 illustrates this intuition through an example. Here, two concurrent versions of a counter, 10 and 4, emerge on different replicas starting from a common ancestor (LCA) version 5. The first version 10 is a result of applying `mult 2` to LCA 5, whereas the second version 4 is a result of performing `sub 1`. To merge these concurrent versions, we ignore the operations and instead focus on the difference between each version and the LCA. Here, the differences (literally) are $+5$ and -1, respectively.

[2]Implicit here is the assumption of an operation-centric model of replication, where an operation is immediately applied at one replica, and lazily propagated to other replicas [49, 61, 65, 71].

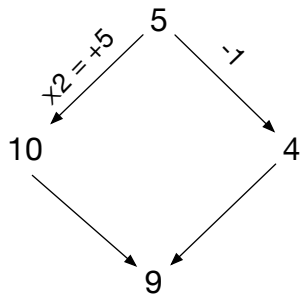


Figure 4.2.: `Counter` merge visualized

The merged version can now be obtained by *composing* the differences and applying the composition on the LCA. Here, composing $+5$ and -1 gives $+4$, and applying it to the LCA 5 gives us 9 as the merged version. In general, the merge strategy for an integer counter can be defined in terms of a three-way merge function as follows:

```
let merge l v1 v2 = l + (v1 - l) + (v2 - l)
```

In the above definition, l is the common ancestor version, whereas v_1 and v_2 are the concurrent versions. Note that the *mergeable* counter described above does not guarantee linearizability (for instance, if the concurrent operations in Fig. 4.2 are `mult 2` and `mult 3`, then the merge result would be 25 and not 30). Nonetheless, it guarantees convergence, and has a meaningful semantics in the sense that the *effect* of each operation is preserved in the final state. Indeed, such a counter type would be useful in practice, for instance, to record the balance in a banking application, which might use `mult` to compute an account's interest.[3]

The `Counter` example demonstrates the utility of a state- and merge-centric view of replication, and the benefit of using *differences* as a means of reasoning about merge semantics. Indeed, the abstract notion of a difference is general enough that it would appear to make sense (intuitively) to apply a similar approach for other data types. However, this notion does not easily generalize because data types often have complex inductive definitions built using other data types, making it hard to uniformly define concepts involving differences, their application, and their composition. It is

[3]Contrary to popular belief, real-world banking applications are weakly consistent [83]

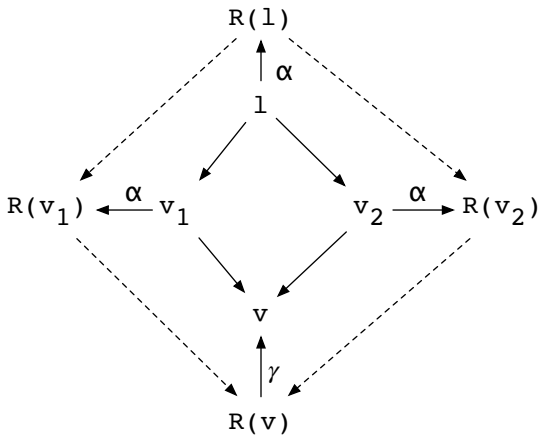


Figure 4.3.: Merging values in relational domain with help of abstraction (α) and concretization (γ) functions. Solid (resp. dashed) unlabeled arrows represent a merge in the concrete (resp. abstract) domain.

in this context that we find our second observation useful. While data types are by themselves quite diverse, we note that they can nonetheless be mapped losslessly to the rich domain of relations over sets, wherein relations so merged can be mapped back to the concrete domain to yield consistent and useful definitions of these aforementioned concepts. The semantics of a merge in the relational set domain, albeit non-trivial, is nonetheless standard in the sense that it is independent of the concrete interpretations (in the data type domain) of the merging relations, and hence can be defined once and for all. This suggests that the merge semantics for arbitrary data types can be automatically derived, given a pair of abstraction (α) and concretization (γ) functions for each data type that map the values of that type to and from the relational domain (the pair (α, γ) is an *invertible relational specification* of the type). The approach, summarized in Fig. 4.3, is indeed the one we use to automatically derive merges in this paper. The resultant *mergeable replicated data types* (MRDTs or *mergeable types*, for short) have well-defined distributed semantics in the same sense as the mergeable counter (i.e., a merge operation applied at each replica results in the same state that preserves the effects of all operations performed on all replicas).

To make MRDTs an effective component of a distributed programming model that yield tangible benefits to programmers, they must be supported by an underlying runtime system that facilitates efficient three-way merges and state replication. Such a system would have to track the provenance (i.e., full history) of concurrently evolving versions, facilitate detection and sharing of common sub-structure across multiple versions, allow efficient computation and propagation of succinct "diffs" between versions, and ideally also support persistence of replicated state. Fortunately, these demands can be readily met by a content-addressable storage abstraction underlying modern version control systems such as Git. In the later sections of this chapter, We describe Quark, an implementation of mergeable data types in OCaml built on top of a distributed, content-addressable, version-based, persistent storage abstraction that enables highly efficient merge operations. A detailed experimental study over a collection of data structure benchmarks as well as well-studied large-scale applications is also presented.

4.1 Motivation

Consider a queue data structure whose OCaml interface is shown in Fig. 4.4. `Queue` supports two operations: `push` a that adds an element a to the tail end of the queue, and `pop` that removes and returns the element at the head of the queue (or returns `None` if the queue is empty). We say the client that performed `pop` has *consumed* the popped element. For simplicity, we realize queue as a list of elements, i.e., we concretize the type `'a Queue.t` as `'a list` for this discussion. Like `Counter` with `mult`, `Queue`'s implementation does not qualify it as a CRDT, since `push` and `pop` do not commute. Hence, its semantics under (operation-centric) asynchronous replication is ill-founded as illustrated in Fig. 4.5.

The execution shown in Fig. 4.5a starts with two replicas, R_1 and R_2, of a queue containing the elements 1 followed by 2. Two distinct clients connect to each of the replicas and concurrently perform `pop` operations, simultaneously consuming 1. The

```
module Queue: sig
  type 'a t
  val push: 'a -> 'a t -> 'a t
  val pop: 'a t -> 'a option * 'a t
end = ...
```

Figure 4.4.: The signature of a queue in OCaml

`pop`s are then propagated over the network and applied at the respective remote replicas to keep them consistent with the origin. However, due to a concurrent `pop` already being applied at the remote replica, the subsequently arriving `pop` operation pops a different and yet-to-be-consumed element 2 in each case. The result is a convergent yet incorrect final state, where the element 2 vanishes without ever being consumed. Fig. 4.5b shows a very similar execution that involves `push`es instead of `pop`s. Starting from a singleton queue containing 1, two concurrent `push` operations push elements 2 and 3 resp. on different replicas. When these operations are eventually applied at the remotes, they are applied in different orders, resulting in the divergence of replica states. Fig. 4.5c shows another example of divergence, this time involving both `push`es and `pop`s. The execution starts with two replicas, R_1 and R_2, of a singleton queue containing the 1. Two `pop` operations are concurrently issued by clients, both (independently) consuming 1. The `pop`s are then applied at the respective remotes after a delay. During this delay, R_1 sees no activity, leaving the queue empty for R_2's `pop`, which effectively becomes a `Nop`. On R_2 however, a `push 2` operation is performed meanwhile, so when R_1's `pop` is subsequently applied, it pops the (yet unconsumed) element 2. As a result, the final state of the queue on R_2 is empty. Like the `pop`s, the `push 2` operation is also propagated and eventually applied on R_1, resulting in the final state on R_1 being a singleton queue. Thus the replicas R_1 and R_2 of the final state of the queue diverge, which preempts

any consistent semantics of the queue operations from being applied to explain the execution.

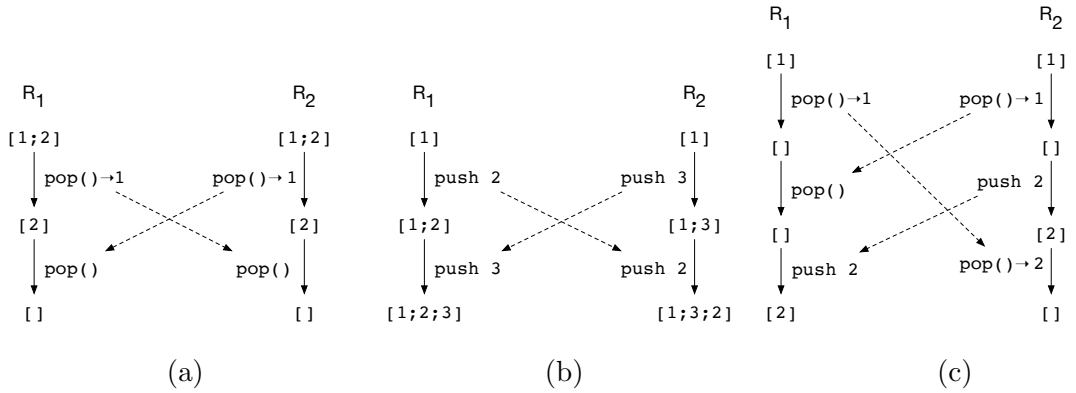


Figure 4.5.: Ill-formed queue executions

Bad executions such as those in Fig. 4.5 can be avoided if every queue operation globally synchronized. However, as explained before, enforcing global synchronization requires sacrificing availability (i.e., latency), an undesirable tradeoff for most applications [12]. It may therefore seem impossible to replicate queues with meaningful and useful semantics without losing availability. Fortunately, this turns out not to be the case. In the context of real applications, there exist implementations of highly available replicated queues whose semantics, albeit non-standard, i.e., not linearizable or serializable, have nonetheless proven to be useful. Amazon's Simple Queue Service (SQS) [84] is one such queue implementation with a non-standard *at-least-once delivery* semantics, which guarantees, among other things, that a queued message is delivered to a client for consumption at least once. Devoid of a formal context, such semantics may seem *ad hoc*; however, casting the `Queue` data type as a mergeable type would let us *derive* such semantics from first principles, thus giving us a formal basis to reason about its correctness.

Recall that our underlying execution model is based on state-centric model of replication with versioned state and explicit three-way merges (which we show how to synthesize). Under this model, two concurrent versions v_1 and v_2 of a queue can

independently evolve from a common ancestor (LCA) version l. The semantics of the queue under replication depends on how these versions are merged into a single version v (Fig. 4.3). The concurrent versions v_1 and v_2 would have evolved from l through several `push` and `pop` applications, however let us ignore the operations for a while and focus on the relationship between the queue states l, v_1, and v_2. Intuitively, the following relationships must hold among the three queues:

1. For every element $x \in l$, if $x \in v_1$ and $x \in v_2$, i.e., if x is not popped in either of the concurrent versions, then $x \in v$, i.e., x must be in the merged version. In other words, a queue element that was never consumed should *not* be deleted.

2. For every $x \in l$ if $x \notin v_1$ or $x \notin v_2$, i.e., if x is popped in either v_1 or v_2, then $x \notin v$. That is, a consumed element (regardless of how many times it was consumed) should never reappear in the queue.

3. For every $x \in v_1$ (resp. v_2), if $x \notin l$, that is x is newly pushed into v_1 (resp. v_2), then $x \in v$. That is, an element that is newly added in either concurrent versions must be present in the merged version.

4. For every $x, y \in l$ (resp. v_1 and v_2), if x occurs before y in l (resp. v_1 and v_2), and if $x, y \in v$, i.e., x and y are not deleted, then x also occurs before y in v. In other words, the order of elements in each queue must be preserved in the merged queue.

To formalize these properties more succinctly, we define two relations on lists: (1). A *membership* relation on a list l (written $R_{mem}(l)$) is a unary relation, i.e., a set, containing all the elements in l, and (2). An *occurs-before* relation on l (written $R_{ob}(l)$) is a binary relation relating every pair of elements x and y in l, such that x occurs before y in l. For a concrete list $l = [1; 2; 3]$, $R_{mem}(l)$ is the set $\{1, 2, 3\}$, and $R_{ob}(l)$ is the set $\{(1, 2), (1, 3), (2, 3)\}$. Note that for any list l $R_{ob}(l) \subseteq R_{mem}(l) \times R_{mem}(l)$, i.e., $R_{ob}(l)$ is only defined for the elements in $R_{mem}(l)$. Using R_{mem}, we

can succinctly specify the relationship among the members of l, v_1, v_2, and v, where v = `merge` l v_1 v_2, as follows[4]:

$$\begin{aligned} R_{mem}(v) \;=\; & R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \\ & \cup \quad R_{mem}(v_1) - R_{mem}(l) \quad \cup \quad R_{mem}(v_2) - R_{mem}(l) \end{aligned} \tag{4.1}$$

The left hand side denotes the set of elements in the merged version v. The right hand side is a union of three components: (1). The elements common among three versions l, v_1, and v_2, (2). The elements in v_1 not in l, i.e., newly added in v_1, and (3). The elements in v_2 not in l, i.e., newly added in v_2. Observe that we applied the same intuitions as the counter merge described at the beginning of this chapter to arrive at the above specification, namely merging concurrent versions by computing, composing and applying their respective *differences* to the common ancestor. However, we have interpreted the *difference* through the means of a relation over sets that abstracts the structure of a queue and captures only its membership property. Another important point to note is that the specification does not appeal to any operational characteristics of queues, either sequentially or in the context of replication.

Similar intuitions can be applied to manage the structural aspects of merging queues by capturing their respective *orders* via the *occurs-before* relation (R_{ob}) over lists, but after accounting for a couple of caveats. First, since $R_{ob} \subseteq R_{mem} \times R_{mem}$, $R_{ob}(v)$ has to be confined to the the domain of $R_{mem}(v) \times R_{mem}(v)$. Second, the order between a pair of elements where each comes from a distinct concurrent version is indeterminate, thus $R_{ob}(v)$ can only be underspecified. Taking these caveats into account, $R_{ob}(v)$ of the merged version v can be specified thus:

$$\begin{aligned} R_{ob}(v) \;\supseteq\; & (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \\ & \cup \quad R_{ob}(v_1) - R_{ob}(l) \quad \cup \quad R_{ob}(v_2) - R_{ob}(l)) \\ \cap \quad & (R_{mem}(v) \times R_{mem}(v)) \end{aligned} \tag{4.2}$$

[4]We elide parentheses for perspicuity. Any ambiguity in parsing should be resolved by assuming that $\cap$ and $-$ bind tighter than $\cup$

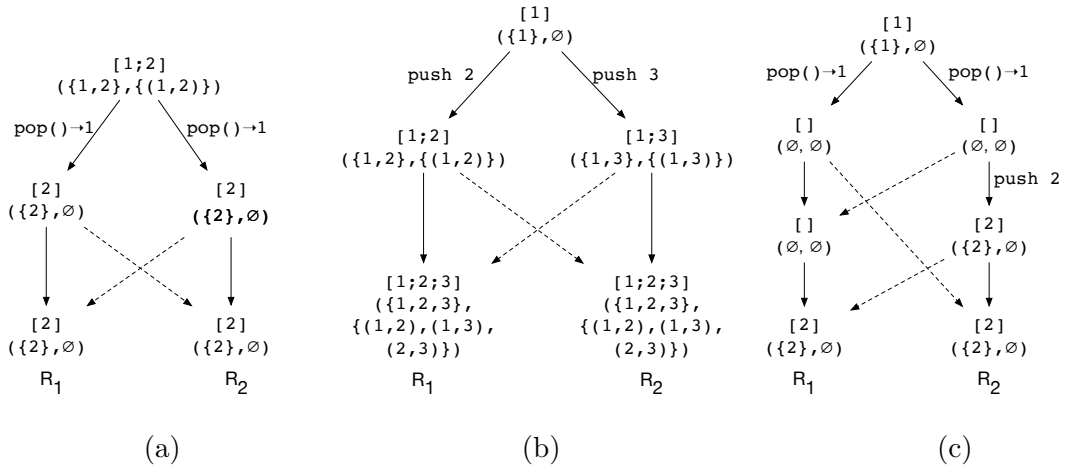


Figure 4.6.: State-centric view of queue replication aided by context-aware merges (shown in dotted lines)

Note the $\supseteq$ capturing the underspecification. The right hand side is essentially same as the right hand side of the R_{mem} equation (above), except that R_{ob} replaces R_{mem}, and we compute an intersection with $R_{mem}(v) \times R_{mem}(v)$ at the top level to confine $R_{ob}(v)$ to the elements in v. As mentioned earlier, the specification does not induce a fixed order among elements coming from different queues. To recover convergence, a merge function on queues can choose to order such elements through a consistent ordering relation, such as a lexicographic order.

The *membership* and *occurs-before* specifications together characterize the merge semantics of the queue data type that we derived from basic principles we enumerated above. We shall now reconsider the executions from Fig. 4.5, this time under a state-centric model of replication, and demonstrate how our merge specification leads us to a consistent distributed semantics for queue, which subsumes a *at-least-once delivery* semantics. The corresponding executions under this model are shown in Fig. 4.6.

Fig. 4.6a is the same execution in Fig. 4.5a with the dotted line representing a version propagation followed by a merge, rather than an operation propagation followed by an application. For each version, the R_{mem} and R_{ob} relations are shown below its actual value. If the version is a result of a merge, then we compute its

R_{mem} and R_{ob} sets using equations 4.1 and 4.2 of the merge specification above. For both the merges shown in the figure, the concurrent versions (v_1 and v_2) are the same: the singleton queue `[2]`, and their LCA version (l) is the initial queue `[1;2]`. Thus each concurrent version is a result of popping 1 from the LCA (which is consumed/delivered twice as acceptable under *at-least-once delivery* semantics). Intuitively, the result of the merge should be a version that incorporates the effect of popping 1, while leaving the rest of the queue unchanged from the LCA. This leaves the queue `[2]` as the only possible result of the merge (and the execution). Indeed, this is the result we would obtain if reconstruct the queue from the merged R_{mem} and R_{ob} relations shown in the figure. Execution in Fig. 4.6b corresponds to the one in Fig. 4.5b. Here we have two merges: one into R_1 and other into R_2. The concurrent versions for both the merges are the same: `[1;2]` and `[1;3]`, and their LCA is the queue `[1]`. Each concurrent version pushes a new element (2 and 3, resp.) to the queue *after* the existing element 1. Intuitively, the merged queue should contain both the new elements ordered after 1. Indeed, this is also what the merged R_{mem} and R_{ob} relations suggest. The order between new elements, however, is left unspecified by R_{ob}. As mentioned earlier, a consistent ordering relation has to be used to order such elements. Choosing the less-than relation, we obtain the result of the merge as `[1;2;3]`. In Fig. 4.6c, there are three merges: two into R_1 and one into R_2. For the first merge into R_1, the concurrent versions are both empty queues, and their LCA is the singleton queue `[1]`. Thus both versions represent a pop of 1, and their merged version, which reconciles both the pops, should be an empty queue, which is also what the merged relations suggest. The second merge into R_1 and the only merge into R_2, both merge an empty queue (`[]`) and a singleton queue `[2]`, with the LCA version being the initial queue `[1]`. While the version `[]` can be understood as resulting from the popping an element from LCA, the concurrent version `[2]` goes one step ahead and pushes a new element 2. Consequently, the merged version should be a queue not containing 1, but containing the new element 2, i.e., `[2]`, which is again consistent with the result obtained by merging R_{mem} and R_{ob} relations. Thus

```
let rec R_mem = function
  | [] -> ∅
  | x::xs -> {x} ∪ R_mem(xs)
```

```
let rec R_ob = function
  | [] -> ∅
  | x::xs -> {x}×R_mem(xs) ∪ R_ob(xs)
```

Figure 4.7.: Functions that compute R_{mem} and R_{ob} relations for a list. Syntax is stylized to aid comprehension.

in all three executions discussed above, the relational merge specification (Eqs. 4.1 and 4.2) consistently guides us towards a meaningful result, imparting a well-defined distributed semantics to the queue data type in the process.

To operationalize the merge specification discussed above, i.e., to derive a merge function that *implements* the specification, we require functions (α and γ resp.) to map a queue to the relational domain and back. The abstraction function α is simply a pair-wise composition of functions that compute R_{mem} and R_{ob} relations for a given list. The eponymous functions are shown in Fig. 4.7. The R_{mem} function computes the set of elements in a given list l, which is its unary membership relation. The function R_{ob} computes the set of all pairs (x, y) such that x occurs before y in l. The concretization function γ reconstructs a list/queue given its R_{mem} and R_{ob} relations. One way this can be done is by constructing a directed graph G whose vertices are $R_{mem}(v)$, and edges are $R_{ob}(v)$. A topological ordering of vertices in G, where ties are broken as per a consistent *arbitration* order (e.g., lexicographic order) yields the merged list/queue. We have generalized the aforementioned graph-based approach for concretizing ordering relations, and have abstracted it away as a library function γ_{ord} that concretizes (any subset of) an ordering relation of a data structure as a graph isomorphic to that structure, given *ord* an arbitration order to break ties; we provide details in Sec. 4.4. For an integer list v for example, $\gamma_{<}(R_{ob}(v))$, where $(<)$ is the less-than relation on integers, is a linear graph (i.e., a linked list), which can be straightforwardly translated to a list. The γ_{ord} function thus (mostly) automates the task of concretizing orders, which is usually the non-trivial part of writing γ. Given both α and γ, the merge function for queues (lists, in general) follows straightforwardly from the merge specification as shown in Fig. 4.8. For

```
let merge l v1 v2 =
  let (rmem_l, robs_l) = α(l) in
  let (rmem_v1, robs_v1) = α(v1) in
  let (rmem_v2, robs_v2) = α(v2) in
  let rmem_v = rmem_l ◇ rmem_v1 ◇ rmem_v2 in
  let robs_v = (robs_l ◇ robs_v1 ◇ robs_v2) ∩ (rmem_v × rmem_v) in
  γ(rmem_v, robs_v)
```

Figure 4.8.: A merge function for queues derived via the relational approach to merge

brevity, we write $A \diamond B \diamond C$ to denote the three-way merge of sets A, B, and C, which is defined thus:

$$A \diamond B \diamond C = (A \cap B \cap C) \ \cup \ (B - A) \ \cup \ (C - A)$$

4.2 Abstracting Data Structures as Relations

The various data structures defined by a program differ in terms of the patterns of data access they choose to support, e.g., value lookups in case of a tree and insertions in case of an unordered list. Nonetheless, regardless of its access pattern priorities, a data structure can be uniquely characterized by the contents it holds, and the structural relationships defined among them. This observation lets us capture salient aspects of an arbitrary data structure using concrete artifacts, such as sets and relations.

The relational encoding of the list data type has already been demonstrated in Sec. 4.1. As shown, membership and order properties of a list l, represented by relations $R_{mem}(l)$ and $R_{ob}(l)$, *characterize* l in the sense the one can reconstruct the list l given these two relations[5]. We call such relations the *characteristic relations* of a data type, a notion we shall formalize shortly. Note that characteristic relations

[5]One might think R_{ob} itself is sufficient, but that is not true. R_{ob} is empty for both singleton and empty lists, making it impossible to distinguish between them.

need not be unique. For instance, we could equivalently have defined an *occurs-after* (R_{oa}) relation - a dual of the occurs-before relation, that relates the list elements in reverse order, and use it in place of R_{ob} as a characteristic relation for lists without any loss of generality.

Relational abstractions can be computed for other data types too, but before describing a general procedure for doing so, we first make explicit certain heretofore implicit conventions we have been using in the presentation thus far. First, we often use a relation name (e.g., R_{mem}) interchangeably to refer to the relation as well as the function that computes that relation. To be precise, $R_{mem}(l)$ is the membership relation for a list l, whereas R_{mem} is a function that computes such a relation for any list l. But we prefer to call them both relations, with the latter being thought of as a relation parameterized on lists. Second, we use relations and sets to characterize data structures in this presentation, when the proper abstraction is multi-sets, i.e., sets where each element carries a unique cardinal number. While using sets leads to a simpler formulation and typically does not result in any loss of generality, we explicitly use multi-sets when they are indeed required.

As another example of a relational specification, consider the characteristic relations that specify a binary tree whose OCaml type signature is given below:

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

An R_{mem} function can be defined for trees similar to lists that computes the set of elements in a tree. A tree may denote a binary heap, in which case an *ancestor* relation is enough to capture its structure (since relative order between siblings does not matter). The definition is shown below:

```
let rec R_ans = function
  | E -> ∅
  | N(l,x,r) -> R_ans(l)  ∪  {x}×R_mem(l)  ∪  {x}×R_mem(r)  ∪  R_ans(r)
```

The full structure of the tree, including the relative order between siblings, can be captured via as a ternary *tree-order* relation (R_{to}) that extends the ancestor relation

Table 4.1.: Characteristic relations for various data types

Data Type	Characteristic Relations
Binary Heap	Membership (R_{mem}), Ancestor ($R_{ans} \subseteq R_{mem} \times R_{mem}$)
Priority Queue	Membership (R_{mem})
Set	Membership (R_{mem})
Graph	Vertex (R_V), Edge (R_E)
Functional Map	Key-Value (R_{kv})
List	Membership (R_{mem}), Order (R_{ob})
Binary Tree	Membership (R_{mem}), Tree-order ($R_{to} \subseteq R_{mem} \times$ `label` $\times R_{mem}$)
Binary Search Tree	Membership (R_{mem})

with labels denoting whether an element is to the left of its ancestor or to its right. The definition of R_{to} is shown below:

```
type label = L | R
let rec R_to = function
  | E -> ∅
  | N(l,x,r) ->
    R_to(l)  ∪  {x}×{L}×R_mem(l)  ∪  {x}×{R}×R_mem(r)  ∪  R_to(r)
```

However, the shape of a data structure may not always be relevant. For instance, given two binary search trees with the same set of elements, it does not matter whether they have the same shape. Their extensional behavior is presumably indistinguishable since they would give the same answers to the same queries. In such cases, a membership relation is enough to completely characterize a tree. Indeed, different data types have different definitions of extensional equality, so we take that into account in formalizing the notion of characteristic relations:

Definition 4.2.1 *A sequence of relations $\overline{R}_T$ is called the characteristic relations of a data type T, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ implies $x =_T y$, where $=_T$ denotes the extensional equality relation as interpreted by T.*

Our formalization requires the type of each characteristic relation to be specified in order to derive a merge function for that relation. This type is not necessarily the same as its OCaml type for we let additional constraints be specified to precisely characterize the relation. The syntax of relation types and other technicalities are discussed in Sec. 4.3.

The approach of characterizing data structures in terms of relations is applicable to many interesting data types as shown in Table 4.1. The vertex and edge relations of a graph are essentially its vertex and edge sets respectively. The key-value relation of a functional map is a semantic relation that relates each key to a value. Concretely, it is just a set of key-value pairs.

Basic data types, such as natural numbers and integers, can also be given a relational interpretation in terms of multi-sets, although such an interpretation is not particularly enlightening. For example, a natural number n can be represented as a multi-set $\{1 : n\}$, meaning that it is equal to a set containing n ones. Zero is the empty set $\{\}$. Addition corresponds to multi-set union, subtraction to multi-set difference, and a minimum operation to multi-set intersection.

4.3 Deriving Relational Merge Specifications

In Sec. 4.1, we presented a merge specification for queues expressed in terms of the membership (R_{mem}) and order (R_{ob}) relations of the list data type. The specification realizes the abstract idea of merging concurrent versions by computing, composing and applying *differences* to the LCA. Similar specifications can be derived for other inductive data types, such as trees, graphs, etc. in terms of their characteristic relations listed in Table 4.1. Beyond these data types, however, the approach suggested thus far is presumably hard to generalize as it ignores an important aspect of data type

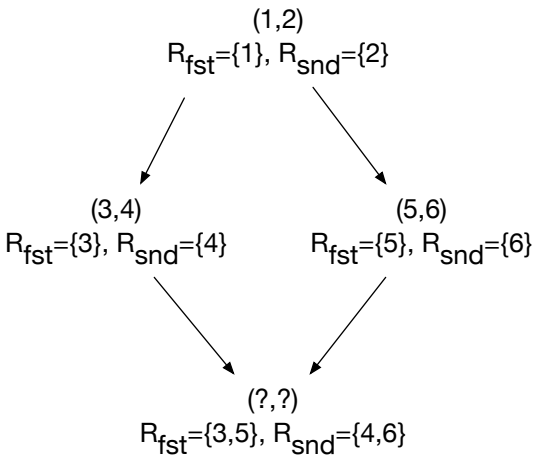


Figure 4.9.: Incorrect merge of integer pairs

construction, namely composition. In this section, we first demonstrate the challenges posed by data structure composition, and subsequently generalize our approach to include such compositions. We also formalize our approach as a set of (algorithmic) rules to *derive* merge specifications for arbitrary data structures and their compositions, given their characteristic relations, and abstraction/concretization functions.

4.3.1 Compositionality

Consider an integer pair type - `int*int` . One might define relations R_{fst} and R_{snd} on `int*int` as follows: R_{fst} and R_{snd} comprise the characteristic relations of integer

```
let R_fst = fun (x,_) -> {x}    let R_snd = fun (_,y) -> {y}
```

pairs since if the relations are equal for two integer pairs, then the pairs themselves must be equal. Using these relations, one might try to specify the merge semantics of the pair type by emulating the membership (R_{mem}) specification from the queue example of Sec. 4.1. Let v_1 and v_2, each an integer pair, denote the merging versions, and let l be their LCA version. Let v be the result of their three-way merge, i.e.,

v = `merge` l v_1 v_2. Substituting R_{mem} with R_{fst} (resp. R_{snd}) in queue's merge specification leads to the following:

$$\begin{aligned} R_{fst}(v) &= R_{fst}(l) \cap R_{fst}(v_1) \cap R_{fst}(v_2) \\ &\quad \cup \quad R_{fst}(v_1) - R_{fst}(l) \quad \cup \quad R_{fst}(v_2) - R_{fst}(l) \\ R_{snd}(v) &= \ldots (\texttt{respectively for } R_{snd}) \end{aligned}$$

Unfortunately, the specification is meaningless in the context of a pair. Fig. 4.9 illustrates why. Here, two concurrent `int*int` versions, (3,4) and (5,6), evolve from an initial version (1,2). Their respective R_{fst} and R_{snd} relations are as shown in the figure. Applying the above specification for the `int*int` merge function, we deduce that the R_{fst} and R_{snd} relations for the merged version should be the sets $\{3, 5\}$ and $\{4, 6\}$, respectively. However, the sets do not correspond to any integer pair, since R_{fst} and R_{snd} for any such pair is expected to be a singleton set. Hence the specification is incorrect.

Clearly, the approach we took for queue does not generalize to a pair. The problem lies in how we view these two data structures from the perspective of merging. While the merge specification we wrote for queue treats it as a collection of unmergeable atoms, such an interpretation is not sensible for pairs, as the example in Fig. 4.9 demonstrates. Unlike a queue, a pair defines a fixed-size container that assigns an ordinal number ("first", "second" etc) to each of its elements. Two versions of a pair are mergeable only if their elements with corresponding ordinals are mergeable. In Fig. 4.9, if we assume the integers are in fact (mergeable) counters (i.e., `Counter.t` objects), we can use `Counter.merge` to merge the first and second components of the merging pairs independently, composing them into a merged pair as described below:

```
let merge l v1 v2 = (Counter.merge (fst l) (fst v1) (fst v2),
                     Counter.merge (snd l) (snd v1) (snd v2))
```

Recall that the `Counter.merge` is the following function:

```
let merge l v1 v2 = l + (v1 - l) + (v2 - l)
```

Thus the result of merging the pair of counters and their LCA from Fig. 4.9 is:

```
(Counter.merge 1 3 5, Counter.merge 2 4 6) = (7,8)
```

The pair example demonstrates the need and opportunity to make merges compositional. The specification of such a composite merge function is invariably compositional in terms of the merge specifications of the types involved. Let $\phi_c(l, v_1, v_2, v)$ denote the counter merge specification defined, for instance, thus:

$$\phi_c(l, v_1, v_2, v) \Leftrightarrow v = l + (v_1 - l) + (v_2 - l)$$

We can now define a merge specification ($\phi_{c\times c}$) for counter pairs in terms of ϕ_c, and the relations R_{fst} and R_{snd} as follows:

$$\begin{aligned}\phi_{c\times c}(l, v_1, v_2, v) \quad \Leftrightarrow \quad & \forall x, y, z, s.\ x \in R_{fst}(l) \wedge y \in R_{fst}(v_1) \wedge z \in R_{fst}(v_2) \\ & \wedge \phi_c(x, y, z, s) \Rightarrow s \in R_{fst}(v) \\ & \wedge \forall s.\ s \in R_{fst}(v) \Rightarrow \exists x, y, z.\ x \in R_{fst}(l) \wedge y \in R_{fst}(v_1) \\ & \wedge z \in R_{fst}(v_2) \wedge \phi_c(x, y, z, s) \\ & \wedge \ldots (\texttt{respectively for } R_{snd})\end{aligned}$$

The first conjunct on the right hand side essentially says that if (counters) x, y, and z are respectively the first components of the pairs l, v_1 and v_2, and s is the result of merging x, y and z via `Counter.merge`, then s is the first component of the merged pair v. The second conjunct states the converse. Similar propositions also apply for the second components (accessible via R_{snd}), but elided. Observe that the specification captures the merge semantics of a pair while abstracting away the merge semantics of its component types. In other words, $\phi_{a\times b}$, the merge specification of the type `a*b` is parametric on the merge specifications ϕ_a and ϕ_b of types `a` and `b` respectively. Thus, the merge specification for a pair of queues, i.e., $\phi_{q\times q}$, can be obtained by replacing ϕ_c with ϕ_q, the queue merge specification (Sec. 4.1) in the above definition. The ability to compose merge specifications in this way is key to deriving a sensible merge semantics for any composition of data structures.

A pair is an example of a composite data structure that assigns implicit ordinals to its constituents. Alternatively, a data structure may assign explicit ordinals or

$$
\begin{array}{rcll}
T, \tau & \in & \texttt{Data Types} & R \in \texttt{Relation Names} \\
\rho & \in & \texttt{Tuple Types} & \coloneqq \; T \mid R(\nu) \mid \rho \times \rho \\
s & \in & \texttt{Relation Types} & \coloneqq \; \{\nu : T\} \to \mathcal{P}(\rho)
\end{array}
$$

Figure 4.10.: Type specification syntax for (functions that compute) relations

identifiers to its members. For instance, a map abstract data type (implemented using balanced trees or hash tables) identifies its constituent values with explicit keys. In either case, the top-level merge is essentially similar to the one described for pair, and involves merging constituent values that bear corresponding ordinals or identifiers. Note that this assumes that the values are indeed mergeable. Data structures may be composed of types that are not mergeable *by design*, e.g., the keys in a map data type are not mergeable, although they serve to identify the values which are mergeable. Since the merge strategy of a data structure should work differently for its mergeable and non-mergeable constituents, we need a way to identify them as such. This can be done through the type specification of relations, as described below.

4.3.2 Type Specifications for Characteristic Relations

As mentioned in Sec. 4.2, characteristic relations of a data type need to be explicitly typed. Fig. 4.10 shows the syntax of type specifications for such relations. We use both T and τ to refer to data types, with the latter used to highlight that the type being referred to is mergeable. A relation maps a value ν of a data type T to a set of tuples each of type ρ. A tuple type is specified in terms of the set from which it is drawn. It could be the set of all values of a (different) type T, or the set defined by a (different) relation R on ν, or a cross product of such sets. Note that the cross-product operator is treated as associative in this context, hence for any three

sets A, B and C, $A \times (B \times C) = (A \times B) \times C = A \times B \times C$. The syntax allows the type of a relation R on $\nu : T$ to refer another relation R' on $\nu : T$ to constrain the domain of its tuples. Some examples of relations with type specifications are given below.

Example The characteristic relations of `int list` data type can be specified thus:

$$R_{mem} \ : \ \{\nu \ : \ \texttt{int list}\} \ \rightarrow \ \mathcal{P}\,(\texttt{int}),$$
$$R_{ob} \ : \ \{\nu \ : \ \texttt{int list}\} \ \rightarrow \ \mathcal{P}\,(R_{mem}(\nu) \times R_{mem}(\nu))$$

Example The characteristic relations of a map data type with string keys and counter values can be specified thus:

$$R_{k} \ : \ \{\nu \ : \ \texttt{(string,int) map}\} \ \rightarrow \ \mathcal{P}\,(\texttt{string}),$$
$$R_{kv} \ : \ \{\nu \ : \ \texttt{(string,int) map}\} \ \rightarrow \ \mathcal{P}\,(R_{k}(\nu) \times \texttt{counter})$$

Type constraints, as described above, ensure syntactic correctness of relations. However, not all syntactically valid relations lead to semantically meaningful merge specifications. To identify those that do, we define a well-formedness condition on type specifications of relations. Let ρ_R denote the type of tuples in a relation R defined over $\nu : T$, for some data type T (i.e., $R : \nu : T \rightarrow \mathcal{P}\,(\rho_R)$). Since tuple types can refer to other relations (see ρ in Fig. 4.10, and the R_{ob} and R_{kv} type definitions above), ρ_R could be composed of $R'(\nu)$, where R' is another relation on $\nu : T$. We consider "flattening" such ρ_R by recursively substituting every occurrence of $R'(\nu)$ with the tuple type $\rho_{R'}$ of R' in ρ_R (i.e., $[\rho'_R/R'(v)]\,\rho_R$). For instance, the flattened tuple types of R_{ob} and R_{kv} are `int` $\times$ `int` and `string` $\times$ `int`, respectively. In general, the flattened tuple type of ρ_R (denoted $\lfloor \rho_R \rfloor$) is a non-empty cross product of the form $T_1 \times T_2 \times \ldots T_n$, which we shorten as $\overline{T}$. We define the well-formedness of a relation's type specification by examining its flattened tuple type as follows.

Definition 4.3.1 *A relation $R : \{\nu : T\} \rightarrow \mathcal{P}\,(\rho)$ is said to have a well-formed type specification if and only if there exists a non-empty $\overline{T}$ and a (possibly empty) $\overline{\tau}$ such that:*

- $\lfloor \rho \rfloor = \overline{T} \times \overline{\tau}$, *and*
- *Every* $T_i \in \overline{T}$ *is* not mergeable, *whereas*
- *Every* $\tau_i \in \overline{\tau}$ *is* mergeable.

Informally, a *mergeable type* is a data type for which a merge specification can be derived, and a merge function that meets the specification exists (e.g., queues and counters). Basic data types, such as strings and floats, are considered not mergeable for the sake of this discussion. The well-formedness definition presented above effectively constrains relations to be one of the following two kinds based on the type of their tuples: (a). those containing tuples composed only of non-mergeable types (i.e., $\overline{\tau} = \emptyset$ and $\lfloor \rho \rfloor = \overline{T}$), and (b). those containing tuples composed of non-mergeable types *followed by* mergeable types (i.e., $\lfloor \rho \rfloor = \overline{T} \times \overline{\tau}$ and $\overline{\tau} \neq \emptyset$). The former are relations that capture the *contents* and the *structural relationships* among the contents in a data structure (e.g., R_{mem}, R_{ob}, and R_k), and the latter are those that capture their *semantic relationships*[6] (e.g., R_{kv} - a relation that identifies key-value relationship latent in each element of a map). Based on this categorization, we can now formalize the rules to derive merge specifications of an arbitrary data type from the well-formed type specification of its characteristic relations.

4.3.3 Derivation Rules

Fig. 4.11 shows the derivation rules for merge specifications. The rules define the judgment

$$\phi_T(l, v_1, v_2, v) \supseteq \varphi$$

where ϕ_T is the merge specification for a type T parameterized on the merging versions (v_1 and v_2), their LCA (l), and the merge result (v), and φ is a first-order logic (FOL) formula. The interpretation is that the merge specification ϕ_T should subsume the

[6]This categorization corresponds exactly to the properties of interest that were said to uniformly characterize all data structures (Sec. 4.2).

FOL formula φ. The rules let us derive such constraints for every R on type T with a well-formed type specification $R : T \to \mathcal{P}(\rho)$. Accumulating the constraints derived over several such applications of the rules (until fixpoint) results in the full merge specification of type T. The rules invoke the definitions of flattening, well-formedness, etc. that we introduced above.

Recall that the tuple type of a relation is a cross product involving data types and other relations. We use its set interpretation in set operations such as intersection. For instance, if the characteristic relation on `int list` has the type ν : `int list` $\to \mathcal{P}($ `int` $\times R_{mem}(\nu))$, then its tuple type $\rho =$ `int` $\times R_{mem}(\nu)$ has a natural set interpretation as the cross product of the set of all integers and $R_{mem}(\nu)$, and hence can be used in set expressions such as $R_{ob}(\nu) \cap \rho$, as the rules in Fig. 4.11 do. The notation $A \diamond B \diamond C$ denotes three-way merge of sets A, B, and C, defined formally in Sec. 4.1. We define an *extension* operation on relations that relate ordinals or identifiers of non-mergeable type(s) $\overline{T}$ with values of mergeable type(s) $\overline{\tau}$. Let R be such a relation on type T, and let 0_i denote the "zero" or "empty" value of type τ_i. We call 0 an empty value of a type if $\overline{R}(0) = \emptyset$ for all characteristic relations $\overline{R}$ on that type (e.g., an empty list for type `list`). An extension of R is a relation R_+ that relates ordinals or identifiers not already related by R to empty or zero values. Formally, we define R_+ by defining its containment relation as follows:

$$\forall(\overline{k} : \overline{T}).\forall(\overline{x} : \overline{\tau}).\ \ (\overline{k}, \overline{x}) \in R_+ \Leftrightarrow (\overline{k}, \overline{x}) \in R \vee (\nexists(\overline{y} : \overline{\tau}).\ (\overline{k}, \overline{y}) \in R \wedge \textstyle\bigwedge_i x_i = 0_i)$$

A tuple $(\overline{k}, \overline{x})$ is in R_+ if and only if it is already in R, or R does not relate $\overline{k}$ to anything, and each x_i is an empty value. We also define a *projection* of R, denoted R_k, that is simply the set of ordinals or identifiers in R. The definition is as follows:

$$\forall(\overline{k} : \overline{T}).\ \ \overline{k} \in R_k \Leftrightarrow \exists(\overline{x} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R$$

Note that R_+ and R_k are merely notations to simplify the rules in Fig. 4.11, as will be evident shortly.

The rule Set-Merge derives merge constraints for a relation R that is composed of only non-mergeable types ($\overline{T}$), and do not draw on other relations, i.e., its tuple type ρ is not a cross product of other relations. Thus, R capture the elements of T

$\boxed{\phi_T(l, v_1, v_2, v) \supseteq \varphi}$

SET-MERGE

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\overline{T}\right)}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)}$$

ORDER-MERGE-1

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\rho\right) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)}$$

ORDER-MERGE-2

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\rho\right) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho}$$

REL-MERGE-1

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\rho\right) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{c}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{x}, \overline{y}, \overline{z}, \overline{s} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l) \wedge (\overline{k}, \overline{y}) \in R_+(v_1) \wedge (\overline{k}, \overline{z}) \in R_+(v_2) \\ \wedge \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2)) \wedge \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i) \wedge (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v)\end{array}}$$

REL-MERGE-2

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\rho\right) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{c}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{s} : \overline{\tau}).\ (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho \\ \wedge \exists(\overline{x}, \overline{y}, \overline{z} : \overline{\tau}).\ \ (\overline{k}, \overline{x}) \in R_+(l) \wedge (\overline{k}, \overline{y}) \in R_+(v_1) \wedge (\overline{k}, \overline{z}) \in R_+(v_2) \\ \wedge \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2)) \wedge \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\end{array}}$$

Figure 4.11.: Rules to derive a merge specification for a data type T

rather than their relative order. Examples include R_{mem} (list) and R_k (map). The consequent of SET-MERGE enforces the set merge semantics on R, and is an exact

specification of the merge result, leaving no room for the merge function to conjure new elements of its own. As an example, one can apply the SET-MERGE rule to the `int list` type to obtain a constraint on R_{mem} as described in Sec. 4.1.

The rule ORDER-MERGE-1 constrains a relation R whose tuple type ρ involves cross-product of other relations. Thus the relation R can be construed as an ordering relation over tuples captured by other relations over the same data structure. Examples include R_{ob} (binary relation on lists) and R_{to} (ternary relation on trees). The conclusion of ORDER-MERGE-1 adds a constraint to ϕ_T that merely enforces the set merge semantics over the ordering relation R, while retaining only those tuples that belong to the set ρ. The constraint is only an implication (and not a bi-implication), thereby underspecifying the merge result, and letting the merge function add new orders on existing elements. However, in order to prevent the merge from creating elements out of thin air, we need a constraint in reverse direction, albeit a weaker one. The rule ORDER-MERGE-2 fulfills this need, by restricting the tuples in the merged order relation to be drawn from the cross product of existing relations (ρ). Observe that these two rules together give us the constraints on R_{ob} that we wrote for the queue data structure in Sec. 4.1.

The rules REL-MERGE-1 and REL-MERGE-2 are concerned with the last category of relations that relate a data structure composed of multiple types to the (mergeable) values of those types through (non-mergeable) ordinals or identifiers. The premise of both rules assert this expectation on R by constraining its tuple type ρ to be of the form $\overline{T} \times \overline{\tau}$, where τ stands for a mergeable type. An example of such an R is the R_{kv} relation over a map ν that relates its keys to mergeable values. The REL-MERGE-1 requires a tuple $(\overline{k}, \overline{s})$ to be present in the merged relation if $\overline{k}$ is related to $\overline{x}$, $\overline{y}$, and $\overline{z}$ of type $\overline{\tau}$ respectively by the (extended) relations $R(l)$, $R(v_1)$, and $R(v_2)$, and each s_i is the result of merging x_i, y_i, and z_i as per the merge semantics of τ_i (captured by ϕ_{τ_i}). The rule thus composes the merge specification ϕ_T of T using the merge specifications $\phi_{\overline{\tau}}$ of its constituent mergeable types $\overline{\tau}$. Using the extended relation R_+ instead of R for l, v_1, and v_2 lets us cover the case where $\overline{k}$ is related to something

in one (resp. two) of the three versions, but is left unrelated in the remaining two (resp. one) versions. The extended relation R_+ lets us assume a zero value for $\overline{x}$, $\overline{y}$, or $\overline{z}$, whichever is appropriate, in such cases. We also ensure that $\overline{k}$ *needs* to be related to something in the merged version by separately merging the sets of ordinals in each merging relation as captured by the constraint $\overline{k} \in R_k(l) \diamond R_k(v_1) \diamond R_k(v_2)$. The rule Rel-Merge-2 asserts the converse of the constraint added in Rel-Merge-1, effectively making the merge specification an exact specification like in Set-Merge. Thus, for instance, a merge function of a map cannot introduce new key-value pairs that cannot be derived from the existing pairs by merging their values.

Example The merge specification presented earlier for a pair of counters can now be formally derived, albeit with a few minor changes: we use the R_{pair} relation instead of R_{fst} and R_{snd}, which assigns an explicit (integer) ordinal to each pair component:

$$\texttt{let } R_{pair} \texttt{ (x,y) = \{(1,x), (2,y)\}}$$

The type specification is $R_{pair} : \{\nu : \texttt{counter} * \texttt{counter}\} \to \mathcal{P}(\texttt{int} \times \texttt{counter})$. The tuple type is of the form $T \times \tau$, where T is not mergeable and τ is mergeable (an ordinal type can be defined separately from integers to be non-mergeable). Applying Rel-Merge-1 and Rel-Merge-2 rules yields the following merge specification for counter pairs (simplified for presentation):

$$\begin{aligned}
\phi_{c\times c} \quad = \quad & \forall(k : \texttt{int}).\forall(x,y,z,s : \texttt{counter}).\ (k,x) \in R_{pair}(l) \wedge (k,y) \in R_{pair}(v_1) \\
& \wedge (k,z) \in R_{pair}(v_2) \wedge \phi_c(x,y,z,s) \Rightarrow (k,s) \in R(v) \\
\wedge \quad & \forall(k : \texttt{int}).\forall(s : \texttt{counter}).\ (k,s) \in R_{pair}(v) \Rightarrow \exists(x,y,z : \texttt{counter}).\ (k,x) \in R_{pair}(l) \\
& \wedge (k,y) \in R_{pair}(v_1) \wedge (k,z) \in R_{pair}(v_2) \wedge \phi_c(x,y,z,s)
\end{aligned}$$

To check that the above is indeed a correct merge specification for counter pairs, one can observe that a function that directly implements this specification would correctly merge the example in Fig. 4.9.

4.4 Deriving Merge Functions

We have thus far focused on deriving a merge specification for a data type, given the type specification of its characteristic relations. We now describe how to synthesize a function that operationalizes the specification, given these relation definitions. The synthesis problem is formalized thus:

Definition 4.4.1 (Merge Synthesis Problem) *Given a data type T, a function α that computes the characteristic relations for values of T, a function γ that maps the characteristic relations back to the values of T, and a (derived) merge specification ϕ_T of T expressed in terms of its characteristic relations, synthesize a function F such that for all l, v_1, and v_2 of type T, $\phi_T(l, v_1, v_2, F(l, v_1, v_2))$ holds.*

The synthesis process is quite straightforward as the expressive merge specification ϕ_T already describes what the result of a relational merge should be. For each FOL constraint φ in ϕ_T that specifies the necessary tuples in the merged relation (i.e., of the form $\ldots \Rightarrow \overline{x} \in R(v)$ or $\ldots \Leftrightarrow \overline{x} \in R(v)$ in Fig. 4.11), we describe its operational interpretation $[\![\varphi]\!]$ that *computes* the merged relation in a way that satisfies the constraint. We start with the simplest such φ, which is the constraint added to ϕ_T by Set-Merge. Recall that α is a pair-wise composition of characteristic relations of type T (i.e., $\alpha = \lambda x.\, \overline{R}(x)$). Let R be a characteristic relation, which we obtain by projecting from α, and let `r_l`, `r_v1`, and `r_v2` be variables denoting the sets $R(l)$, $R(v_1)$, and $R(v_2)$, resp. Using these definitions, we translate the Set-Merge constraint almost identically as shown below:

$$[\![\forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)]\!] \quad = \quad \texttt{r\_l} \diamond \texttt{r\_v1} \diamond \texttt{r\_v2}$$

Order-Merge-1 can be similarly operationalized. One aspect that needs attention is the intersection with the set ρ denoting the tuple space of R. Since ρ could be composed of an infinite set like `int`, intersection with ρ cannot be naïvely interpreted. Instead, we synthesize a Boolean function $\mathbb{B}_\rho$ that returns `true` for elements present in the set ρ, and implement the intersection in terms of a `Set.filter` operation that filters a set to contain only those elements that satisfy this predicate:

```
let ks_r_l = Set.map fst r_l in
let ks_r_v1 = Set.map fst r_v1 in
let ks_r_v2 = Set.map fst r_v2 in
let ks = ks_r_l ◇ ks_r_v1 ◇ ks_r_v2 in
let r_l' = r_l ∪ (ks - ks_r_l) × {M.empty} in
let r_v1' = r_v1 ∪ (ks - ks_r_v1) × {M.empty} in
let r_v2' = r_v2 ∪ (ks - ks_r_v2) × {M.empty} in
Set.map (fun (k,x) ->
            let (x,y,z) = (r_l(k), r_v1(k), r_v2(k)) in
            let s = M.merge x y z in
            (k,s)) ks
```

Figure 4.12.: Operational interpretation of the constraint imposed by REL-MERGE-1 rule from Fig. 4.11

$$[\![\forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)]\!] =$$

```
let x = r_l ◇ r_v1 ◇ r_v2 in
Set.filter 𝔹_ρ x
```

REL-MERGE-1 covers the interesting case of compositional merges. In this case, the tuples in R have a sequence of ordinals or identifiers ($\overline{k} : \overline{T}$, which we call *keys*) followed by values of mergeable types ($\overline{\tau}$). Each τ_i is required to have a zero value 0_i for which each characteristic relation has to evaluate to $\emptyset$. In practice, this is enforced by requiring the module `M` that defines τ_i (i.e., `M.t` $= \tau_i$) to have a value `empty:t`, and checking if $R($`empty`$)$ evaluates to $\emptyset$ for each R. Since τ_i is a mergeable type, its implementation `M` should contain a `merge` function for τ_i. The R_+ definition used by REL-MERGE-1 effectively *homogenizes* the keys of $R(l)$, $R(v_1)$, and $R(v_2)$, mapping new keys to `empty`. The values with the corresponding keys are then merged using `M.merge` to compute the key-value pairs in the merged relation. Fig. 4.12 shows the operational interpretation. For brevity, we assume R to be a binary relation relating a single key to a value. `Set.map` is the usual map function with type: `'a set` $\rightarrow$ (`'a` $\rightarrow$ `'b`) $\rightarrow$ `'b set`.

The operational interpretation of derivation rules from Fig. 4.11 let us merge characteristic relations. Applying the concretization function γ on merged relations maps the relations back to the concrete domain, thus yielding the final merged value. Letting ⧫ denote relational merges as described above, the whole process can be now succinctly described:

```
let merge l v1 v2 = γ(α(l) ⧫ α(v1) ⧫ α(v2))
```

4.4.1 Concretizing Orders

The concretization function γ_{ord} aids in the process of concretizing orders, such as R_{ob}, into data structures. An inherent assumption behind γ_{ord} is that there is a single ordering relation that guides concretization. This is indeed true for the data structures listed in Table. 4.1. The ordering relation is required to be ternary, and is naturally interpreted as a directed graph G where each tuple (u, a, v) denotes an edge from u to v with a label a. Binary orders, such as R_{ob}, are a special case where the labels are all same[7] Concretization works in the context G. The first step is transitive reduction, where an edge (u, v) is removed if there exists edges (u, v') and (v', v) for some v'. A transitively reduced graph is said to be *conflict-free* if for every vertex u, there do not exist two edges with the same label a. We assume that α always generates orders that are conflict-free after transitive reduction (like R_{ob} and R_{to}). If there indeed are two edges of form (u, a, v) and (u, a, v'), they are said to be in *conflict*. Conflicts that may arise due to a merge are resolved by *inducing* an order between v and v' using a provided arbitration relation *ord*, which adds either a (v, b, v') or (v', b, v) edge for some label b. Transitive reduction at this point removes one of the conflicting edges, thus resolving the conflict. This process is repeated until all conflicts are resolved, at which point the graph is isomorphic to the merged data structure, and the latter can be reconstructed by simply traversing the former. The process is illustrated for the R_{to} relation shown in Fig. 4.13. On the left hand side of the figure is the graph G

[7]We shorten (u, a, v) in the presentation to (u, v) when appropriate.

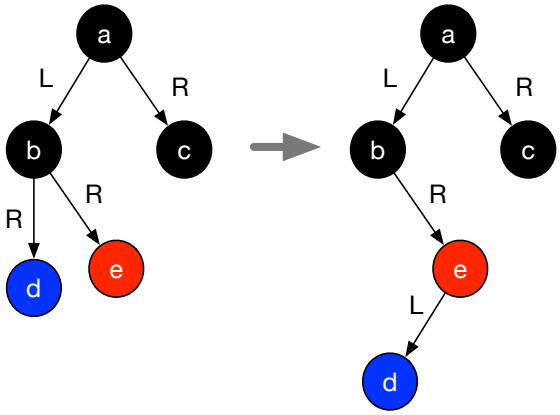


Figure 4.13.: Resolving conflicts while concretizing R_{to}

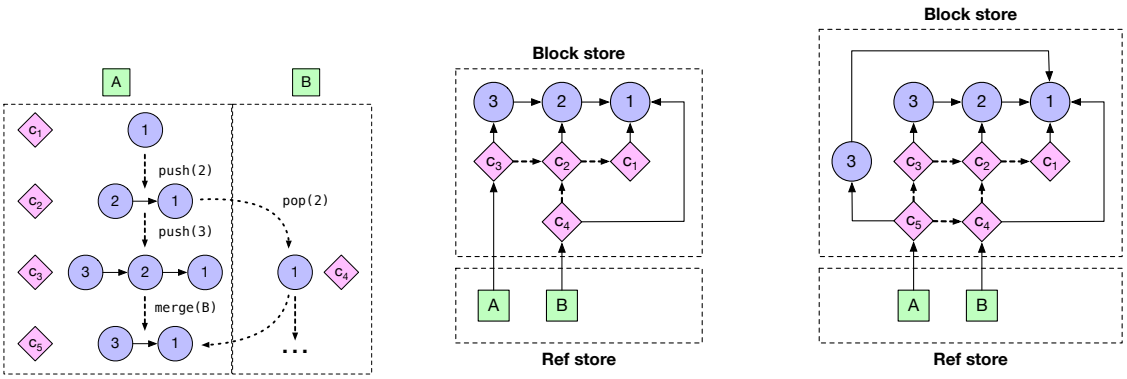
of the R_{to} relation that is obtained by merging the R_{to} relations of two trees. Both trees add d and e (resp.) as a right child to b, which results in tuples (b, R, d) and (b, R, e) in R_{to}. The tuples translate into conflicting edges shown (colored) in G. To resolve conflicts and generate an R_{to} relation consistent with the tree structure, we can invoke γ_{ord} with (for instance) the following definition of `ord`:

```
let ord x y = if x<y then (y,L,x) else (x,L,y)
```

Assuming $d < e$, `ord` adds an edge (e, L, d), which lets (b, R, d) to be removed during transitive reduction, resulting in the graph shown on the right, which is clearly a tree. We have implemented concretization functions using this interpretation for all the data structures shown in Table 4.1.

4.5 Implementation

Quark is an implementation of MRDTs realized in OCaml and built on top of a distributed storage abstraction. Its key innovation is the use of a storage layer that exposes a Git-like API, supporting common Git operations such as cloning a remote repository, forking off branches and merging branches using a three-way merge function. Quark builds on top of these features to achieve a fault-tolerant, highly-available geo-replicated data storage system. For example, creating a new replica is



(a) An execution trace for stack MRDT.

(b) Quark store before merging the commit c4 into c3.

(c) Quark store after merging c4 into c3.

Figure 4.14.: The behavior of Quark content-addressable storage layer for a stack MRDT. A and B are two versions of the stack MRDT. Diamonds represent the commits and circles represent data objects.

realized by cloning a repository, and remote pushes and pulls are used to achieve inter-replica communication. Quark also supports a variety of storage backends including in-memory, file systems and fast key-value storage databases, and distributed data stores.

4.5.1 Quark store

The main challenge in realizing MRDTs as a practical programming model is the need to efficiently store, compute and retrieve the LCA given two concurrent versions. Quark uses a content-addressable block store for storing the data objects corresponding to concurrent versions of the MRDT as well as the history of each of the versions. Given that any data structure is likely to share most of the contents with concurrent and historical versions, content-addressability maximizes sharing between the different versions.

Consider the example presented in Fig. 4.14a which shows an execution trace on a stack MRDT. There are two versions A and B. Version B is forked off from A

and is merged on to A. Since B pops the element 2, it is no longer present in the merged version. B is of course free to further evolve concurrently with respect to A. The diamonds represent the *commits* that correspond to each historical version of the stack and circles represent data objects.

Fig. 4.14b and Fig. 4.14c represent the layout of the Quark store before and after the merge. Quark uses a content-addressable append-only *block* store for data and commit information. Objects in the block store are addressed by the content of their hashes. Correspondingly, links between the objects are hashes of the contents of the objects. The reference to the two versions A and B are stored in a mutable *ref* store. The versions point to a particular commit. The commits in turn may point to parent commits (represented by dashed lines between the diamonds), and additionally may point to a single data object. Data objects stored in the block store may only point to other data objects.

Observe that in Fig. 4.14b, there is only one copy of the stack which is shared among both the concurrent and historical versions. Notice also that the branching structure of the history is apparent in the commit graph. In this example, we are merging the commits c_3 and c_4. Quark traverses the commit graph to identify the lowest common ancestor c_2 and fetches the version of the stack that corresponds to the commit. After the merge, a new commit object c_5 is added along with a new data object for 3 which points to the existing data object 1 in the block store. The version ref for A in the ref store is updated to point to the new commit c_5. As our experimental results indicate, the use of a content-addressable store makes it efficient to implement MRDTs in practice.

4.6 Evaluation

We have evaluated our approach implemented in Quark on a collection of data structure and applications.

Table 4.2.: A description of data structure benchmarks used in Quark evaluation.

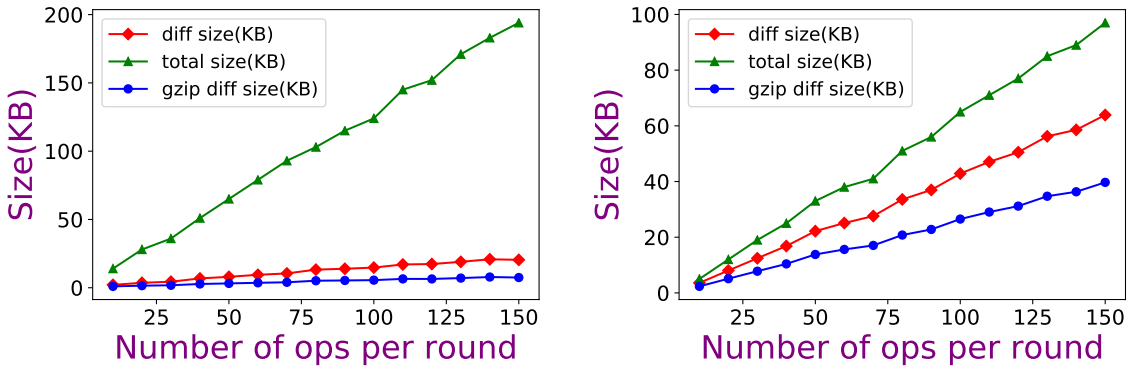
Data Structure	Description
Set	From OCaml stdlib. Implemented using AVL Trees.
Heap	Okasaki's Leftist Heap [85]
RBSet & RBMap	Okasaki's Red-Black Tree with Kahrs's deletion [86]
Graph	From the Functional Graph Library [87, 88]
List	Standard implementation of a cons list
Queue	From OCaml stdlib.
Rope	A data structure for fast string concatenation from [89]
TreeDoc	A CRDT for collaborative editing [24] but without replication awareness.
Canvas	A data structure for collaborative freehand drawing

4.6.1 Data Structure Benchmarks

The summary of data structures that we consider is given in Table. 4.2. Some of these benchmarks are taken directly from the standard library, and span over 500 lines of code defining tens of functions. Quark lets these data structures be used as MRDTs *as such* with just a few (less than 10) additional lines of code to define a relational specification and derive merges. To evaluate how these MRDTs fare under the version control-inspired asynchronous model of replication that is central to our approach, we constructed experiments that specifically answer two questions:

1. How does the size of the *diff* between versions change relative to the size of the data structure as the latter grows over time, and

2. How much is the overhead of merge relative to the computational time on the data structure.

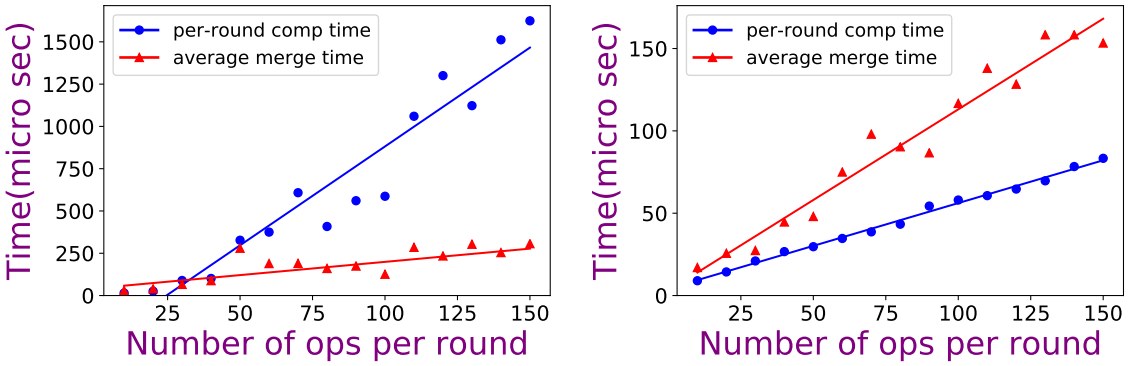


(a) Heap (b) List

Figure 4.15.: Diff vs total-size for Heap and List

As replicas periodically sync, they perform three-way merges to reconcile their versions, which requires both remote and local versions be present. Since transmitting a version in its entirety for each merge operation is redundant and inefficient, Quark computes the diff between the current version and the last version that was merged (using the content-addressable abstraction from Sec. 4.5), and transmits this diff instead. Smaller diff size (relative to the total size of the data structure) indicates that the data structure is well-suited to be a mergeable type, and the corresponding MRDT can be efficiently realized over Quark.

To measure the diff size relative to the data structure size for each data type, we conduct controlled experiments where a single client performs a series of randomly distributed operations on the data structure and *commits* a version. The exact nature of operations is different for different data types (insertion and deletion for a tree, `remove_min` for a (min) heap etc), but in general the insertion-deletion split is 75%-25%, which lets the data structure grow over time. Since a client can perform any number of operations before synchronizing, we conduct experiments by gradually increasing the number of operations between two successive commits (called a *round*) in steps of 10 from 10 to 150. For every experiment, at the end of each round, we measure the size of the data structure and the diff size between the version be-



(a) List

(b) Red-Black Tree

Figure 4.16.: Computation vs merge time for List and Red-Black Tree

ing committed and the previous version (computed by Quark's content-addressable abstraction). The experiments were conducted for all the data structures listed in Table. 4.2, and the results for the best and worst performing ones (in terms of the relative diff size are shown in Fig. 4.15. The graphs also show the size of the gzipped diff size since this is the actual data transmitted over the network by Quark.

Heap performs the best, which is not surprising considering that its tree-like structure lends itself to natural sharing of objects between successive versions. Inserting a new element into a heap, for instance, creates new objects only along the path from the root to that element, leaving the rest same as the old heap (hence shared). Other tree-like structures, including red-black and AVL trees, ropes, and document trees, also perform similarly, with their results being only slightly worse than heap. List performs the worst, again an unsurprising result considering that its linear structure is not ideal for sharing. For instance, adding (or removing) an element close to the end of a list creates a new list which only shares a small common suffix with the previous list. Nonetheless, as evident from Fig. 4.15b, its diff size on average is still less than the total size of the list, and grows sub-linearly relative to the latter. In summary, diff experiments show that version control-inspired replication model can

be efficiently supported for common data structures by transmitting succinct diffs over the network rather than entire versions.

To measure the overhead of merges relative to the computational time, we performed another set of experiments involving three replicas, each serving a client, connected in a ring layout over a (virtual) network with latency distributed uniformly between 10ms and 200ms. Each client behaves the same as with the previous (diff) experiments, except that there is a synchronization that follows the commit at the end of each round that merges the committed version with the remote version and returns the result (remote version comes from the replica upstream in the ring). We record the time spent merging the versions ("merge time"), and also the time spent performing operations in each round. As before, we gradually increase the number of operations per round, which inevitably increases the computational time and *may* increase the merge time depending on the data structure. A better performing data structure is one whose merge time increases sub-linearly, or remains constant, with the increase in computation time. A worse performing one is where merge time increases linearly or more. The results for best and worst performing data structures. in this sense, are shown in Fig. 4.16. A list performs the best here as its insertion and deletion operations are $O(n)$, making its computational time degrade faster with the increase in number of operations (kn time for computation vs n time for merge in a round of k operations). Red-Black tree (-based set) performs the worst as its $O(\log(n))$ operations are asymptotically faster than $O(n)$ merge. Nonetheless, both metrics are the same order of magnitude, which is several orders of magnitude less than the mean network latency. Moreover, since MRDTs do not require any coordination, synchronization (hence merges) can always be performed off the fast path, thus avoiding any latency overhead due to a merge.

Table 4.3.: Quark application Benchmarks

Application	SLOC	Types	Txns	DB Size (MB)	Avg. diff size (KB)
TPC-C	1081	9	3	37.9 - 47.19	19.37
TPC-E	1901	19	5	93.3 - 124.30	22.89
RUBiS	998	8	5	9.69 - 11.06	2.62
Twissandra	870	5	4	1.34 - 3.69	4.612

4.6.2 Application Benchmarks

We have also implemented four large application benchmarks by composing several mergeable data types derived from their relational specifications. Table 4.3 lists their attributes, and the summary of diff experiments we ran on them.

TPC-C and TPC-E are well-known online transaction processing (OLTP) benchmarks in the database community [56]. TPC-C emulates a warehouse application consisting of multiple *warehouses* with multiple *districts*, serving *customers* who place *orders* for *items* in *stock*. Each such application type (e.g., `customer`) is implemented as a record with multiple fields, some of which are mergeable. For instance, `c_ytd_payment` field of `customer` record is a mergeable counter recording the customer's year-to-date payment. Such records themselves are made mergeable through a relational specification similar to that of a pair type (Sec. 4.3). In TPC-C, there are a total of 9 such record types (Types column in Table 4.3). A mergeable red-black tree-based map ("RBMap") performs the role of a database table in our case. The database, which is otherwise a collection of (named) tables, is simply another mergeable record in our case that relates named fields to RBMaps corresponding to each table. The type design is shown in Fig. 4.17. TPC-C has 3 transactions that we implemented in our model as functions that map one version of the database to other, returning a result in the process. Concretely:

```
type 'a txn = db -> 'a*db
```

```
type warehouse = {w_id: id; w_ytd: counter}
type customer = {c_w_id: id; c_d_id: id; c_id: id;
                 c_name: string; c_bal: counter;
                 c_ytd_payment:counter;}
type db = {warehouse_table: (id, warehouse) rbmap;
           customer_table: (id*id*id, customer) rbmap;
           ...}
```

Figure 4.17.: Composition of mergeable data structures in TPC-C (simplified for presentation). Database (db) is composed of mergeable RBMap, which is composed of application-defined types, and ultimately, mergeable counters.

Since the database is not in-place updated, transactions are isolated by default. A transaction commit translates to the commit of a new version of type db, which is then merged with concurrent versions of db created by concurrently running transactions. We evaluated our TPC-C application composed of mergeable types by first populating the database (db) as per the TPC-C specification, and then performing the diff experiments as described above with 500 transactions. The database size grew from 37.9MB to 47.19MB during the experiment (DB Size column in Table 4.3), with the average size of diff due to each transaction being constant around 20KB (Avg. diff size column).

We have implemented three other applications, including the TPC-E and RUBiS [67] benchmarks, and a twitter-clone called Twissandra [66]. Our experience of building and experimenting with these applications has been consistent with our earlier observations that (a). complex data models of applications can be realized by composing various mergeable data types (b). the resultant application state lends itself to efficient replication under Quark's replication model with well-defined and useful semantics.

4.7 Related Work

The idea of versioning state, which is the cornerstone of the approach presented in the paper, bears resemblance to Concurrent Revisions [90, 91], a programming abstraction that provides deterministic concurrent execution, and Tardis [92], a key-value store that also supports a branch-and-merge concurrency control abstraction. However, unlike these previous efforts which provide no principled methodology for constructing merge functions, or reasoning about their correctness, the primary contribution of this chapter is in the development of a type-based compositional derivation strategy for merge operations over inductive data types. It is expected that the formalization provided in this paper significantly alleviates the burden of reasoning about state-based replication. Furthermore, the integration of a version-based mechanism within OCaml allows a degree of type safety and enables profitable use of polymorphism not available in related systems.

Burckhardt *et al.* [70] present an operational model of a replicated data store that is based on the abstract system model presented in [71]; their design is similar to the system model described in the previous chapter. In these approaches, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. In contrast, Quark fully abstracts away such details - while programmers must provide abstraction and concretization functions that map datatype semantics to the language of relations and sets, the reasoning principles involved in performing this mapping are not dependent upon any specific storage or system abstraction, such as eventual consistency [71,93]. Given a library of predefined functions for common data types, and a methodology for deriving their composition, the burden of migrating sequential data types to a replicated setting is substantially reduced.

Modern distributed systems are often equipped with only parsimonious data models (e.g., key-value model) that complicate program reasoning, and make it hard to enforce application integrity. Some authors [82] have demonstrated that it is possible

to *bolt-on* high-level consistency guarantees (e.g., causal consistency) [64,94] as a *shim layer* service over existing stores, but these approaches do not consider integration of these services within the type abstractions provided by a high-level client-facing language.

A number of verification techniques, programming abstractions, and tools have been proposed to reason about program behavior in a geo-replicated weakly consistent environment. These techniques treat replicated storage as a black box with a fixed pre-defined consistency model [8, 36, 47–50]. On the other hand, compositional proof techniques and mechanized verification frameworks have been developed to rigorously reason about various components of a distributed data store [51, 74]. Quark is differentiated from these efforts in its attempt to mask details related to distribution but unnecessary for defining meaningful (convergent) merge operations. An important by-product of this principle is that Quark does not require algorithmic restructuring to transplant a sequential or concurrent program to a distributed, replicated setting; the only additional burden imposed on the developer is the need to provide abstraction and concretization functions for compositional data types that can be used to derive well-formed merge functions. As demonstrated in this chapter, this is significantly simpler than reasoning about weakly-consistent behaviors.

Quark shares some resemblance to conflict-free replicated data types (CRDT) [65]. CRDTs define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. Unlike CRDTs, the operations on mergeable types in Quark need not commute and the reconciliation protocol is defined by merge functions derived from the semantics of the data types whose instances are intended to be replicated. The lack of composability of CRDTs is a major hindrance to their utility that forms an important point of distinction with the approach presented here. A CRDT's inability to take advantage of provenance information (i.e., LCAs) is another important drawback. As a result, constructing even simple data types like counters are more complicated using CRDTs [65] compared to their realization in Quark.

Finally, on the language design front, there have been approaches where relations feature prominently, e.g., Datalog [95] and Prolog [96]. In such languages, data is represented as "facts" described by relations, and computation on data is structured as relational queries. In contrast, Quark does not advocate a new style of programming, but rather uses relations to augment capabilities of data structures in an existing model of programming. Relations have been employed to reason about programs and data structures, for example in shape analysis [97–99], but the focus is always on using relations to prove correctness of programs, not on using them as convenient run-time representations.

5 CONCLUDING REMARKS AND FUTURE WORK

In pursuit of low latency, high availability, trust decentralization, and other such lofty goals, data-intensive applications have unraveled the elegant abstractions of data storage systems that would otherwise guarantee their safety and integrity. While the performance benefits have accrued as expected, the lack of suitable programming models and a non-existent tool support has made it hard to recover the safety guarantees for even simplest of such applications. The high complexity of these applications means that it is almost impossible for developers to have a tractable mental model to reason about their correctness, and obtain any measure of confidence in their ability to handle safety-critical systems. Addressing this problem has been the focus of this thesis. Towards this end, we presented several new formal systems, reasoning methods, automation techniques, and analysis tools that push the envelope in our understanding of complex data-intensive systems.

In Chapter 2, we showed that weakly-isolated transactions, which have hitherto been thought of as implementation hacks, can be put on the same formal footing as their serializable counterparts. Importantly, we showed that this formalism need not be based on low-level concepts such as read-write traces, as some authors have proposed, but rather can be structured around high-level artifacts exposed to applications, such as database states and SQL operations, which are easier to reason about in the context of application semantics. The simplicity of this formalism comes to fore in the succinct and easy-to-understand specifications of weak isolation levels implemented on various commercial databases, and in the lightweight compositional proof system for database transactions that is a modest extension of well-studied Rely-Guarantee reasoning framework. The simplicity of sets and relations – the logical artifacts used in the formalism, has paved way for considerable proof automation implemented in ACIDifier tool.

The work presented in Chapter 2 is only the beginning of what could be a long and fruitful research exploration. Firstly, while the isolation specifications, in conjunction with the Rely-Guarantee proof framework, allow rigorous reasoning about program correctness, the specifications themselves are derived by clever and mostly-informal reasoning about weak isolation implementations on databases (see Sec. 2.5.1 for an example). To gain additional confidence in the verification process, there is a need to formalize this aspect of reasoning. There are at least two ways this can be attempted – either using the gradual refinement technique exemplified by the IronFleet system [100], or by composing "litmus tests" a\'la weak memory models [78] that establish the fidelity of formal models of isolation to their implementations. Secondly, despite considerable proof automation, ACIDifier still requires rely and guarantee annotations to be provided by the programmer. Future research may alleviate this burden either by synthesizing annotations from sample I/O traces [101], or by using bounded verification instead of complete verification. Thirdly, the approach of abstracting away complex lock-based implementations by simple specifications, which was demonstrated for weak isolation, can be applied to concurrency control mechanisms in general. For instance, reasoning about mutual exclusion algorithms, which is currently done at a low level using, for e.g., Concurrent Separation Logic [102], could be lifted to a high-level using appropriate logical abstractions that model the state of a concurrent program as a whole rather than pointers and memory locations.

In Chapter 3, we showed that the operational semantics of a distributed program under weakly-consistent state replication, can be captured succinctly using a single evaluation rule that uses nothing more than sets and relations (Sec. 3.4.3). The rule accounts for vagaries of asynchronous distributed systems, including network partitions, message losses and reorderings, system failures etc. The simple operational semantics immediately led to a symbolic execution and bounded verification technique that is parametric over declarative specifications of weak consistency. The technique, implemented in a tool Q9, has proven to be surprisingly scalable and effective as demonstrated by the statistics from Table 3.3. One direction that future research

could take is towards formulating a Q9-inspired approach for bounded verification of relaxed-memory programs. Like weak consistency, weak memory is also specified via axiomatic semantics operating over program traces, and like weak consistency variants, weak memory variants are comparable. It is therefore conceivable to formulate an evaluation rule that relates the operational semantics of a weak memory program to its axiomatic semantics, and then use weak memory specifications to control the behavior of the program. The abstract machine thus obtained can be used as a basis for symbolic execution and bounded verification. Another potential avenue for future research is formulating an variant of partial-order reduction for Q9-style symbolic model checking, that exploits the repeating patterns in the executions of weak consistency programs. Successful partial-order reduction has the potential to push the verification bound high enough that it can be *enforced* at runtime without significant performance penalty, thus giving promoting bounded verification to full verification essentially for free.

In Chapter 4, we demonstrated that the seemingly random choices (e.g., "add-wins" or "remove-wins") made by conflict-free replicated data types can in fact be based on one underlying principle. We demonstrated how the principle can be used to *derive* replicated data types from inductively-defined sequential data types that are common in functional languages. The chapter also introduces a model of replication based on version control systems such as Git, and shows how it can support *mergeable* replicated data types (MRDTs) that are more expressive that conflict-free replicated data types. A programming framework embedded in OCaml, called Quark, that natively supports MRDTs and includes the aforementioned principled derivation logic for merge functions, has also been presented. One area of concern in Quark is the computational efficiency of derived merges, which could be a topic of future research. In particular, there is a need for an enhanced derivation/synthesis algorithm that exploits the content-addressable abstraction offered by Quark to synthesize efficient merges. Another direction for productive future research is towards generalizing the merge derivation to take into account constraints imposed

by complex application-specific invariants. This lets the benefits of Quark reach such sophisticated applications as blockchains.

REFERENCES

[1] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2017.

[2] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In G M Nijssen, editor, *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 364–394. Elsevier/North Holland Publishing, 1976.

[3] 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.

[4] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[5] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[6] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[7] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[8] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[9] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM.

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association.

[11] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] Eric Brewer. Towards Robust Distributed Systems (Invited Talk), 2000.

[13] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[14] Paolo Viotti and Marko Vukolic. Consistency in Non-Transactional Distributed Storage Systems. *CoRR*, abs/1512.00168, 2015.

[15] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[16] Swaminathan Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[17] Alan Fekete, Shirley N. Goldrei, and Jorge Pérez Asenjo. Quantifying Isolation Anomalies. *Proc. VLDB Endow.*, 2(1):467–478, August 2009.

[18] BTC Stolen from Poloniex, 2016.

[19] How I Stole Roughly 100 BTC From an Exchange and How I Could Have Stolen More!, 2016.

[20] Avoid Race Conditions that Violate Uniqueness Validation - Rails, 2016.

[21] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[22] Kamal Zellag and Bettina Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM.

[23] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[24] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, April 2017.

[26] Peter Gammie, Antony L. Hosking, and Kai Engelhardt. Relaxing safely: Verified on-the-fly garbage collection for x86-tso. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 99–109, New York, NY, USA, 2015. ACM.

[27] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification: 27th International Conference*, pages 449–465. Springer International Publishing, 2015.

[28] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[29] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Cambridge, MA, USA, 1999. AAI0800775.

[30] 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, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] Transaction Isolation, 2016. Accessed: 2016-07-1 10:00:00.

[32] Transaction Isolation Levels, 2016. Accessed: 2016-07-1 10:00:00.

[33] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[34] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone Together: Compositional Reasoning and Inference for Weak Isolation, 2018. https://arxiv.org/abs/1710.09844.

[35] Ergon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag Telos, 1996.

[36] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 371–384, New York, NY, USA, 2016. ACM.

[37] Kamal Zellag and Bettina Kemme. Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *The VLDB Journal*, 23(1):147–172, February 2014.

[38] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[39] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. Precisely Serializable Snapshot Isolation (PSSI). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 482–493, Washington, DC, USA, 2011. IEEE Computer Society.

[40] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[41] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC)*, pages 73–82, 2017.

[42] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20, New York, NY, USA, 2017. ACM.

[43] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[44] Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC, 2016.

[45] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[46] Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu. Semantic Conditions for Correctness at Different Isolation Levels. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, Washington, DC, USA, 2000. IEEE Computer Society.

[47] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[48] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[49] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[50] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System*, EuroSys '15, Bordeaux, France, 2015.

[51] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM.

[52] Vafeiadis, Viktor. RGSep Action Inference. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 345–361, 2010.

[53] Viktor Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22^{nd} International Conference on Computer Aided Verification*, CAV'10, pages 450–464, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Vafeiadis, Viktor and Parkinson, Matthew. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, September 2007.

[55] J C Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17^{th} Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Comput. Soc, 2002.

[56] 2018. TPC Benchmarks.

[57] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[58] 2018. Riak NoSQL Database.

[59] 2009. Voldemort Distributed Database.

[60] M. Shapiro, A. Bieniusa, N. Preguiça, V. Balegas, and C. Meiklejohn. Just-Right Consistency: Reconciling Availability and Safety, January 2018. ArXiv e-prints.

[61] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM.

[62] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[63] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. Modular reasoning about heap paths via effectively propositional formulas. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 385–396, New York, NY, USA, 2014. ACM.

[64] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 626–638, New York, NY, USA, 2017. ACM.

[65] Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Dfago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[66] Twitter clone on Cassandra, 2014. Accessed: 2014-11-4 13:21:00.

[67] Rice University Bidding System, 2014. Accessed: 2014-11-4 13:21:00.

[68] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[69] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[70] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, ECOOP '15, Prague, Czech Republic, 2015.

[71] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[72] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630, New York, NY, USA, 2016. ACM.

[73] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):108:1–108:31, October 2017.

[74] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

[75] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[76] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[77] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory Models. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 258–272, Berlin, Heidelberg, 2010. Springer-Verlag.

[78] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 467–481, New York, NY, USA, 2017. ACM.

[79] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.

[80] 2018.

[81] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[82] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[83] 2013. Myth: Eric Brewer on Why Banks are BASE Not ACID - Availability Is Revenue.

[84] Amazon Simple Queue Service.

[85] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[86] Stefan Kahrs. Red-black trees with types. *J. Funct. Program.*, 11(4):425–432, July 2001.

[87] A Functional Graph Library, 2008.

[88] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, September 2001.

[89] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Softw. Pract. Exper.*, 25(12):1315–1330, December 1995.

[90] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM.

[91] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[92] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1615–1628, New York, NY, USA, 2016. ACM.

[93] Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Dfago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[94] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[95] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. Declarative logic programming. chapter Datalog: Concepts, History, and Outlook, pages 3–100. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2018.

[96] Kenneth A. Bowen. Prolog. In *Proceedings of the 1979 Annual Conference*, ACM '79, pages 14–23, New York, NY, USA, 1979. ACM.

[97] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 247–260, New York, NY, USA, 2008. ACM.

[98] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2):5:1–5:52, February 2010.

[99] Gowtham Kaki and Suresh Jagannathan. A Relational Framework for Higher-order Shape Analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 311–324, New York, NY, USA, 2014. ACM.

[100] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[101] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 491–507, New York, NY, USA, 2016. ACM.

[102] Viktor Vafeiadis. Concurrent Separation Logic and Operational Semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011.

VITA

Gowtham Kaki hails from the town of Rajahmundry in Andhra Pradesh province of India. He obtained his Bachelors in Computer Science from Birla Institute of Technology and Science (BITS), Pilani, India in 2009, and subsequently worked at Yahoo Corporation in Bangalore as a Software Development Engineer, building publishing tools for Yahoo's content curators and editors. Gowtham started his graduate studies at Purdue University in August, 2011, obtaining Masters and Ph.D. degrees in Computer Science in May 2016 and August 2019, respectively. During his tenure as a graduate student, Gowtham won multiple awards and recognitions for his research in Programming Languages and Software Engineering, including a Ph.D Fellowship from Google, Maurice Halstead award from Purdue CS, and being named one among the "30 under 30" most accomplished alumni by his alma mater BITS Pilani. Gowtham's research interests include functional programming languages, distributed systems, concurrent programming, type systems, and program logics.