DATA REPRESENTATION SYNTHESIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Peter Hawkins
April 2012

Abstract

This dissertation introduces Data Representation Synthesis, a technique for specifying combinations of data structures with complex sharing in a manner that is both declarative and results in provably correct code. In our approach, abstract data types are specified using relational algebra and functional dependencies. We describe a language of decompositions that permit a programmer to specify different concrete representations for relations, and show that operations on concrete representations soundly implement their relational specification.

Decompositions also give new insight into the problem of writing safe and efficient concurrent code. We introduce lock placements, which describe, for each heap location, which lock guards the location, and under what circumstances. By incorporating lock placements into decompositions, a compiler can automatically explore the space of possible legal data representations and locking strategies, while simultaneously guaranteeing correctness, serializability of relational operations, and deadlock-freedom.

It is easy to incorporate data representations synthesized by our compiler into existing systems, leading to code that is simpler, correct by construction, and comparable in performance to the code it replaces. We describe our experience with a prototype compiler; code generated by our system can easily be dropped into existing systems in place of complex hand-written implementations with comparable performance.

Acknowledgements

I have been very fortunate to work with my advisor, Alex Aiken, whose mentorship and guidance has been instrumental in helping me develop as a researcher. Alex introduced me to many aspects of computer science, collaborated with me on a wide variety of new and exciting problems, and helped me navigate the shoals and rapids of the research world. Working in Alex's group has always been fun and has taught me a great deal: many thanks go out to Michael Bauer, Sorav Bansal, Isil Dillig, Tom Dillig, Brian Hackett, Mayur Naik, Adam Oliner, Eric Schkufza, Rahul Sharma, Sean Treichler, and Yichen Xie for all the interesting conversations and all the cups of coffee.

I am grateful for the opportunity to work with brilliant collaborators during my time at Stanford: Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Their guidance and advice has taught me many different things about research and how to approach it, and I feel honored to have had the opportunity to work with them. I would especially like to thank Kathleen, who served as my advisor during Alex's absence, and as a friend and mentor thereafter. I would also like to thank my reading committee (Alex Aiken, David Dill, and Kathleen Fisher), and my oral committee (Alex Aiken, David Dill, Kathleen Fisher, Hector Garcia-Molina, and Stephen Sano) for their invaluable assistance.

During my time at Stanford, I have made many wonderful friends with whom I have shared many adventures. To all those who shared these experiences with me, especially my friends, family, and Mariangela: thank you. I couldn't have done this without you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

By automatically synthesizing the low-level representation of a program's data from a high-level relational description, we can obtain code that is simpler, correct by construction, and comparable in performance to the handwritten code that it replaces.

Almost every non-trivial program represents its data internally using dynamically-allocated data structures, collectively called the *heap*. One of the first things a programmer must do when implementing a system is commit to a particular choice of heap data structures that represent the system's state. A choice of data representation must meet several requirements: the representation must support all of the operations required by the code, the data structures must be efficient for the workload, and the implementation must be correct.

Whatever the choice of data structures, it has a pervasive influence on the subsequent code, and as requirements evolve it is difficult and tedious to change the data structures to match. Extending or changing the existing data structures to support a new requirement may require many changes throughout the code. For a data representation to be correct, data structure invariants must be enforced by every piece of code that manipulates the heap. Invariants on multiple data structures with complex patterns of sharing are hard to state, difficult to enforce, and easy to get wrong.

To address this problem this dissertation proposes a method termed *data representation synthesis*. In our approach a data structure client describes and manipulates data at a high level as *relations*; a data structure designer then provides *decompositions* which describe how those relations should be represented in memory as a combination of primitive data structures. Our compiler RELC takes a relation and its decomposition and emits correct and efficient low-level code that implements the relational interface. As the programmer may not always know the best decomposition for a particular relation, we describe an autotuner, which automatically identifies a good decomposition for a particular relation and a particular workload.

Synthesis allows programmers to describe and manipulate data at a high level as relations, while giving control of how relations are represented physically in memory. By abstracting data from its representation, programmers no longer prematurely commit to a particular representation of data. If programmers want to change or extend their choice of data structures, they need only change the decomposition; the code that uses the relation need not change at all. Synthesized representations are correct by construction; so long as the programmer conforms to the relational specification, invariants on the synthesized data structures are automatically maintained.

1.1 Dissertation Overview

For the balance of this chapter we describe data representation synthesis informally, motivated by a running example taken from the Linux kernel, and discuss how our approach relates to the extensive literature on describing and reasoning about the heap.

Chapter 2 formally describes data representation synthesis, which allows us to specify combinations of data structures with complex sharing in a manner that is both declarative and results in provably correct code. In our approach, abstract data types are specified using relational algebra and functional dependencies. We describe a language of decompositions that permit the user to specify different concrete representations for relations, and show that operations on concrete representations soundly implement their relational specification. We describe our experience incorporating data representations synthesized by our compiler into existing systems, leading to code that is simpler, correct, and comparable in performance to the code it replaces.

Chapter 3 and Chapter 4 extend data representation synthesis to support concurrent access from multiple threads. One of the primary benefits of the decomposition language is that it provides a powerful static description of the heap, which we leverage to reason about concurrent data structures protected by locks. Chapter 3 introduces the abstract concept of a lock placement, which maps locations in the heap onto the locks that protect them. Lock placements can describe a wide variety of both conservative and speculative locking strategies. A prerequisite for defining and reasoning about a lock placement is a description of the aliasing patterns in the heap; we progressively consider lock placements for a sequence of increasingly complicated heaps, applying lock placements to a class of flat maps, to tree-like heaps, and culminating in decomposition heaps. Chapter 4 applies the concept of a lock placement in the context of synthesis, yielding a complete concurrent data representation system, and presents experimental results on a variety of benchmarks.

1.2 A Motivating Example

To motivate data representation synthesis, we begin with an example drawn from the Linux kernel. One of the kernel's functions is to maintain a cache in memory of information about filesystems and files on disk. The kernel represents each mounted filesystem in memory as a `super_block` object, and represents each file as an `inode` object. The kernel maintains a variety of data structures that link `super_block` and `inode` objects, each to support a different access pattern. Figure 1.1 shows extracts from the code that defines `super_block` and `inode`, focusing on the data structures that allow us to navigate between them.

Each `super_block` and `inode` object participates in a number of linked lists. Most linked lists in the Linux kernel are circular, doubly-linked lists, with a distinguished head node. Both the head of the list and

```
struct list_head {
    struct list_head *next, *prev;
};
struct super_block {
    struct list_head s_list;      /* Keep this first */
    struct list_head s_inodes;    /* all inodes */
    ...
};
struct inode {
    struct list_head i_list;
    struct list_head i_sb_list;
    unsigned long    i_ino;
    ...
};

struct list_head super_blocks;
struct list_head inode_in_use;
struct list_head inode_unused;
```

Figure 1.1: Extracts from the definitions of `super_block` and `inode` in Linux v2.6.29.4, drawn from the files `include/linux/list.h`, `include/linux/fs.h`, `fs/super.c`, and `fs/inode.c`.

the entries in the list are represented using `struct list_head`, which is a pair of a `next` pointer and a `prev` pointer. The `next` field of the distinguished head node points to the first element of the list, whereas the `next` field of the last element in the list points back to the head node. The head node is always present, even in an empty list; if the list is empty, then the `next` pointer of the head node points to the head node itself. The `prev` pointers are similar to the `next` pointers but in the opposite order.

Linux lists are *intrusive*, that is, the `list_head` structures that make up a list are embedded as fields of the objects that participate in the list. The `next` pointer of a `list_head` object points to the next `list_head` object, which is either a field of an object on the list, or the distinguished head node. The kernel uses pointer arithmetic to convert a pointer to a `list_head` object field into a pointer to the object that contains it[1], analogous to a downcast in C++. One main advantage of the intrusive doubly-linked list representation is that, given a pointer to an object on a list, it is possible to remove that object from the middle of the list in constant time.

Figure 1.2 depicts a typical heap state, consisting of two `super_block` objects. Three inodes belonging to the left-hand superblock are present in the cache, with inode numbers `i_ino` 213, 22, and 619, depicted in a vertical column below the superblock. Similarly two inodes belonging to the right-hand superblock are present in the cache, with inode numbers 477 and 213. Note that there are two inodes with inode number 213 present in the cache; the inode number field `i_ino` is only unique within a particular filesystem.

The kernel needs to be able to iterate over all mounted filesystems, so the kernel maintains a linked list

[1]Technically the use of pointer arithmetic to navigate between fields of a structure is a violation of the C standard; even so, this is a common pattern in systems code to work around the lack of inheritance or templates in C.

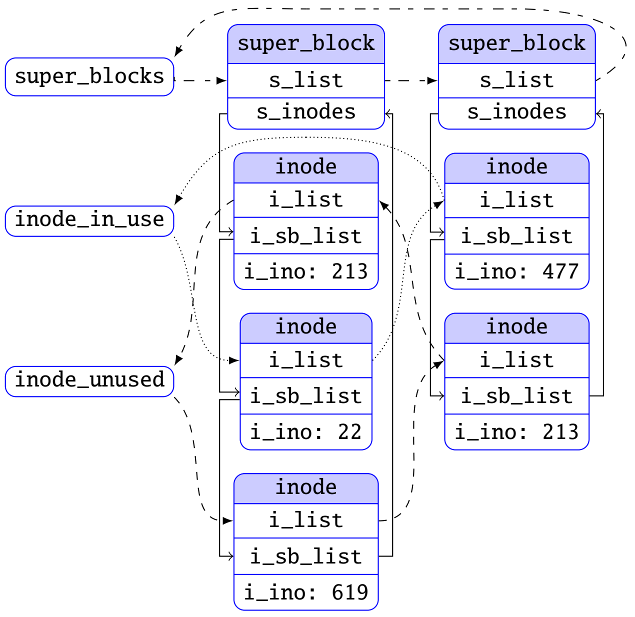


Figure 1.2: A typical Linux `inode` cache heap state. `inode` objects simultaneously participate in multiple circular lists. Different line types denote different lists. Only the `next` edges of each list are shown; we omit the `prev` edges for clarity.

of all of the `super_block` objects in memory. The global variable `super_blocks` is the distinguished head node of the list, whereas the `s_list` fields of the `super_block` objects contain the pointers that form the spine of the list.

When unmounting a filesystem the kernel needs to find all `inode` objects belonging to that filesystem and evict them from the cache. To support efficient iteration over a filesystem's `inode` objects, for each `super_block` the kernel maintains a list of `inode` objects belonging to that filesystem. The distinguished head node of the list is the `s_inodes` field of the `super_block`, whereas the spine of the list is represented using the `i_sb_list` fields of the `inode` objects.

Since there is usually far too much filesystem metadata to keep in memory, the `inode` objects in memory are only a cache for the metadata on the disk. When the cache fills up, the kernel makes space in the cache by evicting `inode` objects that have not been used recently. To support cache eviction each `inode` is also on either of two global lists of `inode` objects: `inode_in_use`, if the inode is in use, or `inode_unused`, if the inode is unused. The kernel keeps the `inode_unused` list in an approximate Least-Recently-Used order; to free space the kernel walks along the list of unused inodes evicting nodes until sufficient space is available.

1.2.1 Characteristics of a Good Data Representation

Are the Linux kernel's data structures a good representation of an `inode` cache? To answer this question, we must first characterize what we want from a data representation; ideally, a representation of data should satisfy the following criteria:

- The data representation must provide a correct implementation of all operations required by client code.
- The implementation must be efficient for the access patterns that occur in practice.
- As the requirements on the data representation change, it should be easy to evolve the code to match.
- It should be easy to reason about the correctness of clients of a data representation.

The `inode` cache by definition supports all of the operations required by the Linux kernel. The implementation is also quite efficient for the access patterns made by the kernel; in particular we can efficiently

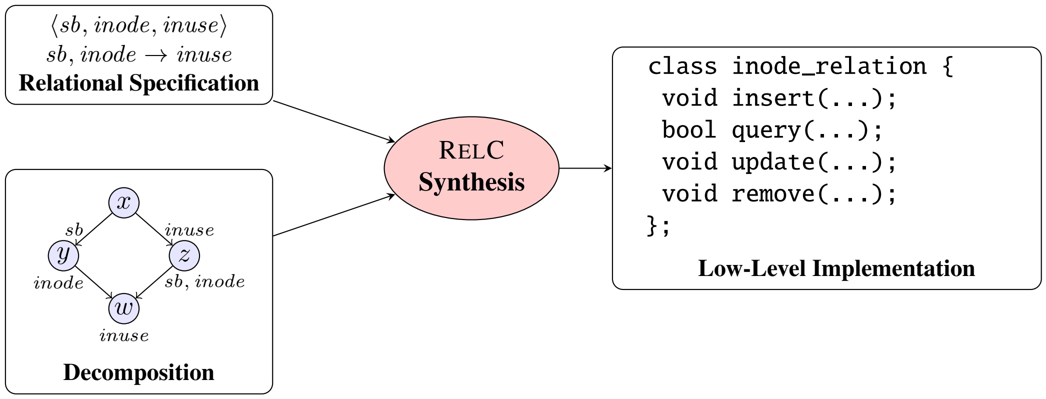
- iterate over all `super_block` objects,
- given a particular `super_block` object, iterate over all `inode` objects belonging to that filesystem,
- iterate over unused `inode` objects on all filesystems in Least-Recently-Used order,
- add and remove `super_block` and `inode` objects from the cache, and
- change the state of an `inode` from used to unused or vice versa.

However it is not a particularly easy task to reason formally or informally about either the `inode` cache or its clients. There is no real abstraction boundary between the data structures of the cache and its clients, and nowhere in the code is there a specification of precisely what the invariants of the `inode` cache data structures are. It is therefore difficult to reason in a modular way about the correctness of the cache and its clients. Because of the lack of abstraction, the choice of data structures has a pervasive influence on the rest of the code, and as requirements evolve it would be difficult and tedious to change the data structures to match.

Furthermore, invariants on multiple, overlapping data structures that represent different views of the same data are hard to state, difficult to enforce, and easy to get wrong. For example, the kernel maintains the invariants that every `inode` is present on its parent `super_block`'s list of children, and that every `inode` is on exactly one of the `inode_in_use` or `inode_unused` lists. Such invariants must be enforced by every piece of code that manipulates the `inode` cache data structures. It would be easy to forget a case, say by failing to move an `inode` to the appropriate list when changing its state, or by failing to remove an `inode` from the per-state list when evicting it from the cache. Invariants of this nature require deep knowledge about the heap's structure, and are difficult to enforce through existing static analysis or verification techniques.

1.3 Data Representation Synthesis

In this dissertation we propose a different approach, called *data representation synthesis*, depicted in Figure 1.3. In our approach a data structure client writes code that describes and manipulates data at a high level as *relations*; a data structure designer then provides *decompositions* which describe how those relations should be



```
class inode_relation {
 void insert(...);
 bool query(...);
 void update(...);
 void remove(...);
};
```

Figure 1.3: Schematic of Data Representation Synthesis.

represented in memory as a combination of primitive data structures. Our compiler RELC takes a relation and its decomposition and synthesizes correct and efficient low-level code that implements the relational interface.

Synthesis allows programmers to describe and manipulate data at a high level as relations, while giving control of how relations are represented physically in memory. By abstracting data from its representation, programmers no longer prematurely commit to a particular representation of data. If programmers want to change or extend their choice of data structures, they need only change the decomposition; the code that uses the relation need not change at all. Synthesized representations are correct by construction; so long as the programmer conforms to the relational specification, invariants on the synthesized data structures are automatically maintained.

Despite its apparent complexity, our key observation is that we can model the Linux `inode` cache as a relation with four columns $\{sb, inode, nr, inuse\}$. Each entry in the relation is a tuple of a `super_block` object (sb), an `inode` object ($inode$), the inode number (nr), and a boolean indicating whether the inode is in use ($inuse$). Not every such relation represents a valid `inode` cache; all meaningful cache relations satisfy a pair of *functional dependencies* $sb, nr \to inode$ and $inode \to sb, nr, inuse$. The functional dependencies imply that there is at most one $inode$ object with a given nr belonging to any given superblock sb; further each $inode$ object has a unique superblock sb, inode number nr, and $inuse$ state.

There are many possible representations of a relation; a data structure designer specifies a particular choice of representation using a *decomposition*, which describes how to assemble container data structures from a library into a representation of the relation. It is the task of the compiler to generate implementations of operations to query and modify the relation, specialized to the particular decomposition. Different choices of decomposition will lead to different performance trade-offs. If the data structure designer does not know the best decomposition or does not want to specify a decomposition by hand, then an autotuner can automatically explore the space of possible decompositions to find a good decomposition for a particular workload.

1.4 Databases and Programming With Relations

The idea of representing data using relations is not new, dating back to the work of Codd [1970]. Relations and relational algebra underlie the field of relational databases. Our insight is that relations are a good abstraction for the kinds of heap data structures that occur in systems code, and unlike a typical relational database we can give the programmer fine-grained control over the underlying representation via the decomposition language.

Classical relational databases represent a table as a list of tuples, together with one or more auxiliary indices that allow particular tuples to be located efficiently. Often clients have little control of how data is represented, beyond specifying the set of auxiliary indices. Since a database table must be capable of representing any possible relation, databases are limited in how much they can specialize the representations of tables by leveraging properties of the data, such as functional dependencies. Traditional databases also usually do not assume that the complete set of queries will be known in advance, which limits how much a database can specialize its representation to the query workload.

Recently there has been much interest in column-oriented databases, in which data is organized in columns, rather than in rows as is traditional. The MonetDB [Boncz and Kersten, 1999] and MonetDB/X100 [Boncz et al., 2005] pioneered the area of modern column-oriented databases. Column-oriented databases have subsequently been explored in the context of the C-Store database [Stonebraker et al., 2005; Abadi et al., 2006]. Experimental evidence suggests that such specialized databases can perform orders of magnitude better than a conventional RDBMS for particular tasks, and a variety of authors have argued that the era of the one-size-fits-all row-oriented database is coming to an end [Stonebraker et al., 2007a,b; Abadi et al., 2008]. This dissertation supports the same agenda in the context of in-memory data, demonstrating that there is significant value to customizing the representation of data to a particular workload.

Most relational databases focus primarily on on-disk representations for data, optimizing to minimize I/O workload. Since our goal is to synthesize in-memory data structures, many of the traditional database constraints do not apply. Researchers have investigated specialized databases that operate entirely in main memory [DeWitt et al., 1984; Garcia-Molina and Salem, 1992], a topic that has seen a recent revival of interest [Ousterhout et al., 2010]. Most main memory databases in the literature represent data using variants of the B+-tree or T-tree index structures [Lehman and Carey, 1986], or one of their refinements [Rao and Ross, 1999, 2000; Lu et al., 2000; Bin, 2003]. Evidence from a variety of systems [Lehman et al., 1992; Baulier et al., 1999; TimesTen, 1999] indicates that substantial performance improvements over traditional on-disk databases are possible by tailoring the indexing structures of a database to an in-memory environment, even when the entire database is small enough to fit in cache. Our work takes the trend a step further by providing a decomposition language that allows a programmer to describe and generate highly specialized in-memory implementations of a relation, thereby extracting the best possible performance.

Unlike a traditional database, in data representation synthesis the compiler knows the set of queries at compile time, and we can specify custom representations tailored to the particular relation and the query workload using the decomposition language. Since our goal is to replace hand-written data structure code, our compiler eliminates much of the overhead of query evaluation by generating code for each query specialized to the decomposition. Synthesized relations are correct by construction; provided that the client code obeys the functional dependencies of the relational specification, we show that the code generated by the compiler is a faithful implementation of the relational specification.

A novel aspect of our approach compared to a traditional database is that our relations can have specified restrictions (specifically, functional dependencies). These restrictions, together with the fact that in the context of compilation the set of possible queries is known in advance, enable a wider range of possible implementations, particularly representations in which objects are shared between multiple different indices.

Beginning with Cohen and Campbell [1993] researchers have proposed light-weight in-memory databases, which compile relational abstract data types into combinations of container data structures. Batory and Thomas [1997] and the subsequent extensions of their work [Smaragdakis and Batory, 1997; Batory et al., 2000] also investigate light-weight databases, including language extensions to support relations, and techniques for assisting the programmer in choosing a good representation. Rothamel and Liu [2007] also describe a relational abstract data type, where the underlying implementation may adapt based on observed usage patterns. We present the first formal results, based on a formally-specified decomposition language capable of representing sharing, together with adequacy conditions that characterize sensible representations of a relation. We show that our compiler generates code that is correct, and we report our experience with an implementation. Finally, we extend our synthesis approach to support shared-memory concurrency using locks.

We also describe a dynamic autotuner that can automatically synthesize the best decomposition for a particular relation, and we present our experience with a full implementation of these techniques in practice. The autotuner framework has a similar goal to AutoAdmin [Chaudhuri and Narasayya, 1997]. AutoAdmin takes a set of tables, together with a distribution of input queries, and identifies a set of indices that are predicted to produce the best overall performance under the query optimizer's cost model. The details differ because our decomposition and query languages are unlike those of a conventional database.

Synthesizing specialized data representations from a relational description has previously been considered in other domains. The Bernoulli project [Kotlyar et al., 1997; Ahmed et al., 2000] investigated transforming dense matrix computations into implementations tailored to specific sparse representations as a technique for handling the proliferation of complicated sparse representations.

1.4.1 Relational Programming Constructs

Many authors propose adding relations to both general- and special-purpose programming languages (e.g., [Bierman and Wren, 2005; Meijer et al., 2006; Rothamel and Liu, 2007; Vaziri et al., 2007]). We focus on the orthogonal problem of generating efficient implementations for a relational abstraction. Data models such as Entity-Relationship diagrams and the Unified Modeling Language also rely heavily on relations. One potential application of our technique is to close the gap between modeling languages and implementations.

The problem of automatic data structure selection was first explored in SETL [Schonberg et al., 1979; Paige and Henglein, 1987; Cai and Paige, 1991]; recent work has also explored automatic selection of Java collection implementations [Shacham et al., 2009]. The SETL representation sublanguage [Dewar et al., 1979] maps abstract SETL set and map objects to implementations, although the details are quite different from our work. Unlike SETL, we handle relations of arbitrary arity, using functional dependencies to enforce sharing invariants. In SETL, set representations are dynamically embedded into carrier sets under the control of the runtime system, while by contrast our compiler synthesizes low-level representations for a specific decomposition with no runtime overhead.

1.5 Data Representation Synthesis and Verification

Reasoning about the heap is a fundamental building block of tools that reason about and transform programs; despite decades of research, reasoning automatically and precisely about the heap is an extremely challenging task. Data representation synthesis gives us new insight into the problem of pointer analysis. Relations are a convenient abstraction for encapsulating a collection of data structures. This encapsulation allows us to state and guarantee sharing properties of data structures, even properties that would be difficult to verify on similar code written by hand.

Consider the task of adding a fresh `inode` into the `inode` cache. To do so in the original Linux formulation requires two operations—firstly, the `inode` must be added the list of `inode` objects belonging to its parent `super_block`. Secondly, the `inode` must be added to one of the two lists `inode_in_use` and `inode_unused`, depending on whether the inode is in use or not. It is the programmer's responsibility to keep the data structures consistent by ensuring that an inode is on a superblock's list of children if and only if it is also on exactly one of the in-use or unused lists. Stating such aliasing properties is difficult, and checking them is at the forefront of shape analysis techniques. In the synthesis context, the code generated by the compiler maintains the correct aliasing automatically—since we have proven the translation correct, no additional verification of the generated code is required.

There is a large body of literature devoted to the problem of analyzing and reasoning about pointer-manipulating code. We briefly survey conventional approaches to reasoning about the heap, and show that our synthesis approach is a compelling alternative.

1.5.1 Static Analysis and Verification Techniques

Researchers have developed many techniques for reasoning about pointer-manipulating code, including both fully-automated techniques such as points-to and shape analyses, and partially-automated techniques based on decision procedures and proof assistants. Each approach differs in its level of automation, its level of precision, and its computational complexity.

Alias and Points-To Analyses The most common types of pointer analysis are alias analyses and points-to analyses. Alias analyses attempt to determine whether two expressions in a program may refer to the same object at run time. Points-to analyses attempt to determine which object names each pointer expression may point to at run time. Pointer analyses are broadly classified by how pointer expressions are named (context-sensitivity), how objects are named (context-sensitivity, object-sensitivity), and whether the flow of information in the analysis respects the control flow of the program (inclusion-based versus unification-based, flow-sensitivity). Influential examples of pointer analyses from the literature include the works of Weihl [1980], Landi and Ryder [1992], Deutsch [1992], Landi et al. [1993], Emami et al. [1994], Andersen [1994], Steensgaard [1996], Wilson and Lam [1995], Fähndrich et al. [2000], Das [2000], and Whaley and Lam [2004].

Although alias and points-to analyses are essential building blocks for a wide range of optimizing compilers, bug-finding tools, and program-understanding tools, neither technique has the precision necessary for general verification problems. With very few exceptions, all alias and points-to analyses lose precision when reasoning about programs that manipulate recursive data structures. Since object and expression contexts can be of unbounded size in the presence of recursion, almost all practical analyses resort to bounding the size of contexts, trading precision for tractability.

Shape Analysis and Separation Logic Shape analyses are static pointer analyses which are more accurate than alias or points-to analyses, albeit with a larger computational cost. No existing shape analysis or separation logic domain is precise enough to verify arbitrary code, while simultaneously being scalable to large systems, or robust enough to give accurate results without expert supervision. Shape analysis is a difficult inference problem; a shape analysis must infer a high-level model of the heap from a collection of low-level pointer manipulations, divining the programmer's unwritten intent. The insight of this dissertation is that, by comparison, synthesizing low-level pointer manipulations from a high-level specification is far easier!

Early shape analyses included the works of Chase et al. [1990] and Hendren et al. [1992]. One of the most influential shape analysis systems is the Three-Valued Logic Analysis (TVLA) [Sagiv et al., 2002], which is a parametric abstract domain based on three-valued logic. TVLA allows an analysis designer to tune the precision of the analysis by choosing different instrumentation predicates; the ability to customize the precision of the analysis allows TVLA to perform many challenging verification tasks. TVLA has two main drawbacks: firstly, designing suitable instrumentation predicates requires a great deal of expertise, and secondly TVLA has not been shown to scale to large systems. Kreiker et al. [2010] extended TVLA to model intrusive data structures like those exhibited by the Linux filesystem example of Section 1.2.

Much recent work on reasoning about the heap is based on Separation Logic [Reynolds, 2002]. Researchers have used separation logic to prove code correct by hand [Bornat et al., 2004; Birkedal et al., 2004], and as the basis for a variety of automatic shape-analysis domains [Lee et al., 2005; Berdine et al., 2005; Magill et al., 2006; Distefano et al., 2006; Gotsman et al., 2006; Guo et al., 2007]. Two important research goals have been to make analyses based on separation logic more expressive [Berdine et al., 2007; Lee et al., 2011], and more scalable [Yang et al., 2008; Calcagno et al., 2009]; despite much progress, shape analyses with the necessary combination of precision and scalability do not yet exist. By contrast the techniques described in this thesis allow us to synthesize examples of data structures with sharing patterns that are extremely difficult for analyses based on separation logic to infer [Lee et al., 2011], without the need for unscalable whole-program analyses.

Researchers have also proposed other abstract domains for shape analysis [e.g., Gulwani and Tiwari, 2007], although these suffer from the same problems of precision and scalability. Marron et al. [2007] proposed a novel abstract domain that models collection libraries as primitives, much as we take containers as a primitive building block for synthesis. Several authors have described weaker but more scalable analyses whose precision lies in between classical pointer analyses and shape analyses [Hackett and Rugina, 2005; Naik and Aiken, 2007]; such analyses cannot reason about the kinds of sharing patterns which our decomposition language can describe.

Decidable Logics and Verification Approaches Another important research focus has been on decision procedures for restricted classes of heap structures. Graph types [Klarlund and Schwartzbach, 1993] and the subsequent Pointer Assertion Logic Engine [Møller and Schwartzbach, 2001] are a powerful formalism based on monadic second-order logic for reasoning about heaps that have a tree-like backbone, together with extra pointer edges functionally determined by the tree backbone. By contrast the decomposition language developed in this dissertation allows us to describe overlapping data structures, which do not satisfy the tree backbone condition by definition.

McPeak and Necula [2005] describe a logic based on equality axioms for reasoning about pointer data structures. Yorsh et al. [2007] describe a decidable logic of reachable patterns, based on a fragment of first-order logic with transitive closure. Chatterjee et al. [2007] describe a verifier based on reachability predicates. None of these techniques are capable of reasoning about the patterns of sharing between data structures that are common in systems code. This dissertation takes a fundamentally different approach; rather than attempting to verify data structure code after the fact, instead we synthesize correct code from a declarative specification.

The Hob system uses abstract sets of objects to specify and verify end-to-end properties of complete systems [Kuncak et al., 2006; Lam et al., 2005; Lam, 2007]. Researchers have also developed systems capable of mechanically verifying structures such as hash tables that implement a binary relational interface, such as Jahob [Zee et al., 2008, 2009], and Ynot [Chlipala et al., 2009]. Our approach considers a more general relational interface, taking individual container data structures as primitives. The code generated by our compiler could be combined with containers verified by systems such as Jahob, thereby obtaining end-to-end correctness guarantees.

Other Approaches Role analysis [Kuncak et al., 2002] characterizes objects by their aliasing relationships with other objects. Like a points-to analysis role-analysis is incapable of distinguishing between multiple objects with the same role, and hence is not precise enough for verification; however, role analysis is one of the few techniques capable of performing useful reasoning about objects that participate in multiple data structures simultaneously, such as some of the examples we generate. Ownership types [Heine and Lam, 2003] are a type-based approach for reasoning about tree-like heaps where every object has exactly one owner; ownership types have been used to solve problems such as detect memory leaks. Ownership types typically can only reason about tree-like ownership relationships.

Other previous work investigated modular reasoning about data structures shared between different modules [Juhasz et al., 2009], whereas we focus on patterns of sharing within the representation of a single relation.

Monotonic typestates enable aliased objects to monotonically change their typestates in the presence of sharing without violating type safety [Fähndrich and Leino, 2003]; our approach does not assume monotonicity.

Region type systems [Tofte and Talpin, 1994] aggregate objects into groups, called regions; by coarsening the granularity of analysis to regions consisting of many related objects rather than individual objects, some verification problems become more tractable. At granularities smaller than a region such type systems do not provide any useful aliasing information; the decomposition language provides a precise characterization of aliasing relationships within the representation of a relation.

1.6 Collaborators and Publications

The work described in this thesis is joint work, in collaboration with Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv, and previously published in a sequence of conference papers [Hawkins et al., 2010, 2011, 2012a,b].

Chapter 2

Data Representation Synthesis

In this chapter we formally describe *data representation synthesis*. In our approach, a data structure client writes code that describes and manipulates data at a high-level as *relations*; a data structure designer then provides *decompositions* which describe how those relations should be represented in memory as a combination of primitive data structures. Our compiler RELC takes a relation and its decomposition and synthesizes efficient and correct low-level code that implements the relational interface.

Each section of this chapter highlights a contribution of our work:

- We describe a scheme for synthesizing efficient low-level data representations from abstract relational descriptions of data (Section 2.1). We describe a relational interface that abstracts data from its concrete representation.
- The decomposition language (Section 2.2) specifies how relations should be mapped to low-level physical implementations, which are assembled from a library of primitive data structures. The decomposition language provides a new way to specify high-level heap invariants that are difficult or impossible to express using standard data abstraction or heap-analysis techniques. We describe adequacy conditions that ensure a decomposition faithfully represents a relation.
- We synthesize efficient implementations of queries and updates to relations, tailored to the specified decomposition (Section 2.3). Key to our approach is a query planner that chooses an efficient strategy for each query or update. We show queries and updates are sound, that is, each query or update implementation faithfully implements its relational specification.
- A programmer may not know the best decomposition for a particular relation. We describe an autotuner (Section 2.4), which given a relational specification and a performance metric finds the best decomposition up to a user-specified bound.
- The compiler RELC (Section 2.5) takes as input a relation and its decomposition, and generates C++ code implementing the relation, which is easily incorporated into existing systems. We show different

decompositions lead to very different performance characteristics. We incorporate synthesis into three real systems, namely a web server, a network accounting daemon and a map viewer, in each case leading to code that is simpler, correct by construction, and comparable in performance.

Related work for this chapter was discussed in Section 1.4 and Section 1.5.

2.1 Relational Abstraction

We first introduce the relation abstraction via which data structure clients manipulate synthesized data representations. Representing and manipulating data as relations is familiar from databases, and our interface is largely standard. We use relations to abstract a program's data from its representation. Describing particular representations is the task of the decomposition language of Section 2.2.

A *relational specification* is a set of column names C and functional dependencies Δ. As a running example, consider a simple operating system process scheduler. Each process has an ID pid and a namespace ns, a $state$ (running or sleeping), and a variety of statistics such as the cpu time consumed. The combination of a namespace ns and a process ID pid uniquely identify a process; two processes in different namespaces may share the same pid value. A natural way to model the processes is as a relation with columns $\{ns, pid, state, cpu\}$, where the values of $state$ are drawn from the set $\{S, R\}$, representing sleeping and running processes respectively, and the other columns have integer values. Not every relation represents a valid set of processes; all meaningful sets of processes satisfy a functional dependency $ns, pid \rightarrow state, cpu$, which allows at most one $state$ or cpu value for any given process. To formally define relational specifications, we need to fix notation for values, tuples, and relations:

Values, Tuples, Relations We assume a set of untyped values v drawn from a universe $\mathbb{V}$ that includes the integers ($\mathbb{Z} \subseteq \mathbb{V}$). A *tuple* $t = \langle c_1\colon v_1, c_2\colon v_2, \dots \rangle$ maps a set of *columns* $\{c_1, c_2, \dots\}$ to values drawn from $\mathbb{V}$. We write $\operatorname{dom} t$ for the columns of t. A tuple t is a *valuation* for a set of columns C if $\operatorname{dom} t = C$. A *relation* r is a set of tuples $\{t_1, t_2, \dots\}$ over identical columns C. We write $t(c)$ for the value of column c in tuple t. We write $t \supseteq s$ if the tuple t *extends* tuple s, that is $t(c) = s(c)$ for all c in $\operatorname{dom} s$. We say tuple t *matches* tuple s, written $t \sim s$, if the tuples are equal on all common columns. Tuple t matches a relation r, written $t \sim r$, if t matches every tuple in r. We write $s \lhd t$ for the merge of tuples s and t, taking values from t wherever the two disagree on a column's value. For example, the scheduler might represent three processes as the relation:

$$\begin{aligned} r_s = \{ &\langle ns\colon 1, pid\colon 1, state\colon S, cpu\colon 7\rangle, \\ &\langle ns\colon 1, pid\colon 2, state\colon R, cpu\colon 4\rangle, \\ &\langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle\} \end{aligned} \tag{2.1}$$

Functional Dependencies A relation r has a *functional dependency* (FD) $C_1 \rightarrow C_2$ if any pair of tuples in r that are equal on columns C_1 are also equal on columns C_2. We write $r \models_{\text{fd}} \Delta$ if the set of FDs Δ hold on relation r. If a FD $C_1 \to C_2$ is a consequence of FDs Δ we write $\Delta \vdash_{\text{fd}} C_1 \to C_2$. Sound and complete inference rules for functional dependencies are well-known [Beeri et al., 1977].

Relational Algebra We use the standard notation of relational algebra. Union ($\cup$), intersection ($\cap$), set difference ($\setminus$), and symmetric difference ($\ominus$) have their usual meanings. The operator $\pi_C\, r$ projects relation r onto a set of columns C, and $r_1 \bowtie r_2$ is the natural join of relation r_1 and relation r_2.

Relational Operations We provide five operations for creating and manipulating relations. Here we represent relations as ML-like references to a set of tuples; ref x denotes creating a new reference to x, $!r$ fetches the current value of r and $r \leftarrow v$ sets the current value of r to v:

$$\begin{aligned}
\mathsf{empty}\ () &= \mathsf{ref}\ \emptyset \\
\mathsf{insert}\ r\ t &= r \leftarrow\ !r \cup \{t\} \\
\mathsf{remove}\ r\ s &= r \leftarrow\ !r \setminus \{t \in\ !r \mid t \supseteq s\} \\
\mathsf{update}\ r\ s\ u &= r \leftarrow \{\mathsf{if}\ t \supseteq s\ \mathsf{then}\ t \lhd u\ \mathsf{else}\ t \mid t \in\ !r\} \\
\mathsf{query}\ r\ s\ C &= \pi_C \{t \in\ !r \mid t \supseteq s\}
\end{aligned}$$

Informally, empty () creates a new empty relation. The operation insert $r\ t$ inserts tuple t into relation r, remove $r\ s$ removes tuples matching tuple s from relation r, and update $r\ s\ u$ applies the updates in tuple u to each tuple matching s in relation r. Finally query $r\ s\ C$ returns the columns C of all tuples in r matching tuple s. The tuples s and u given as arguments to the remove, update and query operations may be partial tuples, that is, they need not contain every column of relation r. Extending the query operator to handle comparisons other than equality or to support ordering is straightforward; however, for clarity of exposition we restrict ourselves to queries based on equalities.

For the scheduler example, we call empty () to obtain an empty relation r. To insert a new running process into r, we invoke:

$$\mathsf{insert}\ r\ \langle ns{:}\, 7, pid{:}\, 42, state{:}\, R, cpu{:}\, 0\rangle$$

The operation

$$\mathsf{query}\ r\ \langle state{:}\, R\rangle\ \{ns, pid\}$$

returns the namespace and ID of each running process in r, whereas

$$\mathsf{query}\ r\ \langle ns{:}\, 7, pid{:}\, 42\rangle\ \{state, cpu\}$$

returns the state and cpu of process 42 in namespace 7. By invoking

$$\mathsf{update}\ r\ \langle ns{:}\, 7, pid{:}\, 42\rangle\ \langle state{:}\, S\rangle$$

```
class scheduler_relation {
  void insert(tuple_cpu_ns_pid_state const &r);
  void remove(tuple_ns_pid const &pattern);
  void update(tuple_ns_pid const &pattern,
              tuple_cpu_state const &changes);
  void query(tuple_state const &input,
             iterator_state__ns_pid &output);
  ...
};
```

Figure 2.1: The C++ class generated for the scheduler relation.

we can mark process 42 as sleeping, and finally by calling

$$\mathsf{remove}\ r\ \langle ns\colon 7, pid\colon 42\rangle$$

we can remove the process from the relation.

The RELC compiler emits C++ classes that implement the relational interface, which client code can then call. For the scheduler relation example the compiler generates the class shown in Figure 2.1. Each method of the class instantiates a relational operation. We could generate instantiations of each operation for all possible kinds of tuples passed as arguments, however in practice we allow the programmer to specify the needed instantiations.

2.2 Decompositions and Decomposition Instances

Decompositions describe how to represent relations as a combination of primitive data structures. Our goal is to prove that the low-level representation of a relation faithfully implements its high-level specification. In this section, we develop the technical machinery to reason about the correspondence between relations and their decompositions.

A decomposition is a static description of the structure of data, akin to a type. Its run-time (dynamic) counterpart is the *decomposition instance*, which describes the representation of a particular relation using the decomposition. We define an *abstraction function* that computes the relation represented by a given decomposition instance, and *well-formedness criteria* that check that a decomposition instance is a well-formed instance of a particular decomposition. Finally, we define *adequacy conditions* which are sufficient conditions for a decomposition to faithfully represent a relation.

2.2.1 Decompositions

A *decomposition* is a rooted, directed acyclic graph that describes how to represent a relational specification. The subgraph rooted at each node of the decomposition describes how to represent part of the original relation;

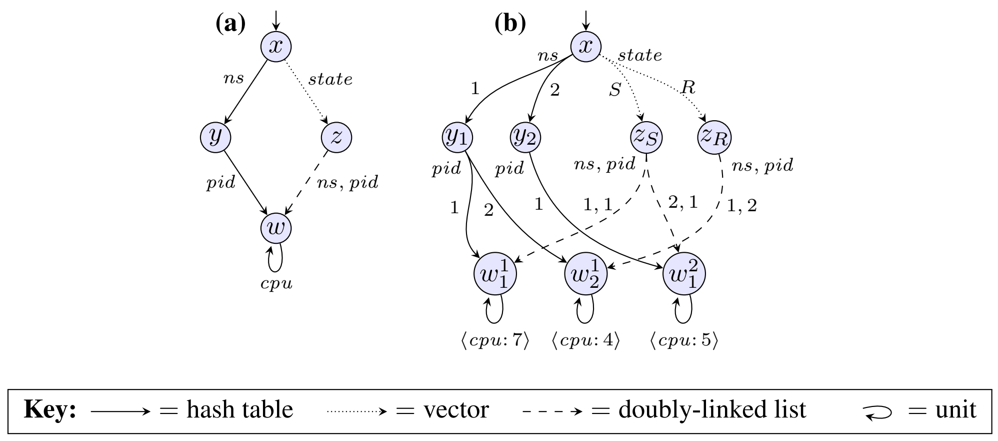


Figure 2.2: Data representation for a process scheduler: **(a)** a decomposition, **(b)** an instance of that decomposition. Solid edges represent hash tables, dotted edges represent vectors, and dashed edges represent doubly-linked lists.

each edge of the decomposition describes a way of breaking up a relation into a set of smaller relations.

We use the scheduler example to explain the features of the decomposition language. Figure 2.2(a) shows one possible decomposition for the scheduler relation. Informally, this decomposition reads as follows. From the root (node x), we can follow the left-hand edge, which uses a hash table to map each value n of the ns field to a sub-relation (node y) with the $\{pid, cpu\}$ values for n. From one such sub-relation, the outgoing edge of node y maps a pid (using another hashtable) to a sub-relation consisting of a single tuple with one column, the corresponding cpu time. The $state$ field is not represented on the left-hand path. Alternatively, from the root we can follow the right-hand edge, which maps a process $state$ (running or sleeping) to a sub-relation of the $\{ns, pid, cpu\}$ values of the processes in that state. Each such sub-relation (rooted at node z) maps a $\{ns, pid\}$ pair to the corresponding cpu time. While the left path from x to w is implemented using a hash table of hash tables, the right path is a vector with two entries, one pointing to a list of running processes, the other to a list of sleeping processes. Because node w is shared, there is only one physical copy of each cpu value, shared by the two access paths.

A *decomposition instance*, or *instance* for short, is a rooted, directed acyclic graph representing a particular relation. Each node of a decomposition corresponds to a set of nodes in an instance of that decomposition. Figure 2.2(b) shows an instance of the decomposition representing the relation r_s defined in Equation (2.1). The structure of an instance corresponds to a low-level memory state; nodes are objects in memory and edges are data structures navigating between objects. Note, for example, node z_S has two outgoing edges, one for each sleeping process; the dashed edge indicates that the collection of sleeping processes is implemented as a doubly-linked list.

To reason formally about decompositions and decomposition instances we encode graphs in a let-binding notation, using the language shown in Figure 2.3 for decompositions and Figure 2.4 for instances. We stress that this notation is isomorphic to the graph notation and only exists to aid formal reasoning.

$\hat{p} ::= C \mid C \overset{\psi}{\mapsto} v \mid \hat{p}_1 \bowtie \hat{p}_2$	decomposition primitives
$\hat{d} ::= \mathsf{let}\ v\colon B \triangleright C = \hat{p}\ \mathsf{in}\ \hat{d} \mid v$	decompositions
$\psi ::= \mathsf{dlist} \mid \mathsf{htable} \mid \mathsf{vector} \mid \cdots$	data structures

Figure 2.3: Grammar of the decomposition language.

$p ::= t \mid \{t \mapsto v_{t'}, \dots\} \mid p_1 \bowtie p_2$	instance primitives
$d ::= \mathsf{let}\ \{v_t = p, \dots\}\ \mathsf{in}\ d \mid v_{\langle\rangle}$	instances

Figure 2.4: Grammar of decomposition instances.

Figure 2.5(a) shows the decomposition of Figure 2.2(a) written in let-notation. In a decomposition a let-binding $\mathsf{let}\ v\colon B \triangleright C = \hat{p}\ \mathsf{in}\ \hat{d}$ allows us to share instances of the sub-relation v with decomposition $\hat{p}$ between multiple parts of a decomposition $\hat{d}$. Let-bound variables must be distinct (to avoid name conflicts) and in $\mathsf{let}\ v\colon B \triangleright C = \hat{p}\ \mathsf{in}\ \hat{d}$, variable v must appear in $\hat{d}$ (to ensure the decomposition graph is connected). Each decomposition variable is annotated with a "type," consisting of a pair of column sets $B \triangleright C$; every instance of variable v in a decomposition instance has a distinct valuation of columns B, and each instance represents a relation with columns C.

Figure 2.5(b) shows the decomposition instance of Figure 2.2(b) written in the let-notation of instances. Each let-binding in the instance parallels a binding of $v\colon B \triangleright C$ in the decomposition; the instance binds a set of variable instances $\{v_t, v_{t'}, \dots\}$, each for different valuations t of columns B. For example, decomposition node z is annotated $z\colon \{state\} \triangleright \{ns, pid, cpu\}$. There are two instances of node z in the decomposition instance, namely $z_{\langle state\colon S\rangle}$ and $z_{\langle state\colon R\rangle}$, one for each valuation of the $state$ column in the relation. The subgraph rooted at each instance of node z represents a subrelation with columns $\{ns, pid, cpu\}$.

We now describe the three decomposition primitives and their corresponding decomposition instance primitives.

- A *unit* C represents a single tuple t with columns C. Unit decompositions in diagrams are shown as self-loops on a node, labeled with columns C. For example, in Figure 2.2(a) node w has a unit decomposition containing a single cpu value.

- A *map* $C \overset{\psi}{\mapsto} v$ represents a relation r as a mapping $\{t \mapsto v_{t'}, \dots\}$ from valuations t of a set of *key columns* C, to decomposition instances $v_{t'}$, where t' is an extension of t. Each decomposition instance $v_{t'}$ represents the *residual relation* r_t, consisting of the tuples of r that match valuation t. The data structure used to implement the map is ψ, which can be any data structure that implements a key-value associative map interface. In the example ψ is one of dlist (an unordered doubly-linked list of key-value pairs), htable (a hash table), or vector (an array mapping keys to values). The set of data structures is extensible; any data structure implementing a common interface may be used. The choice of ψ only

(a)

$$
\begin{aligned}
&\mathsf{let}\ w\colon \{ns, pid, state\} \triangleright \{cpu\} = \{cpu\}\ \mathsf{in}\\
&\mathsf{let}\ y\colon \{ns\} \triangleright \{pid, cpu\} = \{pid\} \xmapsto{\mathsf{htable}} w\ \mathsf{in}\\
&\mathsf{let}\ z\colon \{state\} \triangleright \{ns, pid, cpu\} = \{ns, pid\} \xmapsto{\mathsf{dlist}} w\ \mathsf{in}\\
&\mathsf{let}\ x\colon \emptyset \triangleright \{ns, pid, cpu, state\} = (\{ns\} \xmapsto{\mathsf{htable}} y) \bowtie (\{state\} \xmapsto{\mathsf{vector}} z)\ \mathsf{in}\ x
\end{aligned}
$$

(b)

$$
\begin{aligned}
&\mathsf{let}\ \big\{w_{\langle ns:1, pid:1, state:S\rangle} = \langle cpu\colon 7\rangle\,,\\
&\quad w_{\langle ns:1, pid:2, state:R\rangle} = \langle cpu\colon 4\rangle\,,\\
&\quad w_{\langle ns:2, pid:1, state:S\rangle} = \langle cpu\colon 5\rangle\big\}\ \mathsf{in}\\
&\mathsf{let}\ \big\{y_{\langle ns:1\rangle} = \{\langle pid\colon 1\rangle \mapsto w_{\langle ns:1, pid:1, state:S\rangle}, \langle pid\colon 2\rangle \mapsto w_{\langle ns:1, pid:2, state:R\rangle}\},\\
&\quad y_{\langle ns:2\rangle} = \{\langle pid\colon 1\rangle \mapsto w_{\langle ns:2, pid:1, state:S\rangle}\}\big\}\ \mathsf{in}\\
&\mathsf{let}\ \big\{z_{\langle state:S\rangle} = \{\langle ns\colon 1, pid\colon 1\rangle \mapsto w_{\langle ns:1, pid:1, state:S\rangle}, \langle ns\colon 2, pid\colon 1\rangle \mapsto w_{\langle ns:2, pid:1, state:S\rangle}\},\\
&\quad z_{\langle state:R\rangle} = \{\langle ns\colon 1, pid\colon 2\rangle \mapsto w_{\langle ns:1, pid:2, state:R\rangle}\}\big\}\ \mathsf{in}\\
&\mathsf{let}\ \big\{x_{\langle\rangle} = \{\langle ns\colon 1\rangle \mapsto y_{\langle ns:1\rangle}, \langle ns\colon 2\rangle \mapsto y_{\langle ns:2\rangle}\}\\
&\quad \bowtie \{\langle state\colon S\rangle \mapsto z_{\langle state:S\rangle}, \langle state\colon R\rangle \mapsto z_{\langle state:R\rangle}\}\big\}\ \mathsf{in}\\
&\quad x_{\langle\rangle}
\end{aligned}
$$

Figure 2.5: The process scheduler examples of Figure 2.2 written in let-notation. Part **(a)** shows the decomposition; part **(b)** shows the decomposition instance.

affects the computational complexity of operations on a data structure; where the complexity is irrelevant we omit ψ and simply write $C \mapsto v$. In diagrams we depict map decompositions as edges labeled with the set of columns C. For example, in Figure 2.2(a) the edge from y to w labeled pid indicates that for each instance of vertex y in a decomposition instance there is a data structure that maps each value of pid to a different residual relation, represented using the decomposition rooted at w.

- A *join* $\hat{p}_1 \bowtie \hat{p}_2$ represents a relation as the natural join of two different sub-relations r_1 and r_2, where $\hat{p}_1$ describes how to decompose r_1 and $\hat{p}_2$ describes how to decompose r_2. In diagrams, join decompositions exist wherever multiple map edges exit the same node. For example, in Figure 2.2(a) node x has two outgoing map edges and hence is the join of two map decompositions.

2.2.2 Abstraction Function

The abstraction functions $\alpha(d, \Gamma)$ and $\alpha(p, \Gamma)$ shown in Figure 2.6 map instances d and instance primitives p, respectively, to the relation they represent. Argument Γ is an environment that maps instance variables to definitions. We write "$\cdot$" to denote the initial empty environment.

2.2.3 Well-formed Decomposition Instances

Next we introduce a well-formedness invariant ensuring that the structure of an instance d corresponds to that of a decomposition $\hat{d}$. We say that a decomposition instance d is a *well-formed instance* of a decomposition $\hat{d}$

$$\begin{aligned}
\alpha(t, \Gamma) &= \{t\} \\
\alpha(\{t \mapsto v_{t'}\}_{t \in T}, \Gamma) &= \bigcup_{t \in T} \big(\{t\} \bowtie \alpha(v_{t'}, \Gamma)\big) \\
\alpha(p_1 \bowtie p_2, \Gamma) &= \alpha(p_1, \Gamma) \bowtie \alpha(p_2, \Gamma) \\
\alpha(\mathsf{let}\ \{v_t = p_t\}_{t \in T}\ \mathsf{in}\ d, \Gamma) &= \alpha(d, \Gamma \cup \{v_t \mapsto p_t \mid t \in T\}) \\
\alpha(v_t, \Gamma) &= \{\alpha\left(\Gamma(v_t), \Gamma\right)\}
\end{aligned}$$

Figure 2.6: The abstraction functions $\alpha(d, \Gamma)$ and $\alpha(p, \Gamma)$

$$(\textsc{WfUnit})\ \frac{\operatorname{dom} t = C}{\Gamma, t \models \hat{\Gamma}, C} \qquad (\textsc{WfMap})\ \frac{\begin{array}{c}\forall t \in T.\ \operatorname{dom} t = C \\ t \sim \alpha(v_{t'}, \Gamma) \qquad \Gamma, v_{t'} \models \hat{\Gamma}, v\end{array}}{\Gamma, \{t \mapsto v_{t'}\}_{t \in T} \models \hat{\Gamma}, C \mapsto v}$$

$$(\textsc{WfJoin})\ \frac{\begin{array}{cc}\Gamma, p_1 \models \hat{\Gamma}, \hat{p}_1 & \Gamma, p_2 \models \hat{\Gamma}, \hat{p}_2 \\ r_1 = \alpha(p_1, \Gamma) & r_2 = \alpha(p_2, \Gamma) \\ \multicolumn{2}{c}{\pi_{\operatorname{dom} r_2} r_1 = \pi_{\operatorname{dom} r_1} r_2}\end{array}}{\Gamma, p_1 \bowtie p_2 \models \hat{\Gamma}, \hat{p}_1 \bowtie \hat{p}_2} \qquad (\textsc{WfVar})\ \frac{\Gamma, \Gamma(v_t) \models \hat{\Gamma}, \hat{\Gamma}(v)}{\Gamma, v_t \models \hat{\Gamma}, v}$$

$$(\textsc{WfLet})\ \frac{\forall t \in T.\ \operatorname{dom} t = B \qquad \Gamma \cup \{v_t \mapsto p_t\}_{t \in T}, d \models \hat{\Gamma} \cup \{v \mapsto \hat{p}\}, \hat{d}}{\Gamma, \mathsf{let}\ \{v_t = p_t\}_{t \in T}\ \mathsf{in}\ d \models \hat{\Gamma}, \mathsf{let}\ v \colon B \triangleright C = \hat{p}\ \mathsf{in}\ \hat{d}}$$

Figure 2.7: Well-formed decomposition instances: $\Gamma, d \models \hat{\Gamma}, \hat{d}$ and $\Gamma, p \models \hat{\Gamma}, \hat{p}$

if $\cdot, d \models \cdot, \hat{d}$ follows from the rules given in Figure 2.7. The first argument to the judgment is an environment Γ mapping instance variables to definitions; similarly the third argument $\hat{\Gamma}$ is an environment mapping decomposition variables to definitions. Rule (WFUNIT) checks that a unit node is a tuple with the correct columns. Rule (WFMAP) checks that each key tuple t has the correct columns, that t matches all tuples in the associated residual relation, and that variable instance $v_{t'}$ is well-formed. Rule (WFJOIN) checks that we do not have "dangling" tuples on one side of a join without a matching tuple on the other side. Rule (WFLET) introduces variables into environments Γ and $\hat{\Gamma}$ and checks variable instantiations have the correct columns. Finally rule (WFVAR) checks the definition of a variable is well-formed.

2.2.4 Adequacy of Decompositions

Not every relation can be represented by every decomposition. In general a decomposition can only represent relations with specific columns satisfying certain functional dependencies. For example the decomposition $\hat{d}$

$$\text{(AVAR)}\ \frac{(v\colon \emptyset \rhd C) \in \Sigma}{\Sigma; \emptyset \vdash_{a,\Delta} v; C} \qquad \text{(AUNIT)}\ \frac{A \neq \emptyset \qquad \Delta \vdash_{\text{fd}} A \to C}{\Sigma; A \vdash_{a,\Delta} C; C}$$

$$\text{(AMAP)}\ \frac{(v\colon A \rhd D) \in \Sigma \qquad \Delta \vdash_{\text{fd}} B \cup C \to A \qquad A \supseteq B \cup C}{\Sigma; B \vdash_{a,\Delta} C \mapsto v; C \cup D}$$

$$\text{(AJOIN)}\ \frac{\Delta \vdash_{\text{fd}} A \cup (B \cap C) \to B \ominus C \qquad \Sigma; A \vdash_{a,\Delta} \hat{p}_1; B \qquad \Sigma; A \vdash_{a,\Delta} \hat{p}_2; C}{\Sigma; A \vdash_{a,\Delta} \hat{p}_1 \bowtie \hat{p}_2; B \cup C}$$

$$\text{(ALET)}\ \frac{\Sigma; B \vdash_{a,\Delta} \hat{p}; C \qquad \Sigma, v\colon B \rhd C; A \vdash_{a,\Delta} \hat{d}; D}{\Sigma; A \vdash_{a,\Delta} \mathsf{let}\ v\colon B \rhd C = \hat{p}\ \mathsf{in}\ \hat{d}; D}$$

Figure 2.8: Adequate decompositions: $\Sigma; A \vdash_{a,\Delta} \hat{d}; B$ and $\Sigma; A \vdash_{a,\Delta} \hat{p}; B$

in Figure 2.2(a) cannot represent the relation

$$\begin{aligned} r' = \{ & \langle ns\colon 1, pid\colon 2, state\colon S, cpu\colon 42 \rangle, \\ & \langle ns\colon 1, pid\colon 2, state\colon R, cpu\colon 34 \rangle \}, \end{aligned}$$

since for each pair of ns and pid values the decomposition $\hat{d}$ can only represent a single value for the $state$ and cpu fields. However r' does not correspond to a meaningful set of processes—the relational specification in Section 2.1 requires that all well-formed sets of processes satisfy the functional dependency $ns, pid \to state, cpu$, which allows at most one $state$ or cpu value for any given process.

We say that a decomposition $\hat{d}$ is *adequate* for relations with columns C satisfying FDs Δ if $\cdot; \emptyset \vdash_{a,\Delta} \hat{d}; C$ follows from the rules in Figure 2.8.

There are two forms of the adequacy judgement, one for decompositions: $\Sigma; A \vdash_{a,\Delta} \hat{d}; B$, and one for decomposition primitives: $\Sigma; A \vdash_{a,\Delta} \hat{p}; B$. The first argument to the judgement is an environment Σ that maps a variable v bound in the context to a pair $B \rhd C$, where B is the set of columns bound on any path to node v from the root of the decomposition, and C is the set of columns bound within the subgraph rooted at v. The second argument A is a set of columns fixed by the context. If a decomposition $\hat{d}$ is adequate, then it can represent every possible relation with columns C satisfying FDs Δ:

Lemma 2.1 (Soundness of Adequacy)**.** If $\cdot; \emptyset \vdash_{a,\Delta} \hat{d}; C$ then for each relation r with columns C such that $r \models_{\text{fd}} \Delta$ there is some d such that $\cdot, d \models \cdot, \hat{d}$ and $\alpha(d, \cdot) = r$.

Proof. See Section 2.6.1. □

The adequacy rules enforce several properties, most of which are boundary conditions. Rule (AVAR) ensures the root vertex has exactly one instance (since $\emptyset$ has only one valuation). Rules (AUNIT) and (AMAP) record the columns they contain, and the top-level rule (AVAR) then ensures the decomposition represents all columns of the relation. Rule (AUNIT) also ensures that unit decompositions are not part of the graph root. Since a unit decomposition represents exactly one tuple, a unit decomposition at the root ($A = \emptyset$) would prevent us from representing the empty relation.

Rule (AMAP) is the most involved and consequential rule. Sharing occurs when the same variable is the target of two or more maps (see the uses of variable w in Figure 2.5(a) for an example). Rule (AMAP) checks in two steps that decomposition instances are shared only when the corresponding relations are equal. First, note that $B \cup C$ are columns bound from the root to v, and the functional dependency $B \cup C \to A$ guarantees there is a unique valuation of A per valuation of $B \cup C$. Second, the requirement that $A \supseteq B \cup C$ guarantees that A includes all the columns bound on all paths reaching v (since this same requirement is also applied to other map edges that share v). Because $B \cup C \to A$, and A includes any other key columns used in other maps reaching v, the sub-relation reached via any of these alternative paths is the same.

To split a relation into two parts using a join decomposition, rule (AJOIN) requires a functional dependency that ensures that we can match tuples from each side without anomalies, such as missing or spurious tuples; recall $\ominus$ denotes symmetric difference. Finally rule (ALET) introduces variable typings from let bindings into the variable binding environment Σ.

2.3 Querying and Updating Decomposed Relations

In Section 2.2 we introduced decompositions, which describe how to represent a relation in memory as a collection of data structures. In this section we show how to compile the relational operations described in Section 2.1 into code tailored to a particular decomposition. There are two basic kinds of relational operation, namely queries and mutations. Since we use queries when implementing mutations, we describe queries first.

2.3.1 Queries and Query Plans

Recall that the *query* operation retrieves data from a relation; given a relation r, a tuple t, and a set of columns C, a query returns the projection onto columns C of the tuples of r that match tuple t. We implement queries in two stages: *query planning*, which attempts to find the most efficient execution plan q for a query, and *query execution*, which evaluates a particular query plan over a decomposition instance. This approach is well-known in the database literature, although our formulation is novel.

In the RELC compiler, query planning is performed at compile time; the compiler generates specialized code to evaluate the chosen plan q with no run-time planning or evaluation overhead. The compiler is free to use any method it likes to chose a query plan, as long as the resulting query satisfies the *query validity* criteria described in Section 2.3.2. We describe the query planner implementation of the RELC compiler in Section 2.3.3.

As a motivating example, suppose we want to find the set of pid values of processes that match the tuple $\langle ns\colon 7, state\colon R\rangle$ using the decomposition of Figure 2.2. That is, we want to find the running processes in namespace 7. One possible strategy would be to look up $\langle state\colon R\rangle$ on the right-hand side, and then to iterate over all ns, pid pairs associated with the state, checking to see whether they are in the correct namespace.

$$\begin{aligned} q &::= \mathsf{qunit} \mid \mathsf{qscan}(q) \mid \mathsf{qlookup}(q) \mid \mathsf{qlr}(q, lr) \mid \mathsf{qjoin}(q_1, q_2, lr) \\ lr &::= \mathsf{left} \mid \mathsf{right} \end{aligned}$$

Figure 2.9: Query plan operators

Another strategy would be to look up namespace 7 on the left-hand side, and to iterate over the set of pid values associated with the namespace. For each pid we then check to see whether the ns and pid pair is in the set of processes associated with $\langle state\colon R\rangle$ on the right-hand side. Each strategy has a different computational complexity; the query planner enumerates the alternatives and chooses the "best" strategy.

We describe the semantics of query execution using a function dqexec $q\ d\ t$ which takes a query plan q, a decomposition instance d, an input tuple t, and evaluates the plan over the decomposition, and produces a set of tuples in the denotation of d that match tuple t. We do not implement dqexec directly; instead the compiler emits instances of dqexec specialized to particular queries q.

A *query plan* is a tree of query plan operators, shown in Figure 2.9. The query plan tree is superimposed on a decomposition and rooted at the decomposition's root. A query plan prescribes an ordered sequence of nodes and edges of the decomposition instance to visit. There are five query plan operators:

Unit The qunit operator returns the unique tuple represented by a unit decomposition instance if that tuple matches t. It returns the empty set otherwise.

Scan The operator qscan(q) invokes operator q for each child node v_s where s matches t. Recall a map primitive is a mapping from a set of key columns C to a set of child nodes $\{v_t\}_{t \in T}$. Since operator qscan iterates over the contents of a map data structure, it typically takes time linear in the number of entries.

Lookup The qlookup(q) operator looks up a particular set of key values in a map decomposition; each of the key columns of the map must be bound in the tuple t given as input to the operator. Query operator q is invoked on the resulting sub-decomposition, if any. The complexity of the qlookup depends on the particular choice of data structure ψ. In general, we expect qlookup to have better time complexity than qscan.

Left/Right The qlr(q, lr) operator performs query q on either the left-hand or right-hand side of a join specified by the argument lr. The other side of the join is ignored.

Join The qjoin(q_1, q_2, lr) operator performs a join across both sides of a join decomposition. The computational complexity of the join may depend on the order of evaluation. If lr is the value left, then first query q_1 is executed on the left side of the join decomposition, then query q_2 is executed on the right side of the join for each tuple returned by tuple q_1; the result of the join operator is the natural join of the two subqueries. If lr is the value right, the two queries are executed in the opposite order.

In the scheduler example, the query

$$\mathsf{query}\ r\ \langle ns\colon 7, pid\colon 42\rangle\ \{cpu\}$$

returns the cpu values associated with the process with pid 42 in namespace 7. One possible query plan is:

$$q_{cpu} = \mathsf{qlr}(\mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{left}).$$

To perform query q_{cpu} on an instance d we evaluate

$$\mathsf{dqexec}\ q_{cpu}\ d\ \langle ns\colon 7, pid\colon 42\rangle$$

which first looks up namespace 7 in the data structure corresponding to decomposition edge from x to y, returning an instance of node y. We lookup pid 42 in the data structure corresponding to the edge from y to w, obtaining an instance of node w. We then use the qunit operator to retrieve the cpu value associated with node w.

Recall our motivating example, namely the query

$$\mathsf{query}\ r\ \langle ns\colon 7, state\colon R\rangle\ \{pid\}$$

that returns the set of running processes in namespace 7. Two plans that implement the query are

$$\begin{aligned} q_1 &= \mathsf{qjoin}\big(\mathsf{qlookup}(\mathsf{qscan}(\mathsf{qunit})), \mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{left}\big) \\ q_2 &= \mathsf{qlr}\big(\mathsf{qlookup}(\mathsf{qscan}(\mathsf{qunit})), \mathsf{right}\big). \end{aligned}$$

Plan q_1 first enumerates the set of processes with ns 7 (the left-hand side of the join), and then checks whether each process is associated with the running state (the right-hand side of the join). Plan q_2 iterates over all processes in the running state, checking to see whether they are in the appropriate namespace.

An important property of the query operators is that they all require only constant space; there is no need to construct intermediate data structures to execute a query. Having a predictable space overhead for queries ensures that query execution does not need to allocate memory. Constant-space queries can also be a disadvantage; for example, the current restrictions would not allow a "hash-join" strategy for implementing the join operator, nor is it possible to perform duplicate-elimination. It would be straightforward to extend the query language with non-constant-space operators.

2.3.2 Query Validity

Not every query plan is a correct strategy for evaluating a query. We must check three properties: first that queries produce all of the columns requested as output, second that when performing a lookup we already

$$(\textsc{QUnit})\ \frac{}{\hat\Gamma, C, A \vdash_{q,\Delta} \mathsf{qunit}, C} \qquad (\textsc{QScan})\ \frac{\hat\Gamma, \hat\Gamma(v), (A \cup C) \vdash_{q,\Delta} q, B}{\hat\Gamma, C \mapsto v, A \vdash_{q,\Delta} \mathsf{qscan}(q), B \cup C}$$

$$(\textsc{QLookup})\ \frac{C \subseteq A \qquad \hat\Gamma, \hat\Gamma(v), A \vdash_{q,\Delta} q, B}{\hat\Gamma, C \mapsto v, A \vdash_{q,\Delta} \mathsf{qlookup}(q), B \cup C}$$

$$(\textsc{QJoin})\ \frac{\begin{array}{c}\hat\Gamma, \hat p_1, A \vdash_{q,\Delta} q_1, B_1 \\ \hat\Gamma, \hat p_2, A \cup B_1 \vdash_{q,\Delta} q_2, B_2 \qquad \Delta \vdash_{\mathrm{fd}} A \cup B_1 \to B_2 \qquad \Delta \vdash_{\mathrm{fd}} A \cup B_2 \to B_1\end{array}}{\begin{array}{c}\hat\Gamma, \hat p_1 \bowtie \hat p_2, A \vdash_{q,\Delta} \mathsf{qjoin}(q_1, q_2, \mathsf{left}), B_1 \cup B_2 \\ \text{or } \hat\Gamma, \hat p_2 \bowtie \hat p_1, A \vdash_{q,\Delta} \mathsf{qjoin}(q_2, q_1, \mathsf{right}), B_1 \cup B_2\end{array}}$$

$$(\textsc{QLR})\ \frac{\hat\Gamma, \hat p_1, A \vdash_{q,\Delta} q, B}{\hat\Gamma, \hat p_1 \bowtie \hat p_2, A \vdash_{q,\Delta} \mathsf{qlr}(q, \mathsf{left}), B \qquad \text{or } \hat\Gamma, \hat p_2 \bowtie \hat p_1, A \vdash_{q,\Delta} \mathsf{qlr}(q, \mathsf{right}), B}$$

$$(\textsc{QVar})\ \frac{\hat\Gamma, \hat\Gamma(v), A \vdash_{q,\Delta} q, B}{\hat\Gamma, v, A \vdash_{q,\Delta} q, B} \qquad (\textsc{QLet})\ \frac{\hat\Gamma \cup \{v \mapsto \hat p\}, \hat d, A \vdash_{q,\Delta} q, B}{\hat\Gamma, \mathsf{let}\ v\colon \cdots = \hat p\ \mathsf{in}\ \hat d, A \vdash_{q,\Delta} q, B}$$

Figure 2.10: Valid query plans: $\hat\Gamma, \hat d, A \vdash_{q,\Delta} q, B$

have all of the necessary key columns, and third that enough columns are computed on each side of a join so that tuples from each side can be accurately matched with one another. Figure 2.10 gives inference rules for a validity judgment that is a sufficient condition for query plan correctness. We say a query plan is *valid*, written $\hat\Gamma, \hat d, A \vdash_{q,\Delta} q, B$ if q correctly answers queries over decomposition $\hat d$, where A is the set of columns bound in the input tuple pattern t and B is the set of columns bound in the output tuples; $\hat\Gamma$ is an environment that maps variables in the decomposition to their definitions, whereas Δ is a set of FDs.

Rule (QUnit) states that querying a unit decomposition binds its fields. Rule (QScan) states when scanning over a map decomposition we bind the keys of the map both as input to the sub-query and in the output. Rule (QLookup) is similar, however lookups require that the key columns already be bound in the input. Rule (QJoin) requires that each subquery of a join must bind enough columns so that we can match the results of the two subqueries without any ambiguity. As a special case, rule (QLR) allows arbitrary queries that only inspect one side of a join. Finally rules (QVar) and (QLet) handle introduction and elimination of decomposition variables into the variable binding environment $\hat\Gamma$.

Lemma 2.2 (Decomposition Query Soundness)**.** Suppose we have $\hat d$, C, Δ, d, and r such that decomposition $\hat d$ is adequate (i.e., $\cdot; \emptyset \vdash_{a,\Delta} \hat d; C$), instance d is well-formed (i.e., $\cdot, d \models \cdot, \hat d$), and d represents a relation r (i.e., $\alpha(d, \cdot) = r$) satisfying the FDs Δ (i.e., $\Delta \models_{\mathrm{fd}} r$). If a query plan q is valid for input tuples with columns A and produces columns B (i.e., $\cdot, \hat d, A \vdash_{q,\Delta} q, B$), then for any tuple s with $\mathrm{dom}\, s = A$ we have

$$\pi_B(\mathsf{dqexec}\ q\ d\ s) = \pi_B\{t \in r \mid t \supseteq s\}.$$

Proof. See Section 2.6.2. □

2.3.3 Query Planner

To pick good implementations for each query, the compiler uses a query planner that finds the query plan with the lowest cost as measured by a heuristic cost estimation function. The query planner enumerates the set of valid query plans for a particular decomposition d, input columns B, and output columns C, and it returns the plan with the lowest cost. It is straightforward to enumerate query plans, although there may be exponentially many possible plans for a query.

The RELC compiler uses a simple query cost estimator $E_{\hat{\Gamma}}$ that has performed well in our experiments. Many extensions to our cost model are possible, inspired by the large literature on database query planning. For every edge from node v_1 to node v_2 in a decomposition $\hat{d}$ we require a count $c(v_1, v_2)$ of the expected number of instances of the edge outgoing from any given instance of node v_1. The count can be provided by the user, or recorded as part of a profiling run. Each data structure ψ must provide a function $m_\psi(n)$ that estimates the number of memory accesses to lookup a key in a data structure ψ containing n elements. For a binary tree we might set $m_{\mathsf{btree}}(n) = \log_2 n$, whereas for a linked list we might set $m_{\mathsf{dlist}}(n) = n$. Let $\hat{\Gamma}$ be the environment mapping each let-bound variable in $\hat{d}$ to its definition. We compute $E_{\hat{\Gamma}}(q, v, \hat{d})$, where v is the decomposition root:

$$\begin{aligned}
&E_{\hat{\Gamma}}(\mathsf{qunit}, v, C) = 1 \\
&E_{\hat{\Gamma}}(\mathsf{qscan}(q), v_1, C \xmapsto{\psi} v_2) = c(v_1, v_2) \times E_{\hat{\Gamma}}(q, v_2, \hat{\Gamma}(v_2)) \\
&E_{\hat{\Gamma}}(\mathsf{qlookup}(q), v_1, C \xmapsto{\psi} v_2) = m_\psi(c(v_1, v_2)) \times E_{\hat{\Gamma}}(q, v_2, \hat{\Gamma}(v_2)) \\
&E_{\hat{\Gamma}}(\mathsf{qjoin}(q_1, q_2, \_), v, \hat{p}_1 \bowtie \hat{p}_2) = E_{\hat{\Gamma}}(q_1, v, \hat{p}_1) + E_{\hat{\Gamma}}(q_2, v, \hat{p}_2) \\
&E_{\hat{\Gamma}}(\mathsf{qlr}(q, \mathsf{left}), v, \hat{p}_1 \bowtie \hat{p}_2) = E_{\hat{\Gamma}}(q, v, \hat{p}_1) \\
&E_{\hat{\Gamma}}(\mathsf{qlr}(q, \mathsf{right}), v, \hat{p}_1 \bowtie \hat{p}_2) = E_{\hat{\Gamma}}(q, v, \hat{p}_2)
\end{aligned}$$

The cost estimate for joins is optimistic since it assumes that queries on each side of the join need only be performed once each, whereas in general one side of a join is executed once for each tuple yielded by the other side. We could extend the heuristic to estimate how many tuples are returned by a query, however this has not proved necessary so far.

2.3.4 Mutation: Empty and Insert Operations

Next we turn our attention to compiling the empty and insert operations. The empty operation is implemented using a function dempty $\hat{d}$ which creates an empty instance of a decomposition $\hat{d}$. The insert operation is implemented by a function dinsert $\hat{d}\ t\ d$, which inserts a tuple t into a decomposition instance d.

To create an empty instance of a decomposition, the dempty operation simply creates a single instance of the root node of the decomposition graph; since the relation does not contain any tuples, we do not need to create instances of any map edges. The adequacy conditions for decompositions ensure that the root node does not contain any unit decompositions, so it is always possible to represent the empty relation.

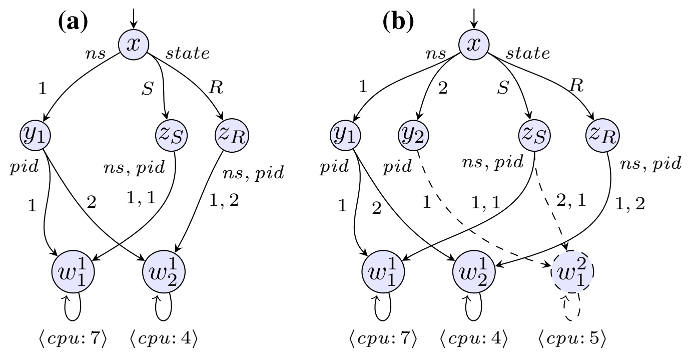


Figure 2.11: Example of insertion and removal. Inserting the tuple $t = \langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle$ into instance (a) produces instance (b); conversely removing tuple t from (b) produces (a). Differences between the instances are shown using dashed lines.

To insert a tuple t into an instance d of a decomposition $\hat{d}$, for each node $v\colon B \triangleright C$ in the decomposition we need to find or create an instance v_s where $s = \pi_B\, t$ in the decomposition instance. For each edge in the decomposition we also need to find or create an instance of the edge connecting the corresponding pair of node instances.

We perform insertion over the nodes of a decomposition in topologically-sorted order. For each node v we locate the existing node instance v_s corresponding to tuple t, if any. If no such v_s exists, we create one, inserting v_s into any data structures that link it to its ancestors. For example, suppose we want to insert the tuple

$$t = \langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle$$

into the decomposition instance shown in Figure 2.11(a). We need to find or create the node instances $x_{\langle\rangle}$, $y_{\langle ns\colon 2\rangle}$, $z_{\langle state\colon S\rangle}$, and $w_{\langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle}$. We consider each in topologically-sorted order. Node $x_{\langle\rangle}$ is the root of the decomposition instance, so we know its location already. Next we lookup the tuple $\langle ns\colon 2\rangle$ in the instance of the map from x to y associated with $x_{\langle\rangle}$; no such tuple exists so we create a new node $y_{\langle ns\colon 2\rangle}$ and insert it into the map. Similarly we look up the tuple $\langle state\colon S\rangle$ in the instance of the map from x to z associated with node $x_{\langle\rangle}$ to discover the existing node instance $z_{\langle state\colon S\rangle}$. Finally, we have a choice; we can either look up tuple $\langle pid\colon 1\rangle$ in the map from y to w or look up the tuple $\langle ns\colon 2, pid\colon 1\rangle$ in the map from z to w; in either case we find that no such tuple exists, hence we must create a new instance of vertex w and insert it into both maps. If tuple t was a duplicate of a tuple already present in the relation then vertex w would have already been present and we would not need to do any further work.

2.3.5 Mutation: Removal and Update Operations

We next consider the remove and update operations. We implement remove using a function dremove $\hat{d}\ s\ d$, which removes tuples matching tuple s from an instance d of decomposition $\hat{d}$. The operation works by

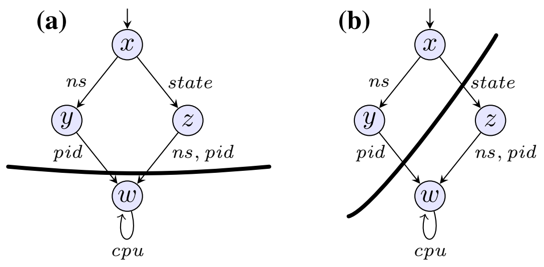


Figure 2.12: Two cuts of a decomposition: (a) the cut for columns $\{ns, pid\}$, and (b) the cut for columns $\{state\}$.

removing any nodes and edges from d that form part of the representation of tuples that only match s.

To implement removal and update we need the notion of a *cut* of a decomposition. Given a tuple t with domain C, a cut of a decomposition $\hat{d}$ is a partition (X, Y) of the nodes of $\hat{d}$ into nodes $y_A \in Y$ that can only be part of the representation of tuples matching t, that is, $\Delta \vdash_{\text{fd}} A \to C$, and nodes $x_B \in X$ that may form part of the representation of tuples that do not match t, that is $\Delta \vdash_{\text{fd}} B \nrightarrow C$. Figure 2.12 shows two possible cuts of the scheduler decomposition for different sets of columns C.

Edges in a decomposition cut (X, Y) may point from X to Y but not vice-versa. This result follows from the adequacy judgment, which ensures that the columns bound in the child of a map edge must functionally determine the columns bound in its parent. The adequacy judgement also guarantees that the cut for a particular decomposition $\hat{d}$ and set of columns C always exists and is unique.

To remove tuples matching a tuple t using a cut (X, Y), we simply break any edges crossing the cut. That is, we remove any references from data structures linking instances of nodes in X to instances of nodes in Y that form part of the representation of tuples that match t. Once all such references are removed, the instances of nodes in Y are unreachable from the root of the decomposition instance and can be deallocated. We can also clean up any map nodes in X that are now devoid of children.

For example, suppose we want to remove all tuples matching the tuple $t = \langle ns\colon 2, pid\colon 1\rangle$ from the decomposition instance shown in Figure 2.11(b). Tuple t has the domain $C = \{ns, pid\}$; Figure 2.12(a) shows the corresponding decomposition cut. Nodes x, y, and z lie above the cut; an instance of node x is always present in every possible relation, instances of node y are specific to a particular namespace but not to any particular process id, and instances of node z are specific to a particular process state but not to any particular process. Node w lies below the cut; each instance of node w forms part of exactly one valuation for the columns C. To perform removal, we break any instances of the edges from instances of nodes y and z to instances of node w which match tuple t; these edges are drawn as dashed lines in Figure 2.11(b). Once the dashed edges/nodes in Figure 2.11 are removed, we have the option to deallocate the map at node y_2 as well. Our implementation deallocates empty maps to minimize space consumption.

To find the edge instances to break we can reuse the query planner. Any query that takes columns C as input and visits each of the edges we want to cut will work. One such plan is

$$\mathsf{qjoin}\big(\mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{left}\big).$$

For some data structures, such as intrusive doubly-linked lists, we can remove edges given the destination node alone. If the edge from z to w uses such a data structure we could use the cheaper plan:

$$\mathsf{qlr}\big(\mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{left}\big).$$

We implement update using a function $\mathsf{dupdate}\ \hat{d}\ s\ u\ d$, which updates tuples matching s using values from u in an instance d of decomposition $\hat{d}$. Semantically, updates are a removal followed by an insertion. In the implementation we can reuse the nodes and edges discarded in the removal in the subsequent insertion—i.e., we can perform the update in place.

We provide only the common case for updates of tuples t matching a tuple pattern s, namely when s is a key for the relation and u does not alter columns appearing in s. Non-key patterns or key-modifying tuple updates may merge multiple tuples, and hence require the implementation to merge nodes. Our restriction guarantees no merging can occur. We can reuse nodes and edges below the cut, and any changes in u that apply to nodes below the cut can be performed in-place.

2.3.6 Soundness of Relational Operations

Next we show that the operations on decompositions faithfully implement the corresponding relational specifications. We show that sequences of relational operations on graph decompositions are sound with respect to their logical counterparts (Theorem 2.5) by induction using initialization and preservation lemmas.

Lemma 2.3 (Decomposition Initialization)**.** For any decomposition $\hat{d}$, if $d = \mathsf{dempty}\ \hat{d}$ then $\cdot, d \models \cdot, \hat{d}$ and $\alpha(d, \cdot) = \emptyset$.

Proof. See Section 2.6.3. □

Lemma 2.4 (Decomposition Preservation)**.** For all $\hat{d}$, Δ, t, d, C, and r where decomposition $\hat{d}$ is adequate ($\cdot; \emptyset \vdash_{a,\Delta} \hat{d}; C$), decomposition instance d is well-formed ($\cdot, d \models \cdot, \hat{d}$), and d represents relation r ($\alpha(d, \cdot) = r$) satisfying FDs Δ ($\Delta \models_{\mathrm{fd}} r$), we have:

(a) If $\mathrm{dom}\, t = C$, $d' = \mathsf{dinsert}\ \hat{d}\ t\ d$, and $\Delta \models_{\mathrm{fd}} r \cup \{t\}$ then $\cdot, d' \models \cdot, \hat{d}$ and $\alpha(d', \cdot) = r \cup \{t\}$.

(b) If $\mathrm{dom}\, t \subseteq C$ and $d' = \mathsf{dremove}\ \hat{d}\ s\ d$ then $\cdot, d' \models \cdot, \hat{d}$ and $\alpha(d', \cdot) = r'$ where $r' = r \setminus \{t \in r \mid t \supseteq s\}$ and $r' \models_{\mathrm{fd}} \Delta$.

(c) Suppose s is a key for r ($\Delta \vdash_{\mathrm{fd}} \mathrm{dom}\, s \to \mathrm{dom}\, C$), the domains of s and u do not intersect ($\mathrm{dom}\, s \cap \mathrm{dom}\, u = \emptyset$), and we have $d' = \mathsf{dupdate}\ \hat{d}\ s\ u\ d$ and $r' = \{\mathsf{if}\ t \supseteq s\ \mathsf{then}\ t \lhd u\ \mathsf{else}\ t \mid t \in r\}$. If $r' \models_{\mathrm{fd}} \Delta$ then $\cdot, d' \models \cdot, \hat{d}$ and $\alpha(d', \cdot) = r'$.

Proof. See Section 2.6.3. □

Theorem 2.5 (Decomposition Soundness)**.** Let C be a set of columns, Δ a set of FDs, and $\hat{d}$ a decomposition adequate for C and Δ. Suppose a sequence of `insert`, `update` and `remove` operators starting from the empty relation produce a relation r, and that each operation satisfies the conditions of Lemma 2.4. Then the corresponding sequence of `dinsert`, `dupdate`, and `dremove` operators given `dempty` $\hat{d}$ as input produce d such that $\cdot, d \models \cdot, \hat{d}$ and $\alpha(d, \cdot) = r$.

Proof. Follows immediately from Lemma 2.3 and Lemma 2.4 by induction. □

2.4 Autotuner

Thus far we have concentrated on the problem of compiling relational operations for a particular decomposition of a relation. However, a programmer may not know, or may not want to invest time in finding the best possible decomposition for a relation. We have therefore constructed an *autotuner* that, given a program written to the relational interface, attempts to infer the best possible decomposition for that program.

The autotuner takes as input a benchmark program that produces as output a cost value (e.g., execution time), together with the name of a relation to optimize. The autotuner then exhaustively constructs all decompositions for that relation up to a given bound on the number of edges, recompiles and runs the benchmark program for each decomposition, and returns a list of decompositions sorted by increasing cost. We do not make any assumptions about the cost metric—any value of interest such as execution time or memory consumption may be used.

2.5 Experiments

We have implemented a compiler, named RELC, that takes as input a relational specification and a decomposition, and emits C++ code implementing the relation. We evaluate our compiler using micro-benchmarks and three real-world systems. The micro-benchmarks (Section 2.5.1) show that different decompositions have dramatically different performance characteristics. Since our compiler generates C++ code, it is easy to incorporate synthesized data representations into existing systems. We apply synthesis to three existing systems (Section 2.5.2), namely a web server, a network flow accounting daemon, and a map viewer, and show that synthesis leads to code that is simultaneously simpler, correct by construction, and comparable in performance to the code it replaces.

We chose C++ because it allows low-level control over memory-layout, has a powerful template system, and has widely-used libraries of data structures from which we can draw. Data structure primitives are implemented as C++ template classes that implement a common associative container API. The set of data structures can easily be extended by writing additional templates and providing the compiler some basic information about the data structure's capabilities. We have implemented a library of data structures that wrap code from the C++ Standard Template Library and the Boost Library [Boost], namely both non-intrusive and intrusive doubly-linked lists (`std::list`, `boost::intrusive::list`), non-intrusive and intrusive binary trees (`std::map`,

```
edges::relation graph_edges;
nodes::relation visited;

// Code to populate graph_edges elided.

stack<int> stk;
stk.push(v0);
while (!stk.empty()) {
  int v = stk.top();
  stk.pop();
  if (!visited.query(nodes::tuple_id(v))) {
    visited.insert(nodes::tuple_id(v));
    edges::query_iterator_src__dst_weight it;
    graph_edges.query(edges::tuple_src(v), it);
    while (!it.finished()) {
      stk.push(it.output.f_dst());
      it.next();
    }
  }
}
```

Figure 2.13: Depth-first search algorithm

`boost::intrusive::set`), hash-tables (`boost::unordered_map`), and vectors (`std::vector`). Since the C++ compiler expands templates, the time and space overheads introduced by the wrappers is minimal.

2.5.1 Microbenchmarks

We implemented a selection of small benchmarks: a benchmark based on our running example of a process scheduler, a cache benchmark based on the real systems discussed in the next section, and a graph benchmark. In this chapter, we focus just on the graph benchmark.

The graph benchmark reads in a directed weighted graph from a text file and measures the times to construct the edge relation, to perform forwards and backwards depth-first searches over the whole graph, and to remove each edge one-by-one. We represent the edges of a directed graph as a relation `edges` with columns $\{src, dst, weight\}$ and a functional dependency $src, dst \to weight$. We represent the set of the graph nodes as a relation `nodes` consisting of a single id column. The RELC compiler emits a C++ module that implements classes `nodes::relation` and `edges::relation` with methods corresponding to each relational operation. A typical client of the relational interface is the algorithm to perform a depth-first search, shown in Figure 2.13. The code makes use of the standard STL `stack` class in addition to an instance of the nodes relation `visited` and an instance of the edges relation `graph_edges`.

To demonstrate the tradeoffs involved in the choice of decomposition, we used the autotuner framework to evaluate three variants of the graph benchmark under different decompositions. We used a single input graph

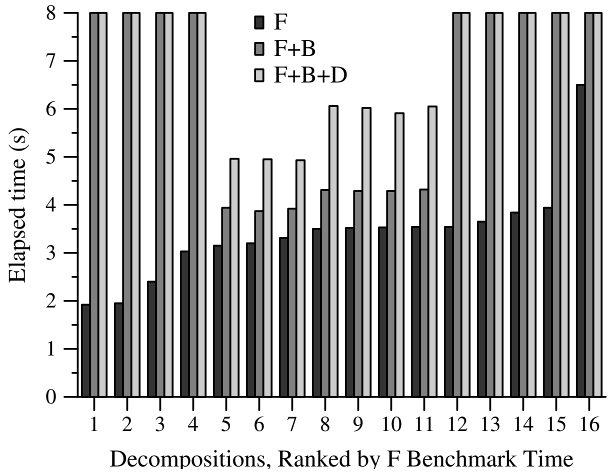


Figure 2.14: Elapsed times for directed graph benchmarks for decompositions up to size 4 with identical input. For each decomposition we show the times to traverse the graph forwards (F), to traverse both forwards and backwards (F+B), and to traverse forwards, backwards and delete each edge (F+B+D). We elide 68 decompositions which did not finish a benchmark within 8 seconds.

representing the road network of the northwestern USA, containing 1207945 nodes and 2840208 edges. We used three variants of the graph benchmark: a forward depth-first search (DFS); a forward DFS and a backward DFS; and finally a forward DFS, a backward DFS, and deletion of all edges one at a time. We measured the elapsed time for each benchmark variant for the 84 decompositions that contain at most 4 map edges (as generated by the autotuner).

Timing results for decompositions that completed within an 8 second time limit are shown in Figure 2.14. Decompositions that are isomorphic up to the choice of data structures for the map edges are counted as a single decomposition; only the best timing result is shown for each set of isomorphic decompositions. There are 68 decompositions not shown that did not complete any of the benchmarks within the time limit. Since the autotuner exhaustively enumerates all possible decompositions, naturally only a few of the resulting decompositions are suitable for the access patterns of this particular benchmark; for example, a decomposition that indexes edges by their weights performs poorly.

Figure 2.15 shows three representative decompositions from those shown in Figure 2.14 with different performance characteristics. Decomposition 1 is the most efficient for forward traversal, however it performs terribly for backward traversal since it takes quadratic time to compute predecessors. Decompositions 5 and 9 are slightly less efficient for forward traversal, but are also efficient for backward traversal, differing only in the sharing of objects between the two halves of the decomposition. The node sharing in decomposition 5 is advantageous for all benchmarks since it requires fewer memory allocations and allows more efficient implementations of insertion and removal; in particular because the lists are intrusive the compiler can find node w using either path and remove it from both paths without requiring any additional lookups.

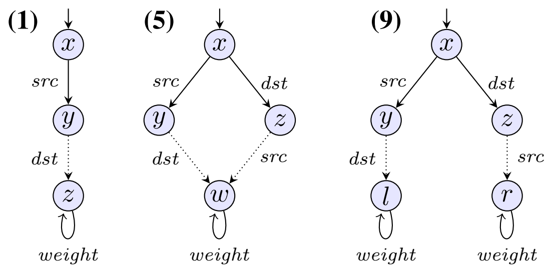


Figure 2.15: Decompositions 1, 5 and 9 from Figure 2.14. Solid edges represent instances of the `map` class from `boost::intrusive`, whereas dotted edges represent instances of `boost::intrusive::list`.

	Original		**Synthesis**	
System	**Everything**	**Module**	**Decomposition**	**Module**
thttpd	7050	402	42	239
Ipcap	2138	899	55	794
ZTopo	5113	1083	39	1048

Table 2.1: Non-comment lines of code for existing system experiments. For each system, we report the size of entire original system and just the source module we altered, together with the size of the altered source module and the mapping file when using synthesis.

2.5.2 Data Representation Synthesis in Existing Systems

To demonstrate the practicality of our approach, we took three existing open-source systems—thttpd, Ipcap, and ZTopo—and replaced core data structures with relations synthesized by RELC. All are publicly-available programs with real-world users.

The thttpd web server is a small and efficient web server implemented in C. We reimplemented the module of thttpd that caches the results of the `mmap()` system call. When thttpd receives a request for a file, it checks the cache to see whether the same file has previously been mapped into memory. If a cache entry exists, it reuses the existing mapping; otherwise it creates a new mapping. If the cache is full then the code traverses through the mappings removing those older than a certain threshold. Other researchers have used thttpd's cache module as a program analysis target [McCloskey et al., 2011].

The IpCap daemon is a TCP/IP network flow accounting system implemented in C. IpCap runs on a network gateway, and counts the number of bytes incoming and outgoing from hosts on the local network, producing a list of network flows for accounting purposes. For each network packet, the daemon looks up the flow in a table, and either creates a new entry or increments the byte counts for an existing entry. The daemon periodically iterates over the collection of flows and outputs the accumulated flow statistics to a log file; flows that have been written to disk are removed from memory. We replaced the core packet statistics data structures with relations implemented using RELC.

ZTopo is a topographic map viewer implemented in C++. A map consists of millions of small image tiles,

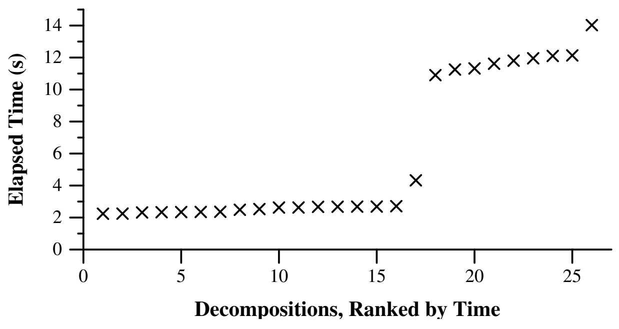


Figure 2.16: Elapsed time for IpCap to log 3×10^5 random packets for 26 decompositions up to size 4 generated by the auto-tuner, ranked by elapsed time. The 58 decompositions not shown did not complete within 30 seconds.

retrieved using HTTP over the internet and reassembled into a seamless image. To minimize network traffic, the viewer maintains memory and disk caches of recently viewed map tiles. When retrieving a tile, ZTopo first attempts to locate it in memory, then on disk, and as a last resort over the network. The tile cache was originally implemented as a hash table, together with a series of linked lists of tiles for each state to enable cache eviction. We replaced the tile cache data structure with a relation implemented using RELC.

Table 2.1 shows non-comment lines of code for each test-case. In each case the synthesized code is comparable to or shorter than the original code in size. Both the thttpd and ipcap benchmarks originally used open-coded C data structures, accounting for a large fraction of the decrease in line count. ZTopo originally used C++ STL and Boost data structures, so the synthesized abstraction does not greatly alter the line count. The ZTopo benchmark originally contained a series of fairly subtle dynamic assertions that verified that the two different representations of a tile's state were in agreement; in the synthesized version the compiler automatically guarantees these invariants, so the assertions were removed.

For each system, the relational and non-relational versions had equivalent performance. If the choice of data representation is good enough, data structure manipulations are not the limiting factor for these particular systems. The assumption that the implementations are good enough is important, however; the auto-tuner considered plausible data representations that would have resulted in significant slow-downs, but found alternatives where the data manipulation was no longer the bottleneck. For example we used the autotuner on the Ipcap benchmark to generate all decompositions up to size 4; Figure 2.16 shows the elapsed time for each decomposition on an identical random distribution of input packets. The best decomposition is a binary-tree mapping local hosts to hash-tables of foreign hosts, which performs approximately $5\times$ faster than the decomposition ranked 18th, in which the data structures are identical but local and foreign hosts are transposed. For this input distribution the best decomposition performs identically to the original hand-coded implementation to within the margin of measurement error.

Our experiments show that different choices of decomposition lead to significant changes in performance (Section 2.5.1), and that the best performance is comparable to existing hand-written implementations (Section 2.5.2). The resulting code is concise (Sections 2.5.1 and 2.5.2), and the soundness of the compiler

(Theorem 2.5) guarantees that the resulting data structures are correct by construction.

2.6 Proofs of Theorems

To minimize notational overhead, in proofs we work with decompositions and instances encoded via environments rather than let-bindings. We represent a decomposition $\hat{d}$ as a pair of the environment $\hat{\Gamma}$ and a root variable z (as constructed by the well-formedness rules). If a decomposition is adequate then there exists an adequacy type environment Σ (as constructed by the adequacy inference rules) such that for each $v \in \operatorname{dom}\Gamma$ we have $\Sigma(v) = A \triangleright B$ and $\Sigma; A \vdash_{a,\Delta} \hat{\Gamma}(v); B$ holds; further we must have $\Sigma(z) = \emptyset \triangleright C$ for some C; we say the decomposition is adequate for C and Δ. We preserve the order of the original let-bindings via a total variable order $\prec$, which is a reverse topological sort of the decomposition graph. As in the case of decompositions, we represent an instance as a pair of an environment Γ and a root variable $v_{\langle\rangle}$.

2.6.1 Soundness of Adequacy

We prove a strengthened version of Lemma 2.1 by induction; Lemma 2.1 then follows as a special case.

Lemma 2.6. Suppose we have an adequate decomposition $\hat{d}$, represented by $\hat{\Gamma}$, v, and Σ, where $\Sigma(z) = \emptyset \triangleright Z$. Then for any relation r such that $\operatorname{dom} r = Z$ and $r \models_{\text{fd}} \Delta$, there exists an environment Γ such that for all v where $\hat{\Gamma}(v) = \hat{p}$ and $\Sigma(v) = X \triangleright Y$ and any $t \in \pi_X\, r$ we have $\Gamma(v_t) = p_t$ such that $\Gamma, p_t \models \hat{\Gamma}, \hat{p}$ and $\alpha(p_t, \Gamma) = \pi_Y\, \sigma_t\, r$.

Proof. By induction on the let binding order. The base case is trivial. Suppose there exists v such that the hypothesis holds for all $w \prec v$. We must have $\Sigma(v) = X \triangleright Y$ and $\Sigma; X \vdash_{a,\Delta} \hat{\Gamma}(v); Y$ since the decomposition is adequate. We now prove the existence of each p_t instance by another induction on the derivation of the adequacy judgement.

If the outermost rule is (AUNIT) we must have $\hat{p} = C$ and $\Delta \vdash_{\text{fd}} X \to Y$. For any $t \in \pi_X\, r$ set $\Gamma(v_t) = p_t = \pi_Y\, \sigma_t\, r$; we are guaranteed the right-hand side of the expression is a single tuple by the functional dependency. We have $\Gamma, p_t \models \hat{\Gamma}, Y$ immediately by WFUNIT, and we have $\alpha(p_t, \Gamma) = \pi_Y\, \sigma_t\, r$ by the definition of the abstraction function.

If the outermost rule is (AMAP), then we have $\hat{p} = C \mapsto v'$, and there exists A, B, C, D such that $X = B$, $Y = C \cup D$, $\Sigma(v') = A \triangleright D$, $\Delta \vdash_{\text{fd}} B \cup C \to A$ and $A \supseteq B \cup C$. For any $t \in \pi_B\, r$ we set

$$\Gamma(v_t) = p_t = \{t \mapsto v'_{t'} \mid t \in \pi_C\, \sigma_u\, r,\ \pi_A\, \sigma_t\, \sigma_u\, r = \{t'\}\}.$$

Well-formedness $\Gamma, p_t \models \hat{\Gamma}, \hat{p}$ follows from (WFMAP), and the induction hypothesis. For $\alpha(p_t, \Gamma)$, note that

$\pi_A\, \sigma_t\, \sigma_u\, r = \{t'\}$ since $r \models_{\text{fd}} \Delta$ and $\Delta \vdash_{\text{fd}} B \cup C \to A$ by assumption. Hence we have:

$$\begin{aligned}
\alpha(p_t, \Gamma) &= \bigcup_{t \in \pi_C\, \sigma_u\, r} \{t\} \bowtie \alpha(v_{t'}, \Gamma) \\
&= \bigcup_{t \in \pi_C\, \sigma_u\, r} \{t\} \bowtie \pi_D (r \bowtie \pi_A\, \sigma_t\, \sigma_u\, r) \\
&= \bigcup_{t \in \pi_C\, \sigma_u\, r} \{t\} \bowtie \pi_D (r \bowtie \pi_A\, \sigma_t\, \sigma_u\, r) \\
&= \bigcup_{t \in \pi_C\, \sigma_u\, r} \{t\} \bowtie \pi_D\, \sigma_t\, \sigma_u (r \bowtie \pi_A\, r) \\
&= \bigcup_{t \in \pi_C\, \sigma_u\, r} \{t\} \bowtie \pi_D\, \sigma_t\, \sigma_u\, r = \pi_{C \cup D}\, \sigma_u\, r
\end{aligned}$$

If the outermost rule is (AJOIN), then we have $X = A$, $Y = B \cup C$, $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$, and by the IH we have p_1 and p_2 such that $\Gamma, p_1 \models \hat{p}_1,$, $\Gamma, p_2 \models \hat{p}_2,$, $\alpha(p_1, \Gamma) = \pi_B\, \sigma_u\, r$, and $\alpha(p_2, \Gamma) = \pi_C\, \sigma_u\, r$. Observe $\Gamma, p_1 \bowtie p_2 \models \hat{p}_1 \bowtie \hat{p}_2$, follows from rule (WFJOIN), and we have

$$\alpha(p_1 \bowtie p_2, \Gamma) = \pi_B\, \sigma_u\, r \bowtie \pi_C\, \sigma_u\, r = \pi_{(B \cup C)}\, \sigma_u\, r$$

by the definition of the abstraction function and the functional dependency hypothesis of the rule. □

We now have:

Proof of Lemma 2.1. Follows from Lemma 2.6. □

2.6.2 Soundness of Queries

To prove Lemma 2.2 we define a variant dqexec formally. We then show a strengthened version of Lemma 2.2 by induction, and the result follows.

We define dqexec $q\ \Gamma\ p\ s$:

dqexec qunit $\Gamma\ t\ s =$ if $s \sim t$ then $\{t\}$ else $\emptyset$
dqexec qscan(q) Γ $\{t \mapsto v_{t'}\}_{t \in T}$ $s = \bigcup \{t \bowtie$ dqexec $q\ \Gamma\ \Gamma(v_{t'})\ s' \mid t \in T,\ s' \in s \bowtie t\}$
dqexec qlookup(q) Γ $\{t \mapsto v_{t'}\}_{t \in T}$ $s = \bigcup \{t \bowtie$ dqexec $q\ \Gamma\ \Gamma(v_{t'})\ s \mid t \in T,\ s = t\}$
dqexec qlr$(q_1, q_2,$ left$)$ Γ $(p_1 \bowtie p_2)$ $s =$ dqexec $q_1\ \Gamma\ p_1\ s$
dqexec qlr$(q_2, q_1,$ right$)$ Γ $(p_2 \bowtie p_1)$ $s =$ dqexec $q_2\ \Gamma\ p_2\ s$
dqexec qjoin$(q_1, q_1,$ left$)$ Γ $(p_1 \bowtie p_2)$ $s =$ dqexec qjoin$(q_2, q_1,$ right$)$ Γ $(p_2 \bowtie p_1)$ $s =$
$\bigcup \{($dqexec $q_2\ \Gamma\ p_2\ s') \bowtie t \mid\ t \in$ dqexec $q_1\ \Gamma\ p_1\ s,\ s' \in s \bowtie t\}$

Note that we write a pair of an environment Γ and p in place of a decomposition d.

We write Δ / C to denote the set of FDs Δ' where instances of columns C are constant and hence have been removed from both the right-hand and left-hand side of all FDs in Δ.

Lemma 2.7. Suppose we have Γ, p, $\hat{\Gamma}$, $\hat{p}$, and Σ, such that $\hat{p}$ is adequate ($\Sigma; F \vdash_{a,\Delta} p; G$ for some F and G), p is well-formed ($\Gamma, p \models \hat{\Gamma}, \hat{p}$), and the denotation of p satisfies FDs Δ/A ($\alpha(p,\Gamma) \models_{\text{fd}} \Delta/A$). If q is a valid query ($\hat{\Gamma}, \hat{p}, A \vdash_{q,\Delta} q, B$) and s a tuple with $\operatorname{dom} s = A$ we have

$$\mathsf{dqexec}\ q\ \Gamma\ p\ s = \pi_B\ \sigma_s\ \alpha(p,\Gamma).$$

Proof. By induction on the derivation of $\hat{\Gamma}, \hat{p}, C \vdash_{q,\Delta} q, D$.

If the outermost rule is (QUNIT), then we have $q = \mathsf{qunit}$, $\hat{p} = B$, and we must have $p = t$ where $\operatorname{dom} t = B$ by well-formedness. Then

$$\begin{aligned}&\mathsf{dqexec}\ \mathsf{qunit}\ \Gamma\ t\ s\\ &= \mathsf{if}\ s \sim t\ \mathsf{then}\ \{t\}\ \mathsf{else}\ \emptyset\\ &= \pi_B\ \sigma_s\ t\\ &= \pi_B\ \sigma_s\ \alpha(t,\Gamma).\end{aligned}$$

If the outermost rule is (QSCAN) then we have $q = \mathsf{qscan}(q')$, $\hat{p} = C \mapsto v$, and we must have $d = \{t \mapsto v_{t'}\}_{t \in T}$ by well-formedness. Now

$$\begin{aligned}&\mathsf{dqexec}\ \mathsf{qscan}(q')\ \Gamma\ \{t \mapsto v_{t'}\}_{t\in T}\ s\\ &= \bigcup\{t \bowtie \mathsf{dqexec}\ q'\ \Gamma\ \Gamma(v_{t'})\ s' \mid t \in T,\ s' \in s \bowtie t\}\\ &= \bigcup\{t \bowtie \pi_B\ \sigma_{s'}\ \alpha(v_{t'},\Gamma) \mid t \in T,\ s' \in s \bowtie t\}\end{aligned}$$

follows by the induction hypothesis

$$\begin{aligned}&= \pi_{B\cup C}\ \sigma_s \bigcup\{t \bowtie \alpha(v_{t'},\Gamma) \mid t \in T\}\\ &= \pi_{B\cup C}\ \sigma_s\ \alpha(\{t \mapsto v_{t'}\}_{t\in T},\Gamma).\end{aligned}$$

Rule (QLOOKUP) is a special case of rule (QSCAN).

If the outermost rule is (QLR) say we have $q = \mathsf{qlr}(q_1, \mathsf{left})$, $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$ and $p = p_1 \bowtie p_2$. Then

$$\begin{aligned}&\mathsf{dqexec}\ \mathsf{qlr}(q_1,\mathsf{left})\ \Gamma\ p_1 \bowtie p_2\ s\\ &= \mathsf{dqexec}\ q_1\ \Gamma\ p_1\ s\\ &= \pi_B\ \sigma_s\ \alpha(p_1,\Gamma)\end{aligned}$$

follows by the induction hypothesis, and

$$= \pi_B\, \sigma_s(\alpha(p_1, \Gamma) \bowtie \alpha(p_2, \Gamma))$$

follows by well-formedness, finally

$$= \pi_B\, \sigma_s\, \alpha(p_1 \bowtie p_2, \Gamma).$$

The right case follows by symmetry.

If the outermost rule is (QJOIN) assume without loss of generality we have $q = \mathsf{qjoin}(q_1, q_2, \mathsf{left})$, $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$ and $p = p_1 \bowtie p_2$; the qright case follows by symmetry. Now,

$$\begin{aligned}&\mathsf{dqexec}\ \mathsf{qjoin}(q_1, q_2, \mathsf{left})\ \Gamma\ p_1 \bowtie p_2\ s\\ &= \bigcup\{(\mathsf{dqexec}\ q_2\ \Gamma\ p_2\ s') \bowtie t \mid t \in \mathsf{dqexec}\ q_1\ \Gamma\ p_1\ s,\ s' \in s \bowtie t\}\end{aligned}$$

and by the induction hypothesis

$$= \bigcup\{(\pi_{B_2}\, \sigma_{s'}\, \alpha(p_2, \Gamma)) \bowtie t \mid t \in \pi_{B_1}\, \sigma_s\, \alpha(p_1, \Gamma),\ s' \in s \bowtie t\}$$

By adequacy and the hypotheses we have

$$\begin{aligned}&= (\pi_{B_2}\, \sigma_s\, \alpha(p_2, \Gamma)) \bowtie (\pi_{B_1}\, \sigma_s\, \alpha(p_1, \Gamma))\\ &= \pi_{B_1 \cup B_2}\, \sigma_s(\alpha(p_2, \Gamma) \bowtie \alpha(p_1, \Gamma))\\ &= \pi_{B_1 \cup B_2}\, \sigma_s\, \alpha(p_1 \bowtie p_2, \Gamma).\end{aligned}$$

□

Proof of Lemma 2.2. Follows from Lemma 2.7. □

2.6.3 Soundness of Mutations

Empty

Formally we define dempty on let-free decompositions.

Suppose we have an adequate decomposition $\hat{d}$ represented by $\hat{\Gamma}$, z, Σ. Then we define

$$\mathsf{dempty}_{\hat{\Gamma}, z}\ () = \{z_{\langle\rangle} \mapsto \mathsf{emp}\ \hat{\Gamma}(z)\},$$

where

$$\begin{aligned}
&\mathsf{emp}\ (C \mapsto v) = \{\}\\
&\mathsf{emp}\ \hat{p}_1 \bowtie \hat{p}_2 = (\mathsf{emp}\ \hat{p}_1 \bowtie \mathsf{emp}\ \hat{p}_2)
\end{aligned}$$

There is no case for unit decompositions; a unit decomposition cannot represent an empty relation and the adequacy conditions guarantee that none is necessary.

Proof of Lemma 2.3. Follows from the definition of dempty and the definitions of well-formedness and the abstraction. □

Insert

To define insertion we define auxiliary functions $\mathsf{singleton}_\Sigma$, which creates a singleton primitive instance, and add_Σ, which inserts a tuple into an existing primitive instance. Each function operates over a decomposition with typing Σ. We define:

$$\begin{aligned}
&\mathsf{singleton}_\Sigma\ C\ t = \pi_C\ t\\
&\mathsf{singleton}_\Sigma\ (C \mapsto v)\ t = \{\pi_C\ t \mapsto v_u\} \text{ where } v\colon A \triangleright B \in \Sigma \text{ and } u = \pi_A\ t\\
&\mathsf{singleton}_\Sigma\ (\hat{p}_1 \bowtie \hat{p}_2)\ t = \mathsf{singleton}_\Sigma\ \hat{p}_1\ t \bowtie \mathsf{singleton}_\Sigma\ \hat{p}_2\ t
\end{aligned}$$

and

$$\begin{aligned}
&\mathsf{add}_\Sigma\ C\ t\ u = t \text{ when } \pi_C\ u = t\\
&\mathsf{add}_\Sigma\ (C \mapsto v)\ \{t \mapsto v_{t'}\}_{t \in T}\ u = \{t \mapsto v_{t'}\}_{t \in T} \cup \{\pi_C\ u \mapsto v_{u'}\} \text{ where } v\colon A \triangleright B \in \Sigma \text{ and } u' = \pi_A\ u\\
&\mathsf{add}_\Sigma\ (\hat{p}_1 \bowtie \hat{p}_2)\ (p_1 \bowtie p_2)\ t = \mathsf{add}_\Sigma\ \hat{p}_1\ p_1\ t \bowtie \mathsf{add}_\Sigma\ \hat{p}_2\ p_2\ t
\end{aligned}$$

Now we can define insertion $\mathsf{dinsert}_{\hat{\Gamma},z,\Sigma}\ u\ \Gamma = \Gamma'$, where for any $(v\colon A \triangleright B) \in \Sigma$ we define

$$\Gamma'(v_t) = \begin{cases} \Gamma(v_t) & \text{if } v_t \in \operatorname{dom}\Gamma,\ t \neq \pi_A\ u\\ \mathsf{add}_\Sigma\ \hat{\Gamma}(v)\ v_t\ u & \text{if } v_t \in \operatorname{dom}\Gamma,\ t = \pi_A\ u\\ \mathsf{singleton}_\Sigma\ \hat{\Gamma}(v)\ u & \text{if } v_t \notin \operatorname{dom}\Gamma,\ t = \pi_A\ u\\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 2.8. Suppose we have a decomposition $\hat{d}$ adequate for Z and Δ, represented by $\hat{\Gamma}$, z, and Σ, and let Γ be a well-formed instance. Let $r = \alpha(z_{\langle\rangle}, \Gamma)$, let u be a tuple with $\operatorname{dom} u = Z$, let $r' = r \cup \{u\}$, and suppose $r' \models_{\text{fd}} \Delta$. Suppose $\mathsf{dinsert}_{\hat{\Gamma},z,\Sigma}\ u\ \Gamma = \Gamma'$. For all $(v\colon A \triangleright B) \in \Sigma$ and all $v_t \in \Gamma'$ let $\hat{p} = \hat{\Gamma}(v)$, let $p_t = \Gamma(v_t)$

if defined, and let $p'_t = \Gamma'(v_t)$. Then $\Gamma', p'_t \models \hat{\Gamma}, \hat{p}$ and

$$\alpha(p'_t, \Gamma') = \begin{cases} \alpha(p_t, \Gamma) \cup \pi_{\operatorname{dom} \hat{p}}\, \sigma_t\{u\} & \text{if } v_t \in \operatorname{dom} \Gamma \\ \pi_{\operatorname{dom} \hat{p}}\, \sigma_t\{u\} & \text{otherwise.} \end{cases}$$

Proof. By induction on let-binding order. The base case is trivial. Take some $(v\colon A \rhd B) \in \Sigma$ and suppose the result holds for all w where $w \prec v$. There are three cases:

If $v_t \in \operatorname{dom} \Gamma$ and $t \neq \pi_A\, u$, the adequacy result ensures that all children $v'_{t'}$ of node v_t have $t' \supseteq t$, and are not modified by the insertion. The well-formedness and abstraction results follow immediately by induction on the structure of $\Gamma(v_t)$.

If $v_t \in \operatorname{dom} \Gamma$ and $t = \pi_A\, u$ then we have $p_t = \mathsf{add}_\Sigma\, \hat{\Gamma}(v)\, \Gamma(v_t)\, u$. We show the result by induction on the structure of $\hat{p}$:

- If $\hat{p} = C$ then we have $p_t = u'$ where $\Delta \vdash_{\text{fd}} \operatorname{dom} t \to C$ and $r' \models_{\text{fd}} \Delta$, so we must have $\pi_C\, u = u' = p'_t$. Well-formedness and abstraction then follow from the definitions.
- If $\hat{p} = C \mapsto v'$ then we have $p_t = \{s \mapsto v'_{s'}\}_{s \in S}$ and $p'_t = \{s \mapsto v'_{s'}\}_{s \in S} \cup \{\pi_C\, u \mapsto v'_{u'}\}$ where $\Sigma(v') = D \rhd E$ and $u' = \pi_D\, u$. The result then follows from the (outer) induction hypothesis and the definitions of well-formedness and abstraction.
- If $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$ then we have $p_t = p_1 \bowtie p_2$ and the result follows easily by the induction hypothesis and the definitions.

If $v_t \notin \operatorname{dom} \Gamma$ and $t = \pi_A\, u$ then we have $\Gamma'(v_t) = \mathsf{singleton}_\Sigma\, \hat{\Gamma}(v)\, u$. We show the result by induction on the structure of $\hat{p}$, where we take $\alpha(p_t, \Gamma)$ to be empty in

- If $\hat{p} = C$ the result follows from the definitions.
- If $\hat{p} = C \mapsto v'$ the result follows from the outer induction hypothesis and the definitions.
- If $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$ the result follows from the inner induction hypothesis and the definitions.

$\square$

Remove

To define removal of tuples matching s where $\operatorname{dom} s = Z$ in a decomposition, recall a decomposition cut (X, Y) is a partition of the nodes $(v\colon A \rhd B) \in \Sigma$, that is, we have $\Delta \vdash_{\text{fd}} A \nrightarrow Z$ for all $v \in X$ and $\Delta \vdash_{\text{fd}} A \to Z$ for all $v \in Y$.

We define an auxiliary function $\mathsf{del}\ \hat{p}\ p\ V$:

$$\begin{aligned}
&\mathsf{del}\ C\ t\ V = t \\
&\mathsf{del}\ (C \mapsto v)\ \{t \mapsto v_{t'}\}_{t \in T}\ V = \{t \mapsto v_{t'} \mid t \in T,\ v_{t'} \notin V\} \\
&\mathsf{del}\ (\hat{p}_1 \bowtie \hat{p}_2)\ (p_1 \bowtie p_2)\ V = \mathsf{del}\ \hat{p}_1\ p_1\ V \bowtie \mathsf{del}_\Sigma\ \hat{p}_2\ p_2\ V
\end{aligned}$$

Suppose we have an instance defined by $(\Gamma, w_{\langle\rangle})$, where $\alpha(w_{\langle\rangle}, \Gamma) = r$. Let s be a tuple. We define $r' = \{t \in r \mid t \not\supseteq s\}$ and

$$V = \{v_t \mid v_t \in \operatorname{dom}\Gamma,\ v \in Y,\ \Sigma(v) = A \triangleright B,\ t \notin \pi_A\, \sigma_s\, r'\}.$$

We then define $\mathsf{dremove}_{\hat{\Gamma},z,\Sigma}\ s\ \Gamma = \Gamma'$, where for any $(v\colon A \triangleright B) \in \Sigma$

$$\Gamma'(v_t) = \begin{cases} \mathsf{del}\ \hat{\Gamma}(v)\ \Gamma(v_t)\ V & \text{if } v \in X,\ v_t \in \operatorname{dom}\Gamma \\ \Gamma(v_t) & \text{if } v \in Y,\ v_t \in \operatorname{dom}\Gamma \setminus V \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the definition of removal given here the set of nodes to prune V is computed in advance, and then pruned. Since we want a constant-space implementation, in practice we interleave the computation of V with pruning.

Lemma 2.9. Suppose we have an adequate decomposition $\hat{d}$, represented by $\hat{\Gamma}$, v, and Σ, and suppose that $\mathsf{dremove}_{\hat{\Gamma},z,\Sigma}\ u\ \Gamma = \Gamma'$. Then for all $(v\colon A \triangleright B) \in \Sigma$ and all $v_t \in \Gamma'$, let $\hat{p} = \hat{\Gamma}(v)$, let $p_t = \Gamma(v_t)$, and let $p'_t = \Gamma'(v_t)$. We have $\Gamma, p_t \models \hat{\Gamma}, \hat{p}$ and

$$\alpha(p'_t, \Gamma') = \alpha(p_t, \Gamma) \setminus \pi_{\operatorname{dom}\hat{p}}\, \sigma_t \{s \in r \mid s \supseteq u\}.$$

Proof. By induction on let-binding order. The base case is trivial. Take some $(v\colon A \triangleright B) \in \Sigma$ and suppose the result holds for all w where $w \prec v$. There are two cases: If $v \in Y$ and $v_t \in \operatorname{dom}\Gamma \setminus V$ then by the adequacy judgement all descendants $v'_{t'}$ of v_t have $t' \supseteq t$ and hence cannot be in V. The well-formedness and abstraction results are immediate.

If $v \in X$ and $v_t \in \operatorname{dom}\Gamma$ then the result follows by induction on the structure of $\hat{p}$:

- If $\hat{p} = C$ the result is immediate.
- If $\hat{p} = C \mapsto v'$ the result follows from the outer induction hypothesis and the definitions.
- If $\hat{p} = \hat{p}_1 \bowtie \hat{p}_2$ then the result follows from the inner induction hypothesis and the definitions.

□

Update

Semantically update is a removal followed by an insertion and for the purposes of the proof we define it that way. In practice we implement optimizations that reuse the node structure that would otherwise be discarded during removal; all we need to do is make any changes in u to the nodes below the cut before reusing the structure. In addition since we are reusing node structure we optimize away cutting an edge followed by recreating the same edge (i.e. nothing changed).

We define

$$\mathsf{dupdate}_{\hat{\Gamma},z,\Sigma}\; t\; u\; \Gamma = \mathsf{dinsert}_{\Sigma,\hat{\Gamma}}\; (t' \lhd u)\; (\mathsf{dremove}_{\Sigma,\hat{\Gamma}}\; t\; \Gamma)$$

where there exists $t' = \sigma_t\, \alpha(z_{\langle\rangle}, \Gamma)$; otherwise we define

$$\mathsf{dupdate}_{\hat{\Gamma},z,\Sigma}\; t\; u\; \Gamma = \Gamma.$$

Lemma 2.10. Suppose we have an decomposition $\hat{d}$ adequate for Z and Δ, represented by $\hat{\Gamma}$, v, and Σ. Suppose we have an instance Γ. Let $r = \alpha(z_{\langle\rangle}, \Gamma)$, and suppose $r \models_{\mathrm{fd}} \Delta$. If t is a key for the relation (i.e. $\Delta \vdash_{\mathrm{fd}} \operatorname{dom} t \to Z$), the domains of t and u do not intersect (i.e. $\operatorname{dom} t \cap \operatorname{dom} u = \emptyset$), and we have $\Gamma' = \mathsf{dupdate}_{\hat{\Gamma},z,\Sigma}\; t\; u\; \Gamma$, we have

$$r' = \{\mathsf{if}\; s \supseteq t\; \mathsf{then}\; s \lhd u\; \mathsf{else}\; s \mid s \in r\},$$

and $r' \models_{\mathrm{fd}} \Delta$. Then $\Gamma', z_{\langle\rangle} \models \hat{\Gamma}, z$ and $\alpha(z_{\langle\rangle}, \Gamma') = r'$.

Proof. Follows from Lemma 2.8 and Lemma 2.9. □

Preservation

Finally we have the preservation lemma.

Proof of Lemma 2.4. Follows from Lemma 2.8, Lemma 2.9, and Lemma 2.10. □

Chapter 3

Lock Placements

In Chapter 2, we introduced data representation synthesis, a technique for synthesizing low-level representations of data from a high-level relational description. Next, we consider the problem of extending our synthesis approach to generate data representations that support concurrent access from multiple threads. Before we can do so, in this chapter we first develop the theory of *lock placements*, which describe the mapping between locks and the data they protect, and which form an important building block for concurrent data representation synthesis (Chapter 4).

Most concurrent software uses *locks* as a primitive for ensuring mutual exclusion between threads. While it is correct to say that the key characteristic of a lock is that it may be held by only one thread at a time, such a description fails to capture the higher-level purposes for which programmers use locks. Universally, locks are used to protect data, guaranteeing that only one thread operates on particular parts of the store at a time. The association between locks and the data they protect is, however, implicit, and in the presence of mutable data structures it is not even clear how to describe the relationship between a possibly changing set of locks and the changing heap the locks protect.

This chapter investigates what it means for locks to protect data. So far as we are aware, there are no proposals in the literature for even stating the relationship between locks and the data they protect in a way that captures the range of ways in which locks are used in practice. In particular, we are interested in explaining *speculative locks* and the common case in which updates to the heap change which data locks protect. We believe ours is the first proposal to address these issues.

To explain our results, we begin with a slightly informal, simple, obviously correct, but impractical locking protocol. We assume the heap consists of a graph of *objects* (nodes), each of which has a set of *fields* (edges) that point to other objects. We also assume that concurrent operations are expressed as transactions that execute atomically (e.g., atomic blocks). Every heap edge has a *logical lock*. Each transaction t must obey a standard two-phase locking protocol:

1. Acquire all logical locks of every edge read or written by t.

2. Perform the reads and writes of t.

3. Release all of t's logical locks.

It is a classic result [Eswaran et al., 1976] that any interleaving of such transactions is *serializable* (equivalent to some sequential schedule of the transactions). However in practice acquiring a separate lock for every field of every object touched by a transaction is exorbitantly expensive. Thus, practical locking protocols use fewer locks. For example, a tree data structure might have a single lock at the root node, or a hash table may have one lock per hash bucket, with no locks on the contents of the buckets.

The key insight is that in such designs the programmer has made an optimization: many logical locks are represented by a single *physical lock*. We can still think of a transaction as acquiring all of the logical locks required, but now instead of acquiring the lock on the actual edge e it must instead acquire the physical lock $\psi(e)$ assigned to the edge by a *lock placement* ψ, which is a mapping from logical locks to physical locks. So, for example, in the tree case $\psi(e) = \rho$, where ρ is the tree's root, for every edge in the tree. For the hash table, $\psi(e) = l_i$, where the i-th bucket has an associated physical lock l_i for every field e in the i-th bucket. When multiple logical locks are represented by a single physical lock, transactions need only acquire the physical lock to obtain access to multiple heap locations.

Lock placements capture several common idioms for programming with locks:

- Locking at different granularities corresponds to altering the granularity of the lock placement. For example, each element of a tree may have its own lock, or every element of the tree may be associated with a single lock at the root. The lock placement makes explicit which locations are guarded by the same lock, and where that lock is placed.

- It is sometimes beneficial to place the lock guarding an object o in a field of o itself, which means that o cannot be locked without first accessing o in an unlocked state. Such *speculative placements* of transaction metadata have previously been proposed in the context of an optimistic software transactional memory [Bronson et al., 2010a]. Lock placements can describe speculatively-placed locking, and our approach allowing us to reason about the serializability of transactions that use it.

- Which locks guard which fields often changes over time. As a simple example, consider a heap in which all `nil` fields are guarded by a global lock, and all non-`nil` fields are guarded by a speculative lock in the object the field points to. When a `nil` field is assigned an object the global lock is *split* and no longer guards the field, and when a pointer field is assigned `nil` that field is *merged* into the global lock. Lock placements can depend on the state of the heap and so naturally capture lock splitting and merging.

We develop our results incrementally, beginning with flat "heaps" that are just a set of global variables with no pointers (Section 3.1). In this simple setting we formalize the key notions of lock placements and *stability*, we give a proof system for showing that transaction traces are *well-locked*, and we prove that well-locked transactions are serializable. We then consider heaps that are mutable trees (Section 3.2), where the main complication is that logical locks are now named by heap paths, which may be updated concurrently.

Symbol	Meaning
$m \in \mathcal{M}$	memory locations
$l \in \mathcal{L},\ L \subseteq \mathcal{L}$	locks, lock sets
$b ::= \mathsf{F} \mid \mathsf{T}$	booleans
$\omega ::= m \mapsto b$	heap assertions
$\psi \subseteq \mathcal{M} \rightarrow 2^{L \times \Phi}$	lock placements
$\Phi \ni \phi ::= b \mid \omega \mid \phi \vee \phi \mid \phi \wedge \phi$	guards
$t ::= \mathsf{read}(m) = b \mid \mathsf{observe}(m) = b \mid \mathsf{write}(m, b) \mid \mathsf{lock}\ l \mid \mathsf{unlock}\ l$	transaction ops.

Figure 3.1: Locations, Lock Placements, Transaction Operations

Finally, we show how to apply our approach to decomposition heaps (Section 3.3). Formally, we show how to apply lock placements for a class of mutable DAG heaps with bounded in-degree which are a relaxation of decompositions; sharing complicates lock placements as there may be multiple access paths to an object.

Because our focus is on formalizing lock placements and their correctness conditions, in this chapter we do not consider liveness properties, such as deadlock, or optimizations, such as early release, since these issues are orthogonal to the ones we explore. The standard techniques for ensuring deadlock-freedom apply, including both static techniques (imposing a total ordering on locks) and dynamic techniques (using a contention manager to resolve deadlocks at runtime).

3.1 Flat Maps

We first consider a simple class of heaps defined over a fixed set of *memory locations* $\mathcal{M}$. A *flat map heap* is a set of mappings $\{m \mapsto b\}_{m \in \mathcal{M}}$ from each location $m \in \mathcal{M}$ to a boolean value b. Let $\mathcal{L}$ be a fixed set of *physical locks*; in this section we assume that memory locations and locks are disjoint. For ease of exposition we consider only exclusive locks — that is, if a transaction holds a lock then no other transaction may acquire concurrent access to the same lock.

A common correctness criterion for concurrent transactions is *serializability*. Informally a concurrent execution of a set of transactions is serializable if the reads and writes that transactions make to the heap are equivalent to the reads and writes in some serial schedule of the same set of transactions. By showing that concurrent executions are serializable we can reason about programs as if only one transaction executes at a time.

A *transaction* $\mathbf{T}$ is a sequence $t^1 t^2 \ldots$ of the atomic *transaction operations* given in Figure 3.1: a possibly unstable read of location m yielding b ($\mathsf{read}(m) = b$), a logical observation of location m yielding b ($\mathsf{observe}(m) = b$), a write of b to location m ($\mathsf{write}(m, b)$), a lock of a physical lock l ($\mathsf{lock}\ l$), or an unlock of physical lock l ($\mathsf{unlock}\ l$). With the exception of the read and observe operations the concrete semantics of transaction operations are standard. We assume the execution of transaction operations is sequentially consistent.

(CLock)
$$\frac{l \notin L}{h, L, \mathsf{lock}\ \ l \rightarrow h, L \cup \{l\}}$$

(CUnlock)
$$\frac{l \in L}{h, L, \mathsf{unlock}\ \ l \rightarrow h, L \setminus \{l\}}$$

(CRead)
$$\frac{(m \mapsto b) \in h}{h, L, \mathsf{read}(m) = b \rightarrow h, L}$$

(CWrite)
$$\frac{h' = h[m \mapsto b]}{h, L, \mathsf{write}(m, b) \rightarrow h', L}$$

(CObserve)
$$\frac{}{h, L, \mathsf{observe}(m) = b \rightarrow h, L}$$

Figure 3.2: Concrete semantics of flat heap transactions. $h, L, t \rightarrow h', L'$ holds if executing operation t on heap h with locks held by any transaction L yields an updated heap h' with new locks held L'.

The transaction language distinguishes between between high-level observe operations, which are observations of the state of memory that affect the outcome of a transaction and for which the locking protocol must ensure serializability, and low-level read operations, which do not directly affect the outcome of a transaction and need not be serializable. A transaction may freely perform a read operation on any memory location at any time, regardless of the locks that it holds; however there is no guarantee that the value read will remain *stable*. A read is *stable* only if no concurrent transaction may write to the same location and invalidate the value that was read. If a transaction holds locks that ensure that the value returned by a read operation is stable and cannot be altered by concurrent transactions, then a transaction may logically observe the result of the read operation and use that value to perform computation. The distinction between stable and unstable reads is key to reasoning about *speculative locking*, which we discuss in Section 3.1.2.

3.1.1 Concrete Semantics

Figure 3.2 describes the concrete semantics of transaction operations over concrete heaps; the semantics are standard but we include them for completeness. The judgment $h, L, t \rightarrow h', L$ holds if executing operation t in global heap h and with global locks taken L yields an updated heap h' and update global lock set L'. Rule (CLock) states that a transaction may acquire any global lock l not already held by any transaction, and the lock l is marked as locked. Rule (CUnlock) allows any transaction to release any lock that is held. There is no requirement that a transaction release only the locks it acquired; such a requirement is enforced by the static well-lockedness rules, not by the concrete semantics.

Rule (CRead) allows any transaction to atomically read the state of any heap cell, whereas rule (CWrite) allows any transaction to atomically update any heap cell. Finally the (CObserve) rule states that every observe operation is physically a no-op; observe is a logical operation that identifies facts relevant to the serializability proof of a transaction; physically it does nothing.

Coarse	**Intermediate**	**Fine**
lock l	lock l_1	lock l_1
read$(m_1) = \mathsf{T}$	read$(m_1) = \mathsf{T}$	read$(m_1) = \mathsf{T}$
observe$(m_1) = \mathsf{T}$	observe$(m_1) = \mathsf{T}$	observe$(m_1) = \mathsf{T}$
read$(m_3) = \mathsf{F}$	read$(m_3) = \mathsf{F}$	lock l_3
observe$(m_3) = \mathsf{F}$	observe$(m_3) = \mathsf{F}$	read$(m_3) = \mathsf{F}$
read$(m_4) = \mathsf{F}$	lock l_0	observe$(m_3) = \mathsf{F}$
observe$(m_4) = \mathsf{F}$	read$(m_4) = \mathsf{F}$	lock l_4
unlock l	observe$(m_4) = \mathsf{F}$	read$(m_4) = \mathsf{T}$
	unlock l_1	observe$(m_4) = \mathsf{T}$
	unlock l_0	unlock l_4
		unlock l_3
		unlock l_1

Figure 3.3: Three transaction traces that observe the values of three arbitrarily chosen memory locations m_1, m_3, and m_4 under three different lock placements.

3.1.2 Lock Placements

We associate a *logical lock* with every heap location $m \in \mathcal{M}$. Whenever a transaction observes or changes the value of a memory location it must hold the associated logical lock. Unfortunately it is inefficient to attach a distinct lock to every memory location. Instead we use a smaller set of *physical locks* (or simply *locks*) $\mathcal{L}$ to implement logical locks; a *lock placement* describes the mapping from logical locks to physical locks. Different choices of placement function describe different granularities of locking.

Formally, a *lock placement* ψ for a boolean heap is a mapping from each location $m \in \mathcal{M}$ to a guarded set of locks that protect it. Each entry in $\psi(m)$ is a pair of a lock $l \in \mathcal{L}$ and a *guard* ϕ, which is a condition under which l protects m. The guard is used to express speculative locking. A guard is a boolean combination of heap assertions $m \mapsto b$; for a given memory location each lock may only appear at most once on the left hand side of a guarded lock pair, and the set of guards must be mutually exclusive and total, that is, exactly one guard is true for any given heap state.

For example, suppose $\mathcal{M} = \{m_0, \ldots, m_{k-1}\}$. Different placements allow us to describe a range of different locking granularities:

- A coarse-grain locking strategy protects every memory location with the same lock, that is, set $L = \{l\}$ and set $\psi(m_i) = \{(l, \mathsf{T})\}$ for all i. To observe or write to any memory location a transaction must hold lock l. Since the lock placement does not depend on the state of the heap, the guard is simply the constant T.

- A medium-grain locking strategy stripes memory locations across a small set of locks. Set $L = \{l_0, \ldots, l_{p-1}\}$, and then set $\psi(m_i) = \{(l_{(i \bmod p)}, \mathsf{T})\}$ for all i. To observe or write to memory location m_i, we must hold lock $l_{(i \bmod p)}$.

- A fine-grain strategy associates a distinct lock with every memory location. Set $L = \{l_0, \ldots, l_{k-1}\}$ and set $\psi(m_i) = \{(l_i, \mathsf{T})\}$ for all i. To observe or write to memory location m_i we must hold lock l_i.

Figure 3.3 shows three variants of a transaction that reads memory location m_1, m_3 and m_4 (chosen arbitrarily for the example), observing values T, F, and F respectively. The figure shows a variant of the transaction for each locking granularity, using $p = 2$ physical locks in the medium-grain case.

A *speculative lock placement* is a placement in which the identity of a lock that protects a memory location depends on the memory location itself. For example a simple speculative placement is as follows. Let $L = \{l_f, l_t\}$ and $\mathcal{M} = \{m\}$. Set

$$\psi(m) = l_f \text{ if } m \mapsto \mathsf{F}, \text{ or } l_t \text{ if } m \mapsto \mathsf{T} \tag{3.1}$$

Under this placement, lock l_f protects memory location m if m contains the value F, whereas lock l_t protects memory location m if m contains the value T.

A more realistic example of speculative lock placement is motivated by transactional predication [Bronson et al., 2010a] which uses a speculative placement of STM metadata. We use a collection $\mathcal{M} = \{m_1, \ldots, m_k\}$ of memory locations to model a concurrent set. Location m_i has value T if value i is present in the set. We use $L = \{l_\perp, l_1, \ldots, l_k\}$ and the placement

$$\psi(m_i) = l_\perp \text{ if } m_i \mapsto \mathsf{F}, \text{ or } l_i \text{ if } m_i \mapsto \mathsf{T}$$

The speculative placement allows us to attach a distinct lock to every entry present in the set, without also requiring that we keep around a distinct lock for every entry that is absent from the set. Two transactions that operate on keys present in the set only contend on the same lock if they are accessing the same key. Transactions that operate on keys that are absent will however contend on $l_\perp$; this strategy is effective if we expect sets to have at most a small fraction of all possible elements at any one time. If necessary, we can reduce contention on absent entries to arbitrarily low levels by striping the logical locks protecting absent entries across a set of physical locks $l_\perp^1, l_\perp^2, \ldots$ as discussed earlier.

It may not be immediately obvious how to acquire a lock on a memory location when we do not know which lock to take without knowing the value of the memory location. The key to this apparent circularity is that a transaction can use unstable reads to guess the identity of the correct lock; once the transaction has acquired the lock it can redo the read to verify that its guess was correct. If the transaction guesses correctly, then the second read is stable. If the transaction guesses incorrectly it can release the lock and repeat the process. Figure 3.4(a) shows a transaction that observes the state of m under the speculative lock placement of Equation (3.1). If another transaction had raced, we might have had to retry the read, as shown in Figure 3.4(b). Finally, to perform an update, we must hold both locks, as shown in Figure 3.4(c); otherwise by changing m we might implicitly release a lock that another transaction holds on the state of m.

	(a)	**(b)**	**(c)**
1:	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{lock}\ l_f$
2:	$\mathsf{lock}\ l_t$	$\mathsf{lock}\ l_t$	$\mathsf{lock}\ l_t$
3:	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{read}(m) = \mathsf{F}$	$\mathsf{read}(m) = \mathsf{T}$
4:	$\mathsf{observe}(m) = \mathsf{T}$	$\mathsf{unlock}\ l_t$	$\mathsf{write}(m, \mathsf{F})$
5:	$\mathsf{unlock}\ l_t$	$\mathsf{lock}\ l_f$	$\mathsf{unlock}\ l_t$
6:		$\mathsf{read}(m) = \mathsf{F}$	$\mathsf{unlock}\ l_f$
7:		$\mathsf{observe}(m) = \mathsf{F}$	
8:		$\mathsf{unlock}\ l_f$	

Figure 3.4: Traces that read and write a memory location m under the speculative lock placement $\psi(m) = \{(l_t, m \mapsto \mathsf{T}), (l_f, m \mapsto \mathsf{F})\}$. In (a) the trace observes the value of m; in (b) the trace incorrectly speculates the lock protecting m due to a concurrent update, and transaction (c) writes to m by taking both locks.

3.1.3 Well-Locked Transactions

We represent the state of a transaction as two sets: the *observation set* Ω and a *lock set* L. The observation set Ω is a set of heap assertions $m \mapsto b$ that represent a transaction's local picture of the heap. The lock set L is a set of *physical locks* held by the transaction. Every heap assertion in the observation set must be *stable*; informally, the facts in the observation set are logically locked and cannot be invalidated by a concurrent interfering transaction. We write $\Omega[m \mapsto b]$ to denote the result of adding or updating the heap observation $m \mapsto b$ to Ω, replacing any existing observations about m. The predicate $\mathsf{locked}_\psi(m, \Omega, L)$ holds for heap location m if a transaction with heap observations Ω and locks L has logically locked location m under lock placement ψ, where $\Omega \vdash \phi$ denotes entailment:

$$\mathsf{locked}_\psi(m, \Omega, L) = \exists (l, \phi) \in \psi(m).\ \ l \in L \land \Omega \vdash \phi$$

The judgement $\Omega, L \vdash_\psi t; \Omega', L'$ defined in Figure 3.5 characterizes *well-locked operations*. The judgment holds if whenever a transaction with observations Ω and holding locks L executes operation t, then on completion of the operation the transaction has new observations Ω' and locks L'. Given the set of physical reads, writes and locks that a transaction performs, the well-lockedness judgement computes the set of stable observations of the transaction, and ensures that a transaction's logical observations and writes only occur on locations on which a transaction holds logical locks.

The (FLOCK) rule allows a transaction to acquire a lock l if the transaction does not already have l in its set of locks L; acquiring a lock has no affect on the observation set Ω. The (FUNLOCK) rule allows a transaction to release any lock l in its lock set L; any facts in Ω that were protected by l are no longer stable, so the rule uses the *stabilization operator* to compute a new stable set of observations Ω'. The *stabilization* of a set of observations Ω_0 under locks L and placement ψ, written $\lceil \Omega_0 \mid L; \psi \rceil$, is the limit of the monotonic sequence:

$$\Omega_{i+1} = \{m \mapsto b \in \Omega_i \mid \mathsf{locked}_\psi(m, \Omega_i, L)\}$$

$$\text{(FLOCK)}\quad \frac{l \notin L}{\Omega, L \vdash_\psi \mathsf{lock}\ l; \Omega, L \cup \{l\}} \qquad \text{(FUNLOCK)}\quad \frac{l \in L \qquad L' = L \setminus \{l\} \qquad \Omega' = \lceil \Omega \mid L'; \psi \rceil}{\Omega, L \vdash_\psi \mathsf{unlock}\ l; \Omega', L'}$$

$$\text{(FOBSERVE)}\quad \frac{(m \mapsto b) \in \Omega}{\Omega, L \vdash_\psi \mathsf{observe}(m) = b; \Omega, L}$$

$$\text{(FREADUNSTABLE)}\quad \frac{\Omega' = \Omega \cup \{m \mapsto b\} \qquad \neg\mathsf{locked}_\psi(m, \Omega', L)}{\Omega, L \vdash_\psi \mathsf{read}(m) = b; \Omega, L}$$

$$\text{(FREADSTABLE)}\quad \frac{\Omega' = \Omega \cup \{m \mapsto b\} \qquad \mathsf{locked}_\psi(m, \Omega', L)}{\Omega, L \vdash_\psi \mathsf{read}(m) = b; \Omega', L}$$

$$\text{(FWRITE)}\quad \frac{\begin{array}{c} m \in \operatorname{dom} \Omega \qquad \Omega' = \Omega[m \mapsto b] \\ \Big(\forall m', l, \phi.\ (l, \phi) \in \psi(m') \wedge m \text{ appears in } \phi \implies l \in L\Big) \end{array}}{\Omega, L \vdash_\psi \mathsf{write}(m, b); \Omega', L}$$

Figure 3.5: Well-locked transaction operations on flat maps: $\Omega, L \vdash_\psi t; \Omega', L'$

Note that the limit always exists, because Ω_0 is finite (since it is constructed by a finite transaction execution) and the empty set is always a fixed point of the equation if no larger set is. A set of observations Ω is *stable* under locks L and placement ψ if Ω is its own stabilization, that is, $\Omega = \lceil \Omega \mid L; \psi \rceil$.

Rule (FOBSERVE) states that a transaction may logically observe any stable fact from its stable observation set Ω. The (FREADUNSTABLE) rule allows a transaction to perform a speculative read on a memory location on which the transaction does not hold a lock; however since the result may not be stable the rule does not update the set Ω. To enable reasoning about speculation, the determination whether the read is stable or not occurs in a context that includes the read of m; since we assume that reads are atomic, there is an instant in time at which both the old stable facts in Ω and the newly read value of m hold, and it is in that context that we determine stability. The (FREADSTABLE) rule allows a transaction to read memory locations on which it holds a lock; since such a read is stable the rule updates the set of observations Ω with the newly read information about the heap. Finally the (FWRITE) rule requires that a transaction can only update a location m if it holds the lock on m; furthermore the lock for any location m' for which m appears in a guard must also be held by the transaction—hence no transaction can destabilize the observations of another transaction. The last condition together with $\mathsf{locked}_\psi(m, \Omega, L)$ also implies that $\mathsf{locked}_\psi(m, \Omega', L)$ holds, which is why the latter is not listed as a precondition of the rule.

A transaction $\mathbf{T} = t^1 \dots t^k$ is *well-locked* if there exists a sequence of lock sets L^i and observation sets Ω^i

such that

$$L^0 = L^k = \emptyset, \quad \Omega^0 = \Omega^k = \emptyset, \text{ and } \Omega^{i-1}, L^{i-1} \vdash_\psi t^i; \Omega^i, L^i \text{ for } 1 \le i \le k.$$

As an example of applying the rules, consider again the speculative read transaction shown in Figure 3.4(b). Let Ω^i and L^i denote the lock sets of the transaction after line i. Initially we have $\Omega^0 = \emptyset$ and $L^0 = \emptyset$. The read on line 1 is unstable, so $\Omega^1 = \emptyset$ and $L^1 = \emptyset$. The lock on line 2 adds an entry to the lock set l_t, so $\Omega^2 = \emptyset$ and $L^2 = \{l_t\}$. The read on line 3 yields $m \mapsto \mathsf{F}$, however the read would only be stable if $\mathsf{locked}_\psi(m, \{m \mapsto \mathsf{F}\}, \{l_t\})$ holds, which it does not; once again we have $\Omega^3 = \emptyset$ and $L^3 = \{l_t\}$. Lines 4 and 5 update the lock set; we have $\Omega^4 = \Omega^5 = \emptyset$, $L^4 = \emptyset$, and $L^5 = \{l_f\}$. The read on line 6 once again yields $m \mapsto \mathsf{F}$, but this time the predicate $\mathsf{locked}_\psi(m, \{m \mapsto \mathsf{F}\}, \{l_f\})$ holds and the read is stable, yielding $\Omega^6 = \{m \mapsto \mathsf{F}\}$ and $L^6 = \{l_f\}$. The logical observation of $m \mapsto \mathsf{F}$ on line 7 is permitted by the judgement since we know $m \mapsto \mathsf{F}$ is part of the stable heap; the observation and lock sets are unchanged ($\Omega^7 = \Omega^6$, $L^7 = L^6$). Finally, line 8 releases lock l_f, so we have $L^8 = \emptyset$. The assertion about m in Ω^7 is no longer stable, so the stabilization operator removes it from the observation set, finally yielding $\Omega^8 = \emptyset$.

3.1.4 Serializability of Well-Locked Transactions

A *schedule* s for a set of transactions $\mathbf{T}_1, \ldots, \mathbf{T}_k$ is a permutation of the concatenation of all transactions in the set, such that each transaction $\mathbf{T}_i$ is a subsequence of s. Formally, a schedule is *valid* if it corresponds to an execution of the concrete semantics. Informally, validity requires the execution respect the mutual exclusion property of locks, and memory accesses must accurately reflect the state of the global heap. A schedule is *serial* if operations of different transactions are not interleaved.

Lemma 3.1. Let s be a valid schedule of a set of well-locked transactions $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$. Let Ω_i^j and L_i^j be the set of observations and locks of transaction $\mathbf{T}_i$ after schedule step j. Let h^j be the heap after schedule step j. Then for all time steps j:

- the lock sets $\{L_i^j\}_{i=1}^k$ are disjoint, and
- the observation sets $\{\Omega_i^j\}_{i=1}^k$ are stable, have disjoint domains, and heap h^j is an extension of each $\{\Omega_i^j\}_{i=1}^k$.

Proof. By induction on the length of the schedule. The result is immediate for a schedule of length 0 since the observation sets of each transaction must necessarily be empty by the definition of well-lockedness. For the inductive case, assume the result holds for all schedules of length n and consider a schedule with length $n+1$. Suppose without loss of generality that operation $n+1$ is an operation t_1 performed by transaction 1. Since transaction 1 is well-locked, we know that $\Omega_1^n, L_1^n \vdash_\psi t_1; \Omega_1^{n+1}, L_1^{n+1}$ holds and we perform a case analysis of the possible derivations:

- Rule (FLOCK): The disjointness of lock sets is an immediate consequence of the validity of schedule s. Acquiring a lock does not change the stability of a transaction's observation set, and cannot destabilize the stable observation set of any transaction, so the stability, disjointness, and extension properties follow from the induction hypothesis.

- Rule (FUNLOCK): Releasing a lock yields disjoint lock sets by the induction hypothesis. It is always sound to discard anything from a transaction's stable observation set Ω, and stability of the resulting observations Ω' holds by definition of the stabilization operator. Disjointness and extension hold by the induction hypothesis and since $\Omega' \subseteq \Omega$.

- Rule (FOBSERVE): Immediate.

- Rule (FREADUNSTABLE): Immediate.

- Rule (FREADSTABLE): The observation sets of each transaction must be the same before and after the read, with the exception of Ω_1, which contains the additional result of the reading m. We must have $\mathsf{locked}_\psi(m, \Omega_1, L_1)$ so by definition we have $(l, \phi) \in \psi(m)$ such that $l \in L_1$ and $\Omega_1 \vdash \phi$. Disjointness follows by the disjointness of the guards ϕ and the disjointness of local heaps due to the induction hypothesis. Extension follows by the validity of the schedule and the induction hypothesis.

- Rule (FWRITE): We have $\mathsf{locked}_\psi(m, \Omega_1, L_1)$ because the rule requires that $m \in \mathrm{dom}\,\Omega$ and hence m must be stable; hence we know that no other transaction may have m in its stable observation set by the disjointness of guards and heaps. Further, the rule requires that the transaction must hold any locks whose association with memory locations may change as a consequence of the write, and hence the stability of each Ω_i holds. Finally, extension follows from the inductive hypothesis.

□

The disjointness of observation sets in Lemma 3.1 is a consequence of the fact that our physical locks are exclusive. If we allowed shared/exclusive locks, then we would also need to allow observation sets to overlap on values protected by shared locks.

A well-locked transaction $\mathbf{T} = (t^i)_{i=1}^k$ is *logically two-phase* if the domains of the observation sets of the transaction have a growing phase and a shrinking phase, that is, there exists some j such that for all i where $1 \le i \le j$, we have $\mathrm{dom}\,\Omega^{i-1} \subseteq \mathrm{dom}\,\Omega^i$ and for all i where $j < i \le k$ we have $\mathrm{dom}\,\Omega^{i-1} \supseteq \mathrm{dom}\,\Omega^i$.

A *logical schedule* $\hat{\mathsf{s}}$ is the subsequence of a schedule s consisting of all the observe and write operations. The definition of a logical schedule mirrors the usual definition of a schedule in the proof of serializability of two-phase locking; here only logical reads (i.e., observe operations) and writes count for the purposes of serializability.

Two operations *conflict* if they access the same memory location m. Two schedules s_1 and s_2 are *conflict-equivalent* if the logical schedule $\hat{\mathsf{s}}_1$ can be turned into the logical schedule $\hat{\mathsf{s}}_2$ by a sequence of swaps of adjacent non-conflicting operations.

Lemma 3.2. Any valid schedule of a set of well-locked, logically two-phase transactions $\{\mathbf{T}_1, \dots, \mathbf{T}_k\}$ is conflict-equivalent to a serial logical schedule.

Proof. Identical to the usual proof of serializability of two-phase locking [Eswaran et al., 1976]. A fact is "locked" when it is added to the observation set of a transaction, and "unlocked" when it is removed from the observation set of the transaction. Disjointness of observation sets is guaranteed by Lemma 3.1. The "read" and "write" operations of the two-phase locking proof correspond to observe and write operations on the observation set. □

3.1.5 Shared/Exclusive Logical Locks

A limitation of the protocol just presented is that locks are exclusive — holding a lock gives a transaction sole access to an edge, even if the transaction only wants to read the edge. Lock placement is a separate issue from whether non-exclusive locks exist for reading. Exclusive locks are sufficient to illustrate all of the important features of our techniques and have the advantage of not introducing the extra and extraneous complications of supporting non-exclusive access. However, non-exclusive locks are important, and so we briefly illustrate how to extend our approach to locks providing shared read access.

To allow shared access to fields we relax the requirement that guards must be mutually exclusive, thereby allowing each logical lock to map to many physical locks at the same time. Under the relaxed definition of placement, a transaction has shared access to a memory location m if it holds at least one of the locks that protect m, whereas a transaction has exclusive access to m if it holds all of the locks that protect m. Formally, a transaction has shared access to a memory location m if $\mathsf{locked}_\psi(m, \Omega, L)$ holds. We also define a new predicate $\mathsf{exclusive}_\psi(m, \Omega, L)$ which holds for heap location m if a transaction with heap observations Ω and locks L has an exclusive logical lock on location m under lock placement ψ:

$$\mathsf{exclusive}_\psi(m, \Omega, L) = \forall (l, \phi) \in \psi(m).\ \ l \in L \vee \Omega \vdash \neg\phi$$

To show serializability, we need to add an additional precondition to the (FWRITE) rule requiring that a transaction have exclusive access to any memory location it writes. The statement of the proof of Lemma 3.1 must be altered since different observation sets may share fields on which they hold a shared lock—only the exclusively held fields must be disjoint between transactions. Finally we must update the definition of a two-phase transaction to ensure that transactions only release exclusive access to a field in the shrinking phase of a transaction.

3.2 Mutable Tree-Structured Heaps

In Section 3.1 we described a locking protocol for a class of flat heaps with a fixed set of memory locations and locks. In this section we extend our results to dynamically allocated, mutable tree-shaped heaps with a placement function based on paths.

A *tree heap* h consists of a set of objects, each with a unique name, usually denoted x or y. Every object has a fixed set of fields $\mathcal{F}$. Each object field $x.f$ contains a pointer either to some object y or nil. The heap contains

Symbol	Meaning
$f, \mathbf{f}, \mathcal{F}$	fields
x, y, ρ	object names
$e ::= \mathsf{nil} \mid x$	expressions
$\omega ::= x.f \mapsto e$	heap assertions
$\psi \subseteq 2^{\mathbf{f}} \to 2^{\mathbf{f}}$	placements
$t ::= \mathsf{write}(x.f, e) \mid \mathsf{read}(x.f) = e \mid \mathsf{observe}(x.f) = e \mid x = \mathsf{new}() \mid \mathsf{lock}\ x \mid \mathsf{unlock}\ x$	transaction ops.

Figure 3.6: Tree transactions

a distinguished *root object*, named ρ. In a quiescent state, that is, in the absence of running transactions, we require that the heap be a forest.

Unlike flat heaps, we associate a logical lock with every field of every object in the heap. Unlike the flat heaps of Section 3.1 we do not assume that we have a separate set of locks distinct from the set of memory locations; instead, following the practice of languages such as Java, we require that every heap object can function as a physical lock, and we use a lock placement function to describe a policy for mapping the logical locks attached to fields onto the physical locks (the objects). To define the lock placement, we use access paths from the root ρ to name both the fields we want to protect and the objects whose physical locks protect them.

We extend the set of transaction operations of Section 3.1 to read from and write to fields of objects, to handle dynamic allocation of new objects, and to apply lock and unlock operations to objects rather than a separate set of locks. The transaction operations, shown in Figure 3.6, are: write an expression e (either an object y or nil) to field f of object x ($\mathsf{write}(x.f, e)$)), a possibly unstable read of field f of object x yielding result e ($\mathsf{read}(x.f) = e$), a stable observation of field f of object x yielding e ($\mathsf{observe}(x.f) = e$), allocation of a fresh object ($x = \mathsf{new}()$), locking an object ($\mathsf{lock}\ x$), and unlocking an object ($\mathsf{unlock}\ x$).

3.2.1 Lock Placements

We name edges in the heap as a non-empty field path (a sequence of field names) $\mathbf{f} = f_1 f_2 \ldots$ from the root, ending in the edge in question. Since the path names a field in the heap, the path must be non-empty. We also name objects using fields, except that the path ends at the object the field points to; note that in the case of objects the empty path names the root of the heap. A *lock placement* ψ is a function from non-empty paths to paths that maps every edge in a heap to an object whose attached physical lock protects it.

Consider heaps with field labels drawn from the set $\mathcal{F} = \{a, b\}$. We can protect every edge of the heap with a single coarse-grain lock at the root by setting $\psi_1(\mathbf{f}) = \epsilon$ for all $\mathbf{f}$. If we want different locks for the a and b subtrees, we can use the lock placement

$$\psi_2(\mathbf{f}) = \begin{cases} a & \text{if } a \prec \mathbf{f} \\ b & \text{if } b \prec \mathbf{f} \\ \epsilon & \text{if } \mathbf{f} = a \text{ or } \mathbf{f} = b \end{cases} \tag{3.2}$$

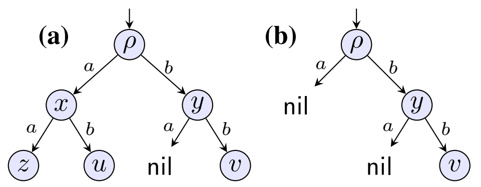


Figure 3.7: Two examples of tree-structured heaps. Nodes (e.g., x, y) represent objects, whereas edges (labeled a, b) represent fields. Node ρ is the root object of the heap.

where $\mathbf{g} \prec \mathbf{f}$ denotes that $\mathbf{g}$ is a strict prefix of $\mathbf{f}$. For example, in Figure 3.7(a), under placement ψ_2 the lock at ρ protects the edges from ρ to x and from ρ to y, the lock at x protects the edges from x to z and from x to u, and the lock at y protects the edge from y to v.

If for a path $\mathbf{f}$ the placement path $\psi(\mathbf{f})$ leads to nil in the heap, we use the lock given by the longest non-nil prefix of the path $\psi(\mathbf{f})$; that is, we use the lock on the object immediately preceding the edge to nil in the placement path. For example, consider the heap of Figure 3.7(b) under the placement ψ_2. The lock that protects the edge named by the path ab according to the placement is $\psi_2(ab) = a$, however the edge a from the root node ρ points to nil. In this case, we use the lock on the longest non-nil prefix of $\psi_2(ab)$ to protect the edge ab, namely ρ itself. In this chapter we only consider the convention of placing locks on the longest non-nil prefix, however other conventions are possible, for example, always placing such locks at the root.

Modifications to the heap may implicitly alter the mapping from logical locks to physical locks. If a transaction updates an edge, then the transaction must hold both before and after the update all logical locks whose mapping to physical locks may change. For example consider again the lock placement ψ_2 in the context of the tree heap shown in Figure 3.7(b). According to the placement the lock on ρ protects the edge a from the root; however since edge a points to nil, edges on any path that begins with a are also protected by the lock at a. If a transaction were to set $\rho.a$ to point to a node w, the lock at w would now protect the edges on paths that begin with a; the transaction has *split* the logical roles of the lock at ρ before the write between the lock at ρ and the lock on node w. Whenever a transaction splits or merges locks (e.g., by setting the field $\rho.a$ to nil again), it must hold every lock involved.

Figure 3.8(a) shows a trace of a transaction that adds a new edge labeled a from object y to a fresh object w to the heap of Figure 3.7(b) under placement ψ_2. The transaction acquires two locks, namely the lock at ρ that protects the edge from ρ to y, and the lock at y that protects the entire subtree rooted at y. We need not hold a lock on w when adding w into the tree; linking w into the heap does not change the association between logical and physical locks, and hence we do not need to hold the lock on w either before or after adding it to the heap.

If we desire a finer-grain locking, we can use a lock attached to every object to protect the fields of that object by using the placement function

$$\psi_3(\mathbf{g}f) = \mathbf{g} \text{ for any } \mathbf{g}, f. \tag{3.3}$$

(a)	**(b)**	**(c)**
lock ρ	lock ρ	read$(\rho.b) = y$
read$(\rho.b) = y$	read$(\rho.b) = y$	lock y
observe$(\rho.b) = y$	observe$(\rho.b) = y$	read$(\rho.b) = y$
lock y	lock y	observe$(\rho.b) = y$
$w =$ new ()	read$(y.a) =$ nil	read$(y.a) =$ nil
write$(y.a, w)$	$w =$ new ()	$w =$ new ()
unlock y	lock w	lock w
unlock ρ	write$(y.a, w)$	write$(y.a, w)$
	unlock w	unlock w
	unlock y	unlock y
	unlock ρ	

Figure 3.8: Three transaction traces that add a new outgoing edge labelled a from node y to the tree of Figure 3.7(b) under: (a) the lock placement ψ_2 defined in Equation (3.2), (b) the lock placement ψ_3 defined in Equation (3.3), and (c) the lock placement ψ_4 defined in Equation (3.4).

The lock placement ψ_3 places the lock that protects each edge f on the object at the head of the edge. Figure 3.8(b) shows a trace of a transaction that again adds the edge labeled a from node y to a fresh node w to the heap of Figure 3.7(b), this time under lock placement ψ_3. Unlike the transaction of Figure 3.8(a), we need to ensure that by adding the new edge the write does not implicitly change the mapping from edges to locks. The well-lockedness conditions, which we introduce shortly, require that a transaction hold all physical locks which may map to different logical locks before and after a write. The operation read$(y.a) =$ nil verifies that there is no existing subtree of y reachable via edge a. Before the update the lock at y protects every possible edge reachable from $y.a$, however after the write the lock y only protects the edge $y.a$ itself, whereas the lock at w protects everything reachable from node w. Hence we must hold lock w when performing the write, since adding the edge splits the lock at y. (In general one must hold locks when connecting objects into the heap; however in this specific case, since the write which links w to the heap is the last write in the transaction it would be possible to optimize away the lock and unlock.)

Finally, we can use a speculative placement, as in the last section. If we set

$$\psi_4(\mathbf{f}) = \mathbf{f} \tag{3.4}$$

the lock that protects each edge is located at the target of the edge. Figure 3.8(c) once again shows a transaction that adds a fresh edge labeled a to node w, this time using lock placement ψ_4. The transaction begins by performing a speculative read to guess that the identity of the object whose lock protects $\rho.b$ is y; the transaction then acquires the lock on y. It is possible that a concurrent write to $\rho.b$ means that the lock on y no longer protects $\rho.b$, so the transaction must perform the read again. The second read also returns y, and hence we know that the second read is stable since the transaction already holds lock y. The transaction then performs a read of $y.a$ which returns nil. The value of the placement function for edge $y.a$ is $\psi(ba) = ba$, however since edge ba points to nil, the lock on the longest non-nil prefix of ba protects ba, in this case path b (node y). Since we hold the lock on y already, we know that the read of $y.a$ is also stable. Finally, the transaction must hold the lock on w when adding it to the heap to maintain the invariant that a transaction must hold all physical locks whose logical/physical mapping changes as a consequence of a write.

3.2.2 Well-Locked Transactions

We represent a transaction's state by three sets L, Ω, and Γ. As before, L is the set of locks that transaction holds, and Ω is an observation set consisting of stable heap observations of the form $x.f \mapsto e$; equivalently, we can think of Ω as a transaction's local part of the heap. We do not require Ω be a forest; a transaction may create any heap shapes it desires within its local heap, allowing the common idiom of temporarily violating invariants and later restoring them during the course of a transaction. However, the forest invariant must be restored when the transaction releases objects in its local heap back into the global heap. Enforcing this condition is the purpose of the *global non-alias set* Γ. An object x is a member of Γ if the transaction has shown that there is no globally visible path from the root to x (i.e., the transaction has locked the edge to x). The well-lockedness rules for tree heaps ensure that there is at most one globally-visible edge to any node and hence the globally-visible part of the heap is a forest. At the start of every transaction Γ is the empty set. Transactions add entries to Γ by discovering global edges to nodes and transferring them into their local heap Ω; entries are removed from Γ when pointers to objects are released from the stable heap Ω back into the global heap.

The *path alias* judgement $\Omega \vdash \mathbf{f} \sim x$ holds if $\mathbf{f}$ is a path in Ω from the root to location x; that is, if $|\mathbf{f}| = k$, then there is a sequence of nodes $\mathbf{v} = v_0 v_1 \cdots$ such that $(\rho.f_0 \mapsto v_0) \in \Omega$, $(v_{i-1}.f_{i-1} \mapsto v_i) \in \Omega$ for all $1 < i < k-1$, and $v_{k-1}.f_{k-1} \mapsto x$. We write $\mathbf{f} \in \Omega$ if the path $\mathbf{f}$ from the root vertex exists in Ω, that is, $\Omega \vdash \mathbf{f} \sim x$ holds for some object x.

The *restriction of a path* $\mathbf{f}$ to a local heap Ω, written $\mathbf{f}|_\Omega$ is defined as:

$$\mathbf{f}|_\Omega = \begin{cases} \mathbf{f} & \text{if } \mathbf{f} \in \Omega \\ \mathbf{g} & \text{where } \exists \mathbf{g}, h.\ \mathbf{g}h \preceq \mathbf{f} \ \wedge\ \Omega \vdash \mathbf{g}h \sim \mathsf{nil} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The restriction of path $\mathbf{f}$ is either $\mathbf{f}$ itself if present in Ω, or a prefix of $\mathbf{f}$ leading to nil in Ω. The restriction of a path is undefined if the path $\mathbf{f}$ leaves the local heap Ω.

We hold the lock on an edge reached via a path if we hold the lock on the node given by corresponding lock placement, restricted to the local heap:

$$\mathsf{pathlocked}_\psi(\mathbf{f}, \Omega, L) ::= \exists x \in L.\ \Omega \vdash \psi(\mathbf{f})|_\Omega \sim x$$

We hold the lock on a field f of an object x under local heap Ω, global non-alias set Γ and locks L, written $\mathsf{locked}_\psi(x.f, \Omega, \Gamma, L)$, if we hold the lock protecting field f on every path to field f in the local heap Ω, and

$$\text{(TNEW)}\ \frac{\Omega' = \Omega \cup \{x.f \mapsto \mathsf{nil} \mid f \in \mathcal{F}\} \qquad x \notin \mathrm{dom}\,\Omega \qquad x \notin \Gamma \qquad \Gamma' = \Gamma \cup \{x\}}{\Omega, \Gamma, L \vdash_{\psi} x = \mathsf{new}(); \Omega', \Gamma', L}$$

$$\text{(TLOCK)}\ \frac{x \notin L}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{lock}\ x; \Omega, \Gamma, L \cup \{x\}}$$

$$\text{(TUNLOCK)}\ \frac{x \in L \qquad L' = L \setminus \{x\} \qquad (\Omega', \Gamma') = \lceil \Omega; \Gamma \mid L'; \psi \rceil \qquad \mathsf{forest}(\Omega, \Omega', \Gamma, \Gamma')}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{unlock}\ x; \Omega', \Gamma', L'}$$

$$\text{(TOBSERVE)}\ \frac{(x.f \mapsto e) \in \Omega}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{observe}(x.f) = e; \Omega, \Gamma, L}$$

$$\text{(TREADUNSTABLE)}\ \frac{x.f \notin \mathrm{dom}\,\Omega \qquad \Omega' = \Omega \cup \{x.f \mapsto e\} \qquad \neg\mathsf{locked}_{\psi}(x.f, \Omega', \Gamma, L)}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{read}(x.f) = e; \Omega, \Gamma, L}$$

$$\text{(TREADSTABLE)}\ \frac{\begin{array}{c} x.f \notin \mathrm{dom}\,\Omega \qquad \Omega' = \Omega \cup \{x.f \mapsto e\} \\ \mathsf{locked}_{\psi}(x.f, \Omega', \Gamma, L) \qquad \Gamma' = \begin{cases} \Gamma & \text{if } e = \mathsf{nil} \\ \Gamma \cup \{y\} & \text{if } e = y \end{cases} \end{array}}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{read}(x.f) = y; \Omega', \Gamma', L}$$

$$\text{(TWRITE)}\ \frac{\begin{array}{c} x.f \in \mathrm{dom}\,\Omega \qquad \Omega' = \Omega[x.f \mapsto y] \\ \Big(\forall \mathbf{g}, \mathbf{h}.\ (\Omega \vdash \mathbf{g} \sim x) \land \mathbf{g}f \preceq \psi(\mathbf{h}) \implies \mathsf{pathlocked}_{\psi}(\mathbf{h}, \Omega, L) \land \mathsf{pathlocked}_{\psi}(\mathbf{h}, \Omega', L)\Big) \end{array}}{\Omega, \Gamma, L \vdash_{\psi} \mathsf{write}(x.f, y); \Omega', \Gamma, L}$$

Figure 3.9: Well-locked tree heap operations: $\Omega, \Gamma, L \vdash_{\psi} t; \Omega', \Gamma', L'$

furthermore Γ implies there are no paths to x outside the local heap. Formally,

$$\mathsf{locked}_{\psi}(x.f, \Omega, \Gamma, L) ::= x \in \Gamma\ \land\ \forall \mathbf{g}.\ \big(\Omega \vdash \mathbf{g} \sim x \implies \mathsf{pathlocked}_{\psi}(\mathbf{g}f, \Omega, L)\big)$$

If the local heap Ω contains cycles, observe that there may be infinitely many paths $\mathbf{g}$ and the locked predicate is well-defined in this case.

The definition of the locked predicate implies the objects that cannot be reached by other transactions are considered locked. In particular if there is no path from the root to a node x in the local heap Ω, and we have $x \in \Gamma$, implying that there is no path to x in the global heap either, then definition of the locked predicate implies that the fields of x are locked. The well-lockedness rules add freshly allocated objects to Γ since they are created disconnected from the global heap, and updates to such nodes are considered correct by the definition of locked, even in the absence of any explicit lock acquisitions.

The judgement $\Omega, \Gamma, L \vdash_{\psi} t; \Omega', \Gamma', L'$ defined in Figure 3.9 captures the class of *well-locked tree operations*. If the judgement holds, then a transaction that executes operation t under stable observation set Ω, objects Γ, and lock set L yields a new stable observation set Ω', objects Γ' and lock set L'. The (TNEW) rule states that all of the fields of a newly allocated object x point to nil, and since there can be no path to x in the heap all of x's fields are stable and $x \in \Gamma$. As before, the (TLOCK) rule allows a transaction to acquire a lock it does not yet hold and has no affect on either Ω or Γ.

In the (TUNLOCK) rule, the stabilization operator is more involved than in the case of flat heaps, because we must compute not just the stable set of heap facts, but also the set of objects for which the transaction has locked the incoming path: if an edge $x.f \mapsto y$ drops out of the stable observation set because a lock is released, the transaction can no longer assume it holds locks on all of the paths to object y. The stabilization (Ω', Γ') of a local heap Ω_0 and global heap Γ_0 under locks L and placement ψ, written $(\Omega', \Gamma') = \lceil \Omega_0; \Gamma_0 \mid L; \psi \rceil$, is the limit of the monotonically decreasing sequence:

$$\Omega_{i+1} = \{x.f \mapsto e \in \Omega_i \mid \mathsf{locked}_\psi(x.f, \Omega_i, \Gamma_i, L)\} \qquad \Gamma_{i+1} = \Gamma_i \setminus \{y \mid x.f \mapsto y \in \Omega_i \setminus \Omega_{i+1}\}$$

In addition, rule (TUNLOCK) requires that transactions maintain the *forest condition*

$$\mathsf{forest}(\Omega, \Omega', \Gamma, \Gamma') ::= \forall y. \left(|\{(x.f \mapsto y) \in \Omega \setminus \Omega'\}| = \begin{cases} 1 & \text{if } y \in \Gamma \setminus \Gamma' \\ 0 & \text{otherwise.} \end{cases} \right)$$

The forest condition ensures that a transaction may only release a pointer to a node y into the global heap if there are no other references to y in the global heap ($y \in \Gamma \setminus \Gamma'$). Furthermore, the condition also ensures that a transaction cannot release two or more pointers to the same location y into the global heap.

The rules (TOBSERVE), (TREADUNSTABLE), and (TREADSTABLE) are similar to the rules in Section 3.1, updated to reflect that the heap now involves objects and fields. Note that (TREADSTABLE) adds the object that is the target of the read to Γ in the case that the field is not nil.

The most interesting rule is (TWRITE). Updating a field $x.f$ to point to y not only changes the paths to y, it changes the paths to every object reachable from y. Thus, as a result of a single field update, the lock placements may change for y and every edge reachable from y. Furthermore, fields no longer reachable from $x.f$ after the update also may have altered lock placements. For this reason a transaction must hold locks on every edge reachable from $x.f$ both before and after the update. These conditions are necessary for safety, but need not be burdensome if the lock placement has a suitable granularity. For example, if the subtrees rooted at x and y are locked by the locks at x and y respectively, the update requires two locks.

A transaction $\mathbf{T} = t^1 \dots t^k$ is *well-locked* if there exists a sequence of lock sets L^i, observation sets Ω^i, and object sets Γ^i such that

$$L^0 = L^k = \emptyset, \qquad \Omega^0 = \emptyset, \qquad \Gamma^0 = \emptyset, \text{ and } \Omega^{i-1}, \Gamma^{i-1}, L^{i-1} \vdash_\psi t^i; \Omega^i, \Gamma^i, L^i \text{ for } 1 \leq i \leq k.$$

A well-locked transaction must begin with all three sets Ω, Γ, L empty. Furthermore at the end of the transaction the set of locks L must be empty again, and hence a transaction must release all of its locks. We do not require that Ω or Γ be empty at the conclusion of a transaction; however since the transaction may not hold any locks on termination, any part of the heap that is stable and in Ω with an empty lock set cannot be reachable from the global and can be garbage-collected.

Lemma 3.3. Let s be a valid schedule of a set of well-locked transactions $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$. Let Ω_i^j, Γ_i^j, and L_i^j be the set of observations, objects, and locks of each transaction after schedule step j. Let h^j be the heap after schedule step j, and suppose h^0 is a forest. Then for all time steps j:

- the lock sets $\{L_i^j\}_{i=1}^k$ are disjoint.
- the observation sets $\{\Omega_i^j\}_{i=1}^k$ are stable, have disjoint domains, and heap h^j is an extension of each $\{\Omega_i^j\}_{i=1}^k$.
- the sets $\{\Gamma_i^j\}_{i=1}^k$ are disjoint, and
- the global heap h^j less edges present in the local heaps $\{\Omega_i^j\}_{i=1}^k$ is a forest. Furthermore if $x \in \Gamma_i^j$ then every pointer to node x in the heap is an element of some local heap $\Omega_{i'}^j$.

Proof. By induction on the length of the schedule. The result is immediate for a schedule of length 0. Suppose the result is true for any schedule of length n, and consider any schedule of length $n + 1$. Without loss of generality, assume transaction number 1 performs the additional operation t^{n+1}. Since transaction 1 is well-locked, we know $\Omega_1^n, \Gamma_1^n, L_1^n \vdash_\psi t; \Omega_1^{n+1}, \Gamma_1^{n+1}, L_1^{n+1}$ holds. There are several possible derivations:

- Rule (TNEW): The result about lock sets is immediate. The concrete semantics of the new operator guarantee that the result x of allocation is fresh, and hence there are no pointers to x in the heap. It follows that all of the fields of x are stable and known only to the current transaction. The result about the observation sets then follows from the induction hypothesis. The result about local heaps and global heaps is a consequence of the freshness of x and the induction hypothesis.
- Rule (TLOCK): Disjointness of lock sets follows from the validity of the schedule. The other results are immediate from the induction hypothesis.
- Rule (TUNLOCK): Disjointness of lock sets is immediate. Stability of the current transaction holds by definition, and for other transactions by the induction hypothesis; unlocking a lock cannot affect other transactions. Disjointness and subsetting hold by the induction hypothesis and the definition of stabilization. The forest condition ensures that a transaction can evict a pointer to a node x from its local heap only if $x \in \Gamma$, and hence the global forest condition is preserved given the induction hypothesis.
- Rule (TOBSERVE): Immediate.
- Rule (TREADUNSTABLE): Immediate.
- Rule (TREADSTABLE): The lock set result is immediate. Stability of each transaction's observation set holds since the rule requires that the transaction hold locks that ensure the newly read heap assertion is stable ($\mathsf{locked}_\psi(x.f, \Omega, \Gamma, L)$) and the induction hypothesis guarantees all other heap assertions are

stable. The lockedness of $x.f$ ensures that the current transaction must hold the lock for $x.f$ on every path. No other transaction can have $x.f$ as part of its stable observation set because of the disjointness of observation sets from the induction hypothesis. The result about Γ and Ω follows from the forest condition on the heap; if we acquired a stable reference to a node y then it must have been the unique reference in the global heap, and hence no other transaction may have $y \in \Gamma$ by the induction hypothesis.

- Rule (TWRITE): The lock set result is immediate. The newly written memory location must be stable since the transaction requires that $x.f$ be present in Ω and from the condition that ensures the transaction holds any locks reachable from $x.f$ that may change identity; the latter condition also ensures that the write cannot affect the stable observation sets of any other transaction. The disjointness and extension conditions hold by the induction hypothesis. The result about local and global heaps is immediate from the induction hypothesis; a transaction may freely update its local heap since the tree-structure condition need only be maintained at at unlock time.

□

Finally, we have a logical serializability lemma analogous to Lemma 3.2:

Lemma 3.4. Any valid schedule of a set of well-locked, logically two-phase tree transactions $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$ is conflict-equivalent to a serial logical schedule.

As in the flat-heap case, these results can be extended to shared/exclusive locks using the approach in Section 3.1.5.

3.3 Transactions on Decomposition Heaps

At the core of the locking protocol of Section 3.2 is the invariant that the global heap is a forest. Since lock placements are defined using access paths, for soundness the locking protocol must verify that a transaction holds the locks that protect an edge on every possible access path. In a forest this requirement is simple, since there can be at most one path between nodes.

In this section we show how to relax the forest restriction and apply lock placements to heaps that are decompositions (Chapter 2), which may contain multiple paths to the same object. The technical machinery developed so far remains almost unchanged, with the exception that the forest condition is replaced by a bounded in-degree condition which ensures that all of the paths to a node are known.

To minimize the complexity of our proof system, and to obtain the most general result possible, in this section we work with heaps that are a relaxation of decompositions; our proof system enforces just enough structure to guarantee that heaps have bounded in-degree, which is the analogue of the forest condition for tree heaps. Enforcing all of the structural conditions mandated by a decomposition would unnecessarily complicate the well-lockedness rules, and is not necessary to address our main interests in this section, which are the safety conditions forced on lock placements because of the heap structure.

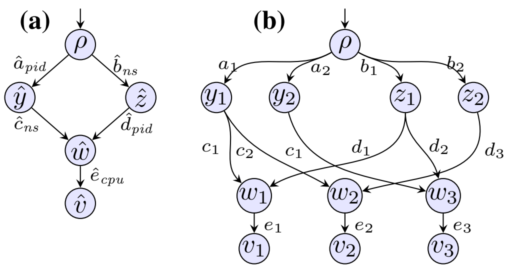


Figure 3.10: **(a)**: A decomposition heap shape, and **(b)**: a decomposition heap that is an instance of decomposition heap shape (a).

A *decomposition heap shape* $\hat{h}$ is a rooted, connected, directed acyclic graph $(\hat{V}, \hat{E})$ consisting of a set of vertices $\hat{V} = \{\hat{u}, \hat{v}, \dots\}$ and a set of edges $\hat{E} \subseteq \hat{V} \times \hat{F} \times \hat{V}$. To simplify the definition of lock placements, we require that each edge of a decomposition be labeled with a unique name drawn from a set $\hat{F}$.

We will use the decomposition of Figure 3.10(a) as a running example, which depicts a decomposition for the process scheduler example of Chapter 2. Recall in the scheduler example, every process has associated fields *pid* (process ID), *ns* (name space), and the process' assigned *cpu*; a pair of a pid and a ns uniquely identify a process. To find the cpu of a particular process using the decomposition, we can first look up the process id by following edge $\hat{a}_{pid}$ and then the process' name space by following edge $\hat{c}_{ns}$, or we can first look up the name space by following edge $\hat{b}_{ns}$ and then the process id by following edge $\hat{d}_{pid}$. For a given pair of process ID and name space, the shared node $\hat{w}$ in the decomposition shape assures us we will get the same result regardless of which path we take.

Like a decomposition, a decomposition heap shape is a static description of a class of heaps. Let $\mathsf{in}(\hat{v})$ be the set of field names incoming to $\hat{v}$ in a decomposition shape and let $\mathsf{out}(\hat{v})$ be the set of outgoing field names. A heap (V, E) is an *instance* of a decomposition shape $\hat{d}$ if

- every vertex in V is an instance v_i of some vertex $\hat{v} \in \hat{V}$,
- every edge $(u_i, f_j, v_k) \in E$ is an instance of some $(\hat{u}, \hat{f}, \hat{v}) \in \hat{E}$, and
- every vertex v_i has exactly one instance f_i of every incoming edge $\hat{f} \in \mathsf{in}(\hat{v})$.

The last condition provides a bound on the in-degree of a vertex, which is the key to applying path-based lock placements to decomposition heaps.

The distinction between a decomposition as defined in Chapter 2 and a decomposition heap shape is that decomposition heap shape only requires that each node in the heap have at most one instance of each of the edges given by the decomposition, and does not enforce that all such edges exist. Decomposition heap shapes also do not attempt to describe or reason about the values attached to each edge or require that those edges form part of a meaningful relation under an abstraction function. Decomposition heap shapes retain just enough of the structural invariants of a decomposition to permit reasoning about lock placements.

Symbol	Meaning
$\hat{u}, \hat{v}$	decomposition vertices
$u_i, v_j, \mathcal{V}$	object names
$\hat{f}, \hat{\mathbf{f}}$	abstract fields
$f_i, f_j, \mathbf{f}$	concrete fields
$e ::= \mathsf{nil} \mid v_i$	expressions
$\omega ::= u_i.f \mapsto e$	heap assertions
$\psi \subseteq 2^{\hat{\mathbf{f}}} \to 2^{\hat{\mathbf{f}}}$	placements
$t ::= \mathsf{write}(u_i.f, e) \mid \mathsf{read}(u_i.f) = e \mid \mathsf{observe}(u_i.f) = e$ $\mid v_i = \mathsf{new}\ \hat{v} \mid \mathsf{lock}\ v_i \mid \mathsf{unlock}\ v_i$	transaction operations

Figure 3.11: Transactions on decomposition heaps.

Figure 3.10(b) shows an instance of the decomposition in Figure 3.10(a). Every edge $\hat{f}$ from a vertex $\hat{u}$ to a vertex $\hat{v}$ of the decomposition has a corresponding set of edges $\{f_1, f_2, \cdots\}$ outgoing from any instance u_i of $\hat{u}$ in the corresponding decomposition instance.

The well-lockedness rules defined below quantify over all paths that are a suffix of a particular path $\mathbf{f}$. To keep our transaction-language small, we impose an additional requirement that the set of possible instances f_i of each abstract edge $\hat{f}$ be drawn from a bounded set; that is $i \in \{1, \dots, k\}$ for some k. The bounded set restriction can be lifted by extending the transaction language with an iteration operation that allows a transaction to iterate over all instances of an edge from a vertex; the addition of iteration gives another way for the rules to conclude a fact holds for all instances of an abstract edge $\hat{f}$.

The transaction operations on decomposition heaps are almost identical to the set of operations on trees (Section 3.2), differing only in how objects and fields are named. The transaction operations, shown in Figure 3.11, are: write an expression e (either nil or some v_k to field f_j of object u_i ($\mathsf{write}(u_i.f_j, e)$)), a possibly unstable read of field f_j of object u_i yielding result e ($\mathsf{read}(u_i.f_j) = e$), a stable observation of field f_j of object u_i yielding e ($\mathsf{observe}(u_i.f_j) = e$), allocation of a fresh object of type $\hat{v}$ ($v_i = \mathsf{new}\ \hat{v}$), locking an object ($\mathsf{lock}\ v_i$), and unlocking an object ($\mathsf{unlock}\ v_i$).

3.3.1 Lock Placements

Lock placements are defined exactly as in the tree case: ψ is a function from non-empty heap paths to paths that maps every edge in a heap to an object whose lock protects it. Because edges may now have multiple paths that reach them, a transaction must hold locks on all paths to an edge to perform a stable read or to write the edge.

We now illustrate some of the possibilities for lock placements on decomposition heaps. For our standard first example, by setting

$$\psi_1(\mathbf{f}) = \epsilon \text{ for all } \mathbf{f} \tag{3.5}$$

we can use a single lock at the root of the heap to protect every edge in a decomposition instance. Figure 3.12(a)

Line	**(a)**	**(b)**
1:	lock ρ	$\mathsf{read}(\rho.a_2) = y_2$
2:	$\mathsf{read}(\rho.a_2) = y_2$	lock y_2
3:	$\mathsf{read}(\rho.b_2) = z_2$	$\mathsf{read}(\rho.a_2) = y_2$
4:	$\mathsf{observe}(\rho.a_2) = y_2$	$\mathsf{observe}(\rho.a_2) = y_2$
5:	$\mathsf{observe}(\rho.b_2) = z_2$	$\mathsf{read}(\rho.b_2) = z_2$
6:	$\mathsf{read}(y_2.c_7) = \mathsf{nil}$	lock z_2
7:	$\mathsf{read}(z_2.d_5) = \mathsf{nil}$	$\mathsf{read}(\rho.b_2) = z_2$
8:	$w_4 = \mathsf{new}\ \hat{w}$	$\mathsf{observe}(\rho.b_2) = z_2$
9:	$\mathsf{write}(y_2.c_7, w_4)$	$\mathsf{read}(y_2.c_7) = \mathsf{nil}$
10:	$\mathsf{write}(z_2.d_5, w_4)$	$\mathsf{read}(z_2.c_5) = \mathsf{nil}$
11:	unlock ρ	$w_4 = \mathsf{new}\ \hat{w}$
12:		$\mathsf{write}(y_2.c_7, w_4)$
13:		$\mathsf{write}(z_2.d_5, w_4)$
14:		unlock z_2
15:		unlock y_2

Figure 3.12: Example transactions that add a new node w_4 with access paths a_2c_7 and b_2d_5 to the decomposition heap instance shown in Figure 3.10(b), under **(a)** lock placements ψ_1, defined in Equation (3.5), or ψ_2, defined in Equation (3.6), and **(b)** lock placement ψ_3 defined in Equation (3.7).

shows a well-locked transaction that adds a fresh instance of $\hat{w}$, namely w_4, to the heap of Figure 3.10(b) under lock placement ψ_1. Acquiring the lock on ρ protects the entire heap graph; the transaction then adds w_4 under both the access path a_2c_7 and b_2d_5.

Another possibility is to use the placement

$$\psi_2(\mathbf{f}) = \begin{cases} \epsilon & \text{if } \mathbf{f} \in \{a_i, b_i, a_ic_j, b_id_j\} \\ a_ic_j & \text{if } \mathbf{f} = a_ic_je_k \\ b_id_j & \text{if } \mathbf{f} = b_id_je_k \end{cases} \tag{3.6}$$

which uses a lock at the root to protect instances of edges $\hat{a}$, $\hat{b}$, $\hat{c}$, and $\hat{d}$, and locks at instances of node $\hat{w}$ to protect instances of edge $\hat{e}$. Instances of edge $\hat{e}$ can be reached by two different paths, and thus to observe $\hat{e}$ a transaction must acquire locks on both paths.

Finally, we can use a speculative lock placement. For example, we could protect instances of edges $\hat{a}$ and $\hat{b}$ using speculative locks placed at their targets, and use locks at y and z to protect edges $\hat{c}$, $\hat{d}$, and $\hat{e}$, via the lock placement

$$\psi_3(\mathbf{f}) = a_i \text{ if } a_i \preceq \mathbf{f}, \text{ and } b_j \text{ if } b_j \preceq \mathbf{f} \tag{3.7}$$

Figure 3.12(b) again shows a transaction that adds a fresh instance w_4 of node $\hat{w}$, this time under the speculative lock placement ψ_3.

3.3.2 Well-Locked Transactions

As in the case of tree heaps we represent the state of a transaction using three sets: the local heap Ω, a set of locks L, and the global non-alias set Γ. Sets Ω and L are defined as for trees, but we extend the definition of Γ to DAGs with bounded sharing.

The purpose of Γ is to track objects for which the transaction holds locks on incoming edges. In particular, if a transaction does not hold locks on some incoming edges to an object o, then there may be a path from the global heap to o and the transaction cannot rely on the stability of o's fields. Thus Γ is the transaction's view of the global heap and what other transactions might be able to do to objects of interest to the transaction. The global non-alias set Γ is a mapping from each vertex v_i in the heap to the subset of the incoming edge labels of the decomposition $\mathsf{in}(\hat{v})$ known to be absent from the global heap (i.e., either non-existent or locked by the transaction). We maintain the invariant that in the global heap there is at most one edge to any instance of a decomposition vertex $\hat{v}$ labeled with an instance of each $\hat{f} \in \mathsf{in}(\hat{v})$. If $\Gamma(v_i) = \emptyset$, then v_i may have an instance of each incoming edge in $\mathsf{in}(\hat{f})$ in the global heap. If $\Gamma(v_i) = \{\hat{f}\}$ then v has no incoming edge in the global heap labeled with an instance of $\hat{f}$. If $\Gamma(v) = \mathsf{in}(\hat{v})$ then v has no incoming edges from the global heap.

As before, we hold the lock on an edge reached via a path if we hold the path's corresponding lock placement, restricted to the heap:

$$\mathsf{pathlocked}_{\psi}(\mathbf{f}, \Omega, L) ::= \exists v_i \in L.\ \Omega \vdash \mathbf{g} \sim v_i \ \wedge\ \psi(\mathbf{f})|_{\Omega} = \mathbf{g},$$

where $\psi(\mathbf{f})|_{\Omega}$ is the restriction of path $\psi(\mathbf{f})$ to heap Ω, defined in Section 3.2.2.

The judgement $\Omega, \Gamma \vdash \mathsf{exposed}(x)$ holds if there may be a path to vertex x in the heap that does not lie entirely in the stable observation set Ω; the judgement is defined by the inference rules:

$$\frac{\Gamma(v_k) \neq \mathsf{in}(\hat{v})}{\Omega, \Gamma \vdash \mathsf{exposed}(v_k)} \qquad \frac{\Omega, \Gamma \vdash \mathsf{exposed}(u_i) \wedge (u_i.f_j \mapsto v_k) \in \Omega}{\Omega, \Gamma \vdash \mathsf{exposed}(v_k)}$$

We hold the lock on a field $x.f$ if we hold a lock on that field on every path in the local heap, and there are no paths to x outside the local heap.

$$\mathsf{locked}_{\psi}(v_i.f_j, \Omega, \Gamma, L) \ ::= \ \neg\mathsf{exposed}(v_i) \ \wedge \ \forall \mathbf{g}.\ (\Omega \ \vdash \ \mathbf{g} \ \sim \ v_i \quad \Longrightarrow \quad \mathsf{pathlocked}_{\psi}(\mathbf{g}f_j, \Omega, L))$$

The judgement $\Omega, \Gamma, L \vdash_{\psi} t; \Omega', \Gamma', L'$ defined by the rules in Figure 3.13 describes the class of *well-locked decomposition operations*, analogous to the class of well-locked tree operations of Section 3.2. The judgement holds if a transaction executing operation t under local heap Ω, global non-alias set Γ, and locks L yields an updated local heap Ω', global non-alias set Γ', and lock set L'. The (DNEW) rule states that the fields of a newly allocated object v_i point to nil; furthermore there can be no heap paths to a freshly allocated object so assertions about the fields of v_i are stable and $\Gamma(v_i) = \mathsf{in}(\hat{v})$. The (DLOCK) rule allows a transaction to acquire a lock that it does not hold at any time.

$$\text{(DNEW)}\ \frac{\Omega' = \Omega \cup \{v_i.f_j \mapsto \mathsf{nil} \mid \hat{f} \in \mathsf{out}(\hat{v})\} \quad v_i \notin \operatorname{dom}\Omega \qquad \Gamma' = \Gamma[v_i \mapsto \mathsf{in}(\hat{v})] \qquad v_i \notin \operatorname{dom}\Gamma}{\Omega, \Gamma, L \vdash_\psi v_i = \mathsf{new}\ \hat{v}; \Omega', \Gamma', L}$$

$$\text{(DLOCK)}\ \frac{v_i \notin L}{\Omega, \Gamma, L \vdash_\psi \mathsf{lock}\ v_i; \Omega, \Gamma, L \cup \{v_i\}}$$

$$\text{(DUNLOCK)}\ \frac{v_i \in L \qquad L' = L \setminus \{v_i\} \qquad (\Omega', \Gamma') = \lceil \Omega; \Gamma \mid L'; \psi \rceil \qquad \mathsf{balias}(\Omega, \Omega', \Gamma, \Gamma')}{\Omega, \Gamma, L \vdash_\psi \mathsf{unlock}\ v_i; \Omega', \Gamma', L'}$$

$$\text{(DOBSERVE)}\ \frac{(u_i.f_j \mapsto e) \in \Omega}{\Omega, \Gamma, L \vdash_\psi \mathsf{observe}(u_i.f_j) = e; \Omega, \Gamma, L}$$

$$\text{(DREADUNSTABLE)}\ \frac{u_i.f_j \notin \operatorname{dom}\Omega \qquad \Omega' = \Omega \cup \{u_i.f_j \mapsto e\} \qquad \neg\mathsf{locked}_\psi(u_i.f_j, \Omega', \Gamma, L)}{\Omega, \Gamma, L \vdash_\psi \mathsf{read}(u_i.f_j) = e; \Omega, \Gamma, L}$$

$$\text{(DREADSTABLE)}\ \frac{\begin{array}{c} u_i.f_j \notin \operatorname{dom}\Omega \qquad \Omega' = \Omega \cup \{u_i.f_j \mapsto e\} \\ \mathsf{locked}_\psi(u_i.f_j, \Omega', \Gamma, L) \qquad \Gamma' = \begin{cases} \Gamma & \text{if } e = \mathsf{nil} \\ \Gamma[v_i \mapsto \Gamma(v_i) \cup \{\hat{f}\}] & \text{if } e = v_i \end{cases} \end{array}}{\Omega, \Gamma, L \vdash_\psi \mathsf{read}(u_i.f_j) = e; \Omega', \Gamma', L}$$

$$\text{(DWRITE)}\ \frac{\begin{array}{c} u_i.f_j \in \operatorname{dom}\Omega \qquad \Omega' = \Omega[u_i.f_j \mapsto e] \\ \Big(\forall \mathbf{g}, \mathbf{h}.\ (\Omega \vdash \mathbf{g} \sim u_i) \wedge \mathbf{g}f \preceq \psi(\mathbf{h}) \implies \mathsf{pathlocked}_\psi(\mathbf{h}, \Omega, L) \wedge \mathsf{pathlocked}_\psi(\mathbf{h}, \Omega', L)\Big) \end{array}}{\Omega, \Gamma, L \vdash_\psi \mathsf{write}(u_i.f_j, e); \Omega', \Gamma, L}$$

Figure 3.13: Well-locked decomposition operations: judgement $\Omega, \Gamma, L \vdash_\psi t; \Omega', \Gamma', L'$.

The (DUNLOCK) rule allows a transaction to release any lock that it holds; the rule applies the stabilization operation to remove any newly unstable facts from Ω. Similar to the tree case, the *stabilization* (Ω', Γ') of a local heap Ω_0 and global heap Γ_0 under locks L and placement ψ, written $(\Omega', \Gamma') = \lceil \Omega_0; \Gamma_0 \mid L; \psi \rceil$, is the limit of the monotonically decreasing sequence:

$$\Omega_{i+1} = \{u_j.f_k \mapsto e \in \Omega_i \mid \mathsf{locked}_\psi(u_j.f_k, \Omega_i, \Gamma_i, L)\}$$
$$\Gamma_{i+1} = \Gamma_i \setminus \{v_k \mapsto \hat{f} \mid u_i.f_j \mapsto v_k \in \Omega_i \setminus \Omega_{i+1}\}$$

To ensure that there is at most one instance of any edge label $\hat{f} \in \mathsf{in}(\hat{v})$ in the global heap, the rule requires the *bounded alias* condition

$$\mathsf{balias}(\Omega, \Omega', \Gamma, \Gamma') ::= \forall v_k.\ \left(|\{u_i.f_j \mid (u_i.f_j \mapsto v_k) \in \Omega \setminus \Omega'\}| = \begin{cases} 1 & \text{if } \hat{f} \in \Gamma(v_k) \setminus \Gamma'(v_k) \\ 0 & \text{otherwise.} \end{cases} \right)$$

The bounded alias condition is exactly analogous to the forest condition of Section 3.2.2, and ensures that a transaction may only release an edge with abstract label $\hat{f}$ to a node v_k into the global heap if there are no other edges to v_k labeled $\hat{f}$ in the global heap ($\hat{f} \in \Gamma(v_k) \setminus \Gamma'(v_k)$). Further the condition forbids releasing two pointers with the same label $\hat{f}$ to the same node v_k into the global heap.

Rule (DOBSERVE) states that a transaction may logically observe stable facts about the heap. The (DREADUNSTABLE) rule allows a transaction to read a value speculatively at any time, however unstable reads do not update Ω or Γ. A transaction may perform a stable read of a pointer if it holds the appropriate lock, transferring the pointer from the global heap into Ω and updating Γ accordingly. Finally, a transaction may write to a field if it holds the associated lock, and further holds locks on any edges whose logical/physical mapping may implicitly change as a result of the update.

A transaction $\mathbf{T} = t^1 \dots t^k$ is *well-locked* if there exists a sequence of lock sets L^i, observation sets Ω^i, and global non-alias sets Γ^i such that

$$L^0 = L^k = \emptyset, \quad \Omega^0 = \emptyset, \quad \Gamma^0 = \emptyset, \text{ and } \Omega^{i-1}, \Gamma^{i-1}, L^{i-1} \vdash_\psi t^i; \Omega^i, \Gamma^i, L^i \text{ for } 1 \le i \le k.$$

Lemma 3.5. Let s be a valid schedule of a set of well-locked transactions $\{\mathbf{T}_1, \dots, \mathbf{T}_k\}$. Let Ω_i^j, Γ_i^j, and L_i^j be the set of observations, global non-alias sets, and locks of each transaction after schedule step j. Let h^j be the heap after schedule step j, and suppose the part of h^0 reachable from the root has a decomposition heap shape. Then for all time steps j:

- the lock sets $\{L_i^j\}_{i=1}^k$ are disjoint,
- the observation sets $\{\Omega_i^j\}_{i=1}^k$ are stable, disjoint, and heap h^j is an extension of each $\{\Omega_i^j\}_{i=1}^k$, and
- the global non-alias sets $\{\Gamma_i\}_{i=1}^k$ are disjoint.
- Let heap h be the heap h^j less edges present in the local heaps $\{\Omega_i^j\}_{i=1}^k$. Then for every vertex $v \in h$ and edge label $\hat{f} \in \mathsf{in}(\hat{v})$ either there is exactly one edge labeled with an instance of $\hat{f}$ pointing to v in h, or $\hat{f} \in \Gamma_i^j$ for some i and there are no edges labeled with an instance of $\hat{f}$ pointing to v in h.

Proof. The structure of the proof (by induction on the length of the schedule) is the similar to the corresponding proofs flat-heaps and trees. The case analysis is also almost the same as for trees; we discuss the two cases where there is some variation:

- Rule (DUNLOCK): Disjointness of lock sets is immediate. Stability of the transaction performing the unlock holds by definition, and for other transactions by the induction hypothesis. The disjointness and extension properties on the local heaps hold by the definition of stabilization and the induction hypothesis. The side condition on the definition of stabilization ensures that a transaction may only release a pointer to a vertex v with label f into the heap only if $\hat{f} \in \Gamma(v)$, and hence the condition on edges in the heap holds.

- Rule (DReadStable): The lock set result is immediate. Stability of the observation sets holds by the induction hypothesis and because the rule requires $\mathsf{locked}_\psi(u_i.f_j, \Omega_1^{n+1}, \Gamma_1^{n+1}, L_1^{n+1})$ as a precondition. The induction hypothesis guarantees that transactions have disjoint observation sets; furthermore the disjointness of the Γ sets guarantees that only one transaction can hold the lock on $u_i.f_j$ at any time. The result follows from the induction hypothesis and the definition of stabilization.

□

Finally, we have a logical serializability lemma similar to Lemma 3.2 and Lemma 3.4.

Lemma 3.6. Any valid schedule of a set of well-locked, logically two-phase decomposition transactions $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$ is conflict-equivalent to a serial logical schedule.

And, again, as in the flat heap case, these results can be extended to shared/exclusive locks using the approach in Section 3.1.5.

3.4 Related Work

Two-phase locking was originally introduced in the context of transactions operating over abstract entities, each with its own associated lock [Eswaran et al., 1976]. The core technical idea of this chapter is that we can use two-phase locking to show serializability of a wide class of locking strategies by adding a layer of indirection between logical locks, which are the entities that are the subject of the original two-phase locking protocol, and the physical locks that implement them. The original paper on two-phase locking made explicit the idea of locking not just objects, but program predicates [Eswaran et al., 1976]. Other authors have explored using predicate locks; for example, Deshmukh et al. [2010] explored deriving predicate locks from sequential correctness proofs. Logical locks are also examples of predicate locks, which we realize in practice by mapping logical locks onto physical locks using a lock placement.

Various authors have investigated techniques for inferring locks to implement atomic sections [McCloskey et al., 2006; Hicks et al., 2006; Emmi et al., 2007; Halpert et al., 2007; Cherem et al., 2008; Cunningham et al., 2008; Zhang et al., 2008]. A related problem is automatically optimizing programs with explicit locking by combining multiple locks into one [Diniz and Rinard, 1998]. A key part of this class of work is constructing a mapping from program objects to the locks that protect them, similar to our lock placement language. Most existing work only considers very simple mappings between data and locks; typically data is protected either by global locks or by locks on the object which contains the data. The lock placements we propose are much more flexible; in particular existing formalisms cannot handle the class of path placements we propose in this chapter, such as speculative locks, or lock placements that vary with heap updates. Our work provides a more general framework for describing the relationship between locks and data, which may be useful to researchers exploring automatic lock inference.

A novel feature of the infrastructure we develop in this chapter is that it allows us to reason about speculative lock placements. Speculative locking is used in highly concurrent libraries and in the context of software transactional memory [Bronson et al., 2010a; Bronson, 2011; Bronson et al., 2010b]. Although we present our ideas in the context of speculative placements of pessimistic locks, the idea of a lock placement can also be used to reason about the safety and consistency of speculative placements of optimistic STM metadata as used in practice.

A variety of locking protocols have been proposed in the literature that extend two-phase locking to handle dynamically changing heaps and to allow early release. Notable examples include the dynamic tree locking and dynamic DAG locking protocols [Attiya et al., 2010], and domination locking [Golan-Gueta et al., 2011]. Unlike existing protocols which use the lock on each object to protect that object's fields, our work explores a more flexible space of mappings between locks and data, where locks and data need not be adjacent. A key concern of many locking protocols is early release, which allows a transaction to release locks earlier than would otherwise be permitted by two-phase locking. We do not consider early release in our work; however early release is orthogonal to lock placement, and lock placements could be profitably combined with techniques for early release.

The concept of a stable set and stabilization is related to rely-guarantee logic [Jones, 1981] and its subsequent developments [Wickerson et al., 2010]. Concurrent extensions of separation logic, such as Concurrent Separation Logic [O'Hearn, 2007], RGSep [Vafeiadis and Parkinson, 2007] and work on storable locks [Gotsman et al., 2007] allow local reasoning about programs with shared mutable state that is accessed concurrently. Our work complements work on direct reasoning about concurrent code; we propose a locking protocol, parameterized by a declarative lock placement, by which we can show conflict-serializability, thereby removing the need for direct concurrent reasoning for programs that obey the locking protocol.

Chapter 4

Concurrent Data Representation Synthesis

In this chapter, we extend the data representation synthesis approach of Chapter 2 to support concurrency, leveraging using lock placements of Chapter 3 to describe the space of possible locking strategies.

Consider the problem of implementing concurrent operations on a directed graph. We must decide how to represent the graph as a collection of data structures, perhaps using a lookup table mapping each node to the set of its adjacent nodes. We will need to pick concrete representations for both the lookup table (e.g., a concurrent hashmap) and the adjacency sets (e.g., linked lists). We must also decide how concurrency will be realized. We could add our own synchronization using locks and/or we could use a concurrent container data structure to implement the lookup table, the sets of adjacent nodes, or both.

Assume for the moment that we decide both containers will be concurrent. We must of course ensure there is enough synchronization to avoid harmful races, but not so much that we either limit scalability or introduce deadlocks. Using off-the-shelf concurrent containers can simplify this task, but even using concurrent containers for both data structures does not automatically imply that high-level graph operations that touch both structures (such as inserting or removing an edge from the graph) are correct. In fact, recent work in bug detection for concurrent programs has shown that programmers fail to use standard concurrent containers correctly as often as 44% of the time, especially when they must compose multiple concurrent operations [Shacham et al., 2011].

On the other hand, it may be more efficient to have only the top-level lookup table be concurrent and use non-concurrent data structures for the sets of adjacent nodes—if it is very infrequent that threads try to access the same node simultaneously the extra overhead of a concurrent data structure for the adjacency sets won't be worthwhile. This design has different correctness requirements and would likely result in a different choice of where to place any needed synchronization to guarantee correctness. The right answer to the decision of whether to use a concurrent or non-concurrent data structure for the adjacency sets likely depends on the typical workload and it will be difficult to modify the interlinked synchronization and data structures if we decide later that the graph should be implemented differently.

In this chapter we present an approach to synthesizing concurrent data representations, meaning that from a high-level specification of data we produce both the concrete data structures and the corresponding synchronization to implement the specification. In our approach, programs are written using *concurrent relations* (Section 4.1), a generalization of standard concurrent collections to relations with a concurrent interface to perform insertions, deletions, and lookups of tuples. Our compiler automatically synthesizes all aspects of the data representation, including the choice of data structures and how they interact, the number and placement of locks to guard access to those data structures (including, for example, whether locking should be fine-grain or coarse-grain), an order in which locks can be acquired to guarantee deadlock freedom, and all of the code to correctly manage the interplay of the data structures and synchronization.

Our method builds on the results presented in the two previous chapters (Section 4.3): we use the *decompositions* of Chapter 2 to describe how relations can be decomposed into a set of cooperating data structures, and we use the theory of *lock placements* of Chapter 3 to describe the space of possible locking strategies. The specific contributions of this chapter are:

- We introduce *concurrent relations*, a generalization of standard concurrent container data structures to containers of tuples, with concurrent operations to insert, remove, and query relations (Section 4.1).
- The selection of data structures is subtler than in the non-concurrent case, because there is the added dimension of using concurrent container structures, which may or may not require additional synchronization depending on the relational specification, and, in addition, different concurrent containers provide varying guarantees about the safety of concurrent access. We give a taxonomy of containers and their properties relevant to concurrent data representation synthesis (Section 4.2).
- We extend the relational decomposition language of Chapter 2 to support concurrent relations. Just as the original decomposition language describes how to assemble a representation of a relation from a library of container data structures, concurrent decompositions describe how to compose concurrent and non-concurrent data structures together with locks to implement a concurrent relation primitive (Section 4.3).
- We show how to integrate the *lock placements* of Chapter 3, which describe a space of possible locking strategies on data structures, with the problem of selecting the data structures themselves. The choice of data structures and lock placements is done in such a way that the resulting code is guaranteed to ensure the serializability of relational operations (Section 4.3.1).
- We adapt and generalize the problem of selecting a good implementation of the relational primitives, called *query planning*, to concurrent relations (Section 4.4). One of the major issues is ensuring deadlock freedom, which we accomplish by selecting a global lock ordering that all relational operations obey by construction. We did not previously address deadlock in Chapter 3.

- The optimal decomposition depends on the usage patterns of the data structure and the target machine. We present results from a full implementation, which includes an autotuner that allows us to discover a good combination of both locks and container data structures automatically for a training workload. We perform an evaluation of a concurrent graph benchmark, showing that the best data representation varies with the workload, and thus it is important to have the flexibility to easily alter the representation of concurrent data (Section 4.5).

4.1 Concurrent Relations

We provide five operations for creating and manipulating *concurrent relations*: empty, remove, update, query, and insert. The relational compiler ensures that the implementations of all relational operations are *linearizable* [Herlihy and Wing, 1990] (equivalently *serializable*, since the relational operations are single operation transactions on a single object). Operations on an object are *linearizable* if every operation appears to take place atomically at a single point in time in between its invocation and response.

The first four operations have the same mathematical specification as their non-concurrent counterparts defined in Section 2.1. Recall that empty () creates a new empty relation. Operation remove $r\ s$ removes tuples matching s; in the concurrent case our implementation requires that s be a key for the relation. Operation update $r\ s\ u$ applies the updates in tuple u to the tuple matching s in relation r, if any; s must be a key for the relation. Finally, operation query $r\ s\ C$ returns columns C of all tuples in r matching tuple s.

The only operation with a different specification from its non-concurrent counterpart is the insertion operation,

$$\mathsf{insert}\ r\ s\ t = \mathsf{if}\ \nexists u.\, u \in\ !r \land s \subseteq u\ \mathsf{then}\ r \leftarrow\ !r \cup \{s \cup t\},$$

which inserts a new tuple x, where x is the union of the columns of tuples s and t, into a relation r, provided there is no existing tuple in r matching s. We require that s and t have disjoint domains.

Insert generalizes the put-if-absent operation provided by standard concurrent key-value maps: *put-if-absent(k,v)* inserts value v into the map if no other value is already associated with key k, and would be written

$$\mathsf{insert}\ r\ \langle key\colon k\rangle\ \langle value\colon v\rangle$$

As in the non-concurrent case insert operations may violate functional dependencies, and it is the client's obligation to ensure functional dependencies are observed. In the non-concurrent case, a client could perform a query to check whether a functional dependency would be violated by a subsequent insertion; this approach is not possible in a concurrent setting since a concurrent thread could change the relation between the query and the subsequent insert. The revised form of the insert operation allows clients to atomically test whether functional dependencies will be satisfied by a new tuple even in the presence of concurrent updates.

For example, consider a relation representing the edges of a directed graph, with columns $\{src, dst, weight\}$

satisfying the functional dependency $src, dst \to weight$. We can create a new, empty graph relation r_0 using the empty () operation. Inserting an edge

$$\mathsf{insert}\ r_0\ \langle src\colon 1,\ dst\colon 2\rangle\ \langle weight\colon 42\rangle$$

results in a new relation $r_1 = \{\langle src\colon 1,\ dst\colon 2,\ weight\colon 42\rangle\}$. A subsequent insertion

$$\mathsf{insert}\ r_1\ \langle src\colon 1,\ dst\colon 2\rangle\ \langle weight\colon 101\rangle$$

leaves the relation unchanged, because relation r_1 already contains an edge with the same src and dst fields.

4.2 A Taxonomy of Concurrent Containers

Concurrent decompositions are an extension of the non-concurrent decompositions of Chapter 2, and describe how to implement concurrent relations as a combination of both concurrent and non-concurrent data structures. Before diving into the details of the concurrent decomposition language (Section 4.3), we first describe the concurrency properties of the container data structures found in the wild, which form the building blocks of concurrent decompositions.

Container Interface A *container* is a data structure that implements an associative key-value map interface consisting of *read operations* $\mathsf{lookup}(k)$ and $\mathsf{scan}(f)$, and a *write operation* $\mathsf{write}(k, v)$.

- The *lookup* operation $\mathsf{lookup}(k)$ returns the value associated with a key k, if any.
- The *scan* operation $\mathsf{scan}(f)$ iterates over the map, and invokes the function $f(k, v)$ once for each key k and its associated value v in the map. A scan may or may not return the entries of the map in sorted order.
- The *write* operation $\mathsf{write}(k, v)$ sets the value associated with a key k to v. Here v is an optional value, in the style of ML. If v is $\mathsf{Some}\ \ w$, then the operation updates the value associated with key k to w, whereas if v is None, representing the absence of a value, then any existing value associated with k is removed. The write operation subsumes operations to insert, update, and remove entries from a map.

4.2.1 Concurrency Safety and Consistency

We next discuss two related properties of containers, *concurrency safety*, which describes whether it is safe for two operations to occur in parallel, and *consistency*, which characterizes what a container guarantees about the possible orders of events in a concurrent execution. Table 4.1 lists the concurrency safety and consistency properties of a selection of Java containers from the JDK; as we show, containers differ greatly in their support for concurrency.

Data Structure	**Concurrency-safety**			
	lookup/lookup lookup/scan scan/scan	lookup/write	scan/write	write/write
`HashMap`	yes	no	no	no
`TreeMap`	yes	no	no	no
`ConcurrentHashMap`	yes	yes	weak	yes
`ConcurrentSkipListMap`	yes	yes	weak	yes
`CopyOnWriteArrayList`	yes	yes	yes	yes

Table 4.1: Concurrency safety properties of selected containers from the JDK. Possible operations are lookup, scan, or write. For an operation pair α/β, concurrently executing operations α and β on a container is either unsafe ("no"), safe but only weakly consistent ("weak"), or both safe and linearizable ("yes").

Concurrency Safety For a given data structure, we say a pair of operations α/β is *concurrency-safe* if two threads may safely execute operations α and β in parallel with no external synchronization. A container is *concurrency-safe* if all pairs of operations are concurrency-safe. Concurrency safety is strictly a statement about the correct usage of the interface of a data structure; it is irrelevant how the data structure guarantees safety internally, whether by locks, atomic instructions, or by some other means.

Consider the data structures described in Table 4.1. Almost all data structures support parallel read operations; for example concurrent threads may safely read or iterate over a Java `HashMap` in parallel without synchronization. Exceptions exist; for example, it would not be safe for threads to perform concurrent reads of a splay tree because splay tree read operations rebalance the tree.

Only a few containers permit write operations in parallel with other operations. For example, it is unsafe to read from or write to a `HashMap` object while another thread is writing to the same `HashMap`. By contrast a `ConcurrentHashMap` or a `CopyOnWriteArrayList` allow concurrent lookup and write operations, or pairs of concurrent write operations. On a concurrency-safe container, such as a `ConcurrentHashMap`, the lookup and write operations are linearizable even in the absence of any external concurrency control. For concurrency-unsafe operations, such as reading a splay tree, linearizability is the responsibility of an external concurrency control primitive, such as a lock.

The scan operation, however, behaves differently. Even many containers that allow iteration in parallel with mutation do not guarantee that iteration is linearizable. We identify two different possibilities. Some containers, such as `ConcurrentHashMap` provide *weakly consistent* concurrent iteration; that is, concurrent iteration is safe, but may or may not reflect inserts and removals that occur in parallel with the iteration. Iteration over a weakly-consistent container may not be linearizable, that is, the result of the iteration may not correspond to the set of entries present in the container at any instant in time. Conversely for containers that provide *snapshot iteration*, such as a `CopyOnWriteArrayList`, iteration behaves as if it operated over a linearizable snapshot of the container.

4.3 Concurrent Decompositions

The concurrent decomposition language describes how to assemble container data structures into representations of relations that support concurrent serializable transactions that implement the various relational operations. By combining concurrent and non-concurrent data structures with locks we can build a representation of a relation with strong concurrency guarantees, even if the constituent data structures themselves have limited support for concurrency.

We extend the relational decomposition language of Chapter 2 to support concurrent operations from multiple threads. Two key ideas underlie safe and scalable concurrent decompositions: leveraging existing concurrent containers to the full extent possible, and supplementing containers with locks as necessary to ensure the safety and serializability of concurrent transactions over the complete decomposition. Formally, a *concurrent decomposition* is a decomposition extended with a *lock placement* that describes the number and placement of locks, and the association between locks and data (Section 4.3.1).

We use Figure 4.1(a) as a running example, which shows a decomposition for a filesystem directory tree relation, based on the directory entry cache in the Linux kernel. The relation has three columns parent, name, and child and obeys a functional dependency $\mathit{parent}, \mathit{name} \to \mathit{child}$. Each 'parent' directory entry has zero or more 'child' directory entries, each with a distinct file 'name'.

In Figure 4.1(a), the edge ρx from the root indicates that the relation is implemented by a `TreeMap` from each parent value to the residual relation of all $(\mathit{name}, \mathit{child})$ pairs for that parent. Recursively, this subrelation is implemented by another `TreeMap` from name to the child directory (edge xy). Finally, the functional dependency guarantees that the child directory is a singleton tuple and is implemented by its single value (edge yy). This structure (a map from parents to the set of child directory names) enables efficient iteration over the children of a directory, which is useful when, for example, unmounting a filesystem. To enable efficient directory lookup the decomposition also includes a global hashtable mapping $(\mathit{parent}, \mathit{name})$ pairs to child objects (edge ρy).

Figure 4.1(b) depicts an instance of the decomposition of Figure 4.1(a) representing the relation containing three directory entries:

$$\begin{aligned} \{\, &\langle \mathit{parent}\colon 1,\ \mathit{name}\colon \text{`a'},\ \mathit{child}\colon 2\rangle\,, \\ &\langle \mathit{parent}\colon 2,\ \mathit{name}\colon \text{`b'},\ \mathit{child}\colon 3\rangle\,, \\ &\langle \mathit{parent}\colon 2,\ \mathit{name}\colon \text{`c'},\ \mathit{child}\colon 4\rangle\}. \end{aligned}$$

4.3.1 Logical Locks, Transactions, and Serializability

Locks Given a decomposition d, we compile each relational operation into a transaction tailored to d. For safety and consistency transactions must acquire locks that protect the invariants upon which a transaction relies. By "lock" here we mean a class of pessimistic synchronization primitives that may be held by a transaction in either of two different *modes*, namely *shared* or *exclusive*.

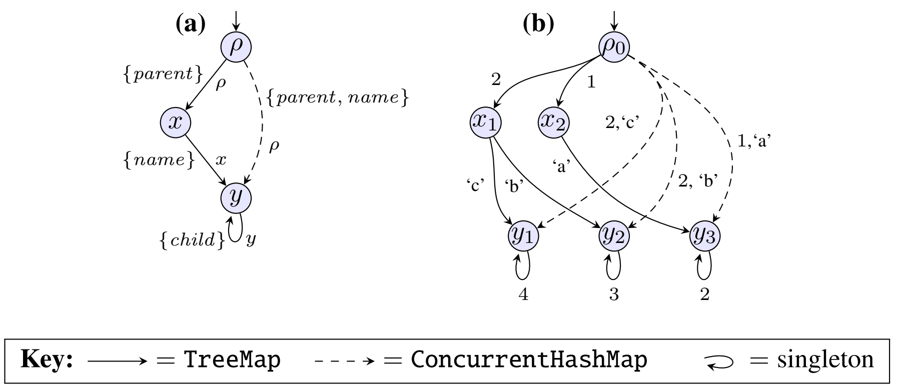


Figure 4.1: **(a)**: A decomposition representing a directory tree relation with three columns: $parent$, $name$, and $child$; and **(b)**: an instance of decomposition (a). Each edge of the decomposition is labeled with a set of columns $\{\cdots\}$, together with the label of the node whose lock protects instances of that edge (Section 4.3.2). Each edge of the instance is labeled with a valuation for the corresponding decomposition edge's columns. Solid edges indicate a `TreeMap`, dashed edges represent a `ConcurrentHashMap`, and self-loops represent singleton tuples.

Logical Locks To ensure that transactions are serializable, the data in decomposition instances are protected by *logical locks*. We associate a distinct logical lock with every edge uv_t of a decomposition instance. Logical locks protect the state, either presence or absence, of an edge instance. If a transaction observes the presence or absence of an edge it must hold shared access to the corresponding logical lock, and if a transaction adds or removes an edge it must hold exclusive access to the corresponding logical lock. Logical locks are defined for every possible edge instance, irrespective of whether the edge is actually present in a particular decomposition instance or not.

For now, we leave the implementation of each logical lock uv_t abstract. In Section 4.3.2, we implement logical locks using a smaller set of *physical locks* attached to the nodes of a decomposition instance. By placing restrictions on the possible mappings from logical locks to physical locks we can ensure that containers (concurrent or not) and compositions of containers are used safely.

4.3.2 Physical Locks and Lock Placements

By associating a unique logical lock with every edge instance we can use two-phase locking to ensure that transactions are serializable. Such an approach would be impractical to implement directly, however. Each edge instance corresponds to a entry in a container in the heap, and it would often be too slow to actually use locks at such a fine granularity, not to mention the practical problem that there are infinitely many logical locks defined for container entries that are absent. Further, as shown in Table 4.1, in practice different containers have different levels of support for safe concurrency, and so while we must use locks to protect some containers from all concurrent accesses, in other cases we can rely on the container to mediate concurrent access. Finally, since we treat container implementations as black boxes, we have no way to attach locks to the edge instances directly.

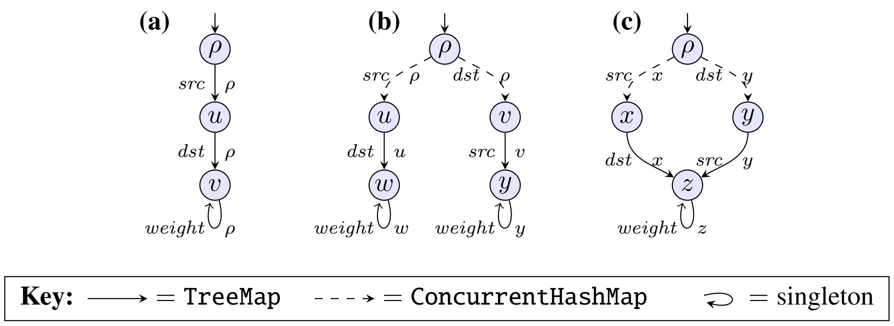


Figure 4.2: Three concurrent decompositions for a directed graph relation: (a) a "stick", with a single coarse lock around non-concurrent data structures, (b) a "split" decomposition, with locks at different granularities, and (c) a "diamond", with a mixture of speculatively-locked concurrent data structures and non-concurrent data structures. Solid edges indicate `TreeMap` containers, dashed edges represent `ConcurrentHashMap` containers, and self-loops represent singleton tuples. Each edge is labeled with a set of columns on the left and the associated lock placement on the right.

Instead of literally maintaining one lock for every possible edge instance, we implement logical locks using a smaller set of *physical locks* attached to instances of nodes in a decomposition. We describe the correspondence between logical and physical locks using a *lock placement*, introduced in Chapter 3, which is a mapping from the set of logical locks onto the set of physical locks. Many logical locks may map onto the same physical lock, and acquiring a physical lock corresponds to taking all of the corresponding logical locks. By choosing different lock placements we can describe different granularities of locking. Since physical locks are attached to node instances, in general there may be an unbounded number of physical locks.

Physical Locks To each node v in a decomposition we attach a set of physical locks $\{v^0, v^1, \dots\}$. If there is only a single physical lock attached to a node we simply write v for both the node and its unique physical lock. Having multiple physical locks per node allows us to protect different outgoing edges from a node with different locks, reducing contention.

Lock Placements A *lock placement* ψ is a function mapping the logical lock associated with each edge onto a physical lock on a node that implements it. We define lock placements on the (static) decomposition which we extend to (dynamic) decomposition instances in the obvious way; if a lock placement ψ maps the logical lock on edge e to a physical lock on node v at compile time, then at runtime the compiler maps the logical lock on edge instance e_t onto the physical lock on the corresponding node instance v_t.

Recall the directed graph example from Section 4.1. The relation in question has three columns: src, dst, and weight, related via the functional dependency $\mathit{src}, \mathit{dst} \to \mathit{weight}$. Figure 4.2 shows three possible decompositions, each with a different choice of data structures and lock placements.

Figure 4.2(a) uses a coarse-grain lock placement

$$\psi_1(e) = \rho \text{ for all edges } e,$$

which protects all edges of the decomposition using a single lock at the root ρ. Since there is only one instance of the root node ρ in any decomposition instance, the same lock is used to protect everything. Further, since the logical locks of all edge instances are mapped to the same physical lock ρ, the lock serializes access to the entire decomposition data structure, ensuring that each (non-concurrent) `TreeMap` is only accessed by one transaction at a time.

Figure 4.2(b) depicts a fine-grain locking strategy decomposition in which each edge is protected by a lock at its head (i.e., objects in a container are protected by a single lock on the container itself), using the lock placement

$$\psi_2(\alpha\beta) = \alpha \text{ for all edges } \alpha\beta.$$

Edges ρu and ρv are protected by a lock at ρ, whereas edges uw, vy, wx and yz are protected by locks at u, v, w and y, respectively.

Both of the example lock placements described so far use a single lock to protect all the entries in each container. Figure 4.2(c) makes use of *speculative locking* (Section 4.3.4), one of two extensions which allow different dynamic instances of an edge in the same container to be protected by different locks. We defer further discussion of this example to Section 4.3.4.

Well-Formed Lock Placements We require that all lock placements satisfy the following conditions:

- The lock placement $\psi(uv)$ of each edge uv either must dominate the edge's source vertex u or be equal to v. (The latter case occurs in the speculatively-placed locks of Section 4.3.4.) By definition this condition ensures that the lock placement for an edge lies on every path from the root of the decomposition including that edge. This condition ensures the instance of the node $\psi(uv)$ named by the lock placement is unique for each edge instance uv_t. The domination requirement also simplifies query planning (Section 4.4), since it ensures that a query plan will always encounter the necessary locks for each edge no matter how the edge is reached.

- All edges between an edge and its lock placement share the same placement. That is, fix any edge uv and take any edge xy in a path in the decomposition from $\psi(uv)$ to u. Then we have $\psi(xy) = \psi(uv)$. This requirement ensures that if a lock protects an edge, then the lock also protects the path from the lock to that edge, thereby ensuring that if we hold a lock then the set of edges protected by that lock cannot change.

Logical Lock Implication Since we implement logical locks by mapping them onto a smaller set of physical locks, a transaction cannot acquire logical locks directly. Instead, a transaction must acquire physical locks that *imply* access to the logical locks that the transaction requires.

We say that a set of physical locks P held by a transaction *imply* exclusive or shared access, respectively, to the logical lock of edge instance uv_t under lock placement ψ if:

- the transaction holds exclusive or shared access, respectively, to the corresponding physical lock, that is, if $\psi(uv) = x$ then $x_t \in P$, and
- the mapping between the logical lock to the corresponding physical lock is stable, that is, there exists a path $\mathbf{w}_t$ from the root of the decomposition instance to v_t such that the transaction holds shared access to every edge in $\mathbf{w}_t$.

The stability criterion means that a physical lock only covers a logical lock if the transaction also holds locks that guarantee that the logical lock does in fact correspond to that physical lock; if not, a concurrent transaction might alter the heap and change the association between logical and physical locks. For example, consider a concurrent hashtable where the elements of each hash bucket are guarded by a lock. If a transaction moves an element v from bucket b_1 to bucket b_2 the lock guarding access to v changes, and any transaction that was concurrently accessing v by acquiring the lock on b_1 no longer holds the correct lock for v for the lock placement. Thus, in the presence of updates that can change the structure of the heap, it is not sufficient to just hold the locks L guarding access to the particular data, but it is also necessary to hold locks on whatever portion of the heap structure guarantees that L remains the correct set of locks to hold!

Since lock placements are defined using a decomposition structure, for locking using a placement to be well-defined we must ensure that transactions always yield heap states that are valid instances of the corresponding decomposition. One of the benefits of data representation synthesis is that we are guaranteed that the operations emitted by the compiler preserve the decomposition structure by construction.

4.3.3 Lock Striping

Lock striping is a technique for boosting the throughput of a transaction by using a set of locks instead of a single lock. Consider again the decomposition of Figure 4.2(b), in which the lock placement maps the logical lock on each edge to a physical lock at the source of the edge. While this lock placement ensures safe and consistent transactions, by protecting each container with a single lock we serialize access to containers, and hence we cannot make effective use of concurrent containers such as `ConcurrentHashMap`. To leverage concurrent containers we can partition the elements of the container into a number of *stripes*, each with its own lock.

For example, in Figure 4.2(b), rather than mapping all instances of edges ρu and ρv to a single physical lock at node ρ, we can use k physical locks $\rho^0, \dots \rho^{k-1}$. We then use a lock placement that stripes the logical locks attached to instances of ρu and ρv across the k physical locks:

$$\psi_3(e,t) = \begin{cases} \rho^i & \text{if } e = \rho u,\ i = t(\mathit{src}) \bmod k \\ \rho^i & \text{if } e = \rho v,\ i = t(\mathit{dst}) \bmod k \\ \alpha_t & \text{otherwise, where } e = \alpha\beta \end{cases} \tag{4.1}$$

Since different instances of the same edge are mapped onto different physical locks, the lock placement takes as input both an edge e and a tuple t identifying a particular edge instance e_t; the fields of tuple t are used to select one of the k physical locks at ρ. If we do not know the relevant tuple fields in advance, for example if we want to iterate over the container, we can always conservatively take all k locks.

Lock striping is only applicable for containers that are concurrency-safe. For a concurrency-unsafe container, such as a `TreeMap`, we are limited to at most one lock for the entire container to ensure that no two threads access the container concurrently.

By increasing the value k we can reduce lock contention to arbitrarily low levels, at the cost of making operations such as iteration that access the entire container more expensive.

4.3.4 Speculative Lock Placements

When striping logical locks across physical locks, as the number of physical locks k increases in the limit each container entry has its own individual lock. Rather than preallocating locks for an unbounded number of objects, we can achieve this limiting case more efficiently by using *speculative locking* as described in Section 3.1.2. Speculative locking lazily constructs a unique physical lock for each logical lock.

The key to speculative locking is the identity of the lock that protects an edge instance depends on the state of the edge instance itself. We map the logical lock to a distinct physical lock for each edge instance present in a container by placing the lock in the node that is the target of the edge instance. For serializability the lock placement must also be defined for edge instances that are absent from the decomposition, not just those edges that are present. Since we cannot place locks for non-existent edge instances at the target of the edge, instead we map the logical locks for absent edges onto physical locks at the edge's source.

Up to this point, we have required that the lock guarding an edge e appear on all paths from the root before e is reached. For speculative locks, this invariant does not hold—we do not know what lock to acquire until we have reached the object we wish to protect. The key is that it is safe to perform unlocked reads of a concurrency-safe container to guess the identity of the lock that we should acquire. Since the container is concurrency-safe, reading without holding a lock is safe, however we have no guarantees that any information that we read will remain stable. Once we have guessed and acquired a lock, we can check to see if our guess was correct. There are two possibilities — either we guessed correctly, in which case we already held the lock that protects the edge and our read was stable, or we guessed incorrectly, in which case the edge must point somewhere else. In the latter case we can release the lock we guessed and try again. While speculatively acquiring a lock is not physically two-phase, a transaction can be viewed as acquiring logical locks in a two phase manner (Chapter 3).

Speculative lock acquisition differs from the well-known but broken double-checked locking idiom in two key ways—firstly, we always acquire a lock and recheck reads under that lock, and secondly we require that concurrent containers are linearizable, that is, with semantics analogous to a Java volatile field.

For example, the decomposition depicted in Figure 4.2(c) uses a mixture of both speculative and non-speculative locking — in particular, the locks that protect edges ρx and ρy are placed at the target of each edge on nodes x and y respectively. To take a lock on an edge instance ρx_t a transaction must first speculatively lookup entry t in the map without locking, acquire the lock on ρ or x_t if the edge instance is absent or present, respectively, and then verify that the correct lock was taken. The data structure implementing edge ρx is a `ConcurrentHashMap`, which is concurrency-safe, so it is safe to speculatively read an edge without holding its lock. Formally, we can express the speculative locking policy as the lock placement

$$\psi_4(e,t) = \begin{cases} u_t & \text{if } e = \rho u,\ e_t \text{ is present} \\ \rho^i & \text{if } e = \rho u,\ e_t \text{ is not present},\ i = t(\mathit{src}) \bmod k \\ v_t & \text{if } e = \rho v,\ e_t \text{ is present} \\ \rho^i & \text{if } e = \rho v,\ e_t \text{ is not present},\ i = t(\mathit{dst}) \bmod k \\ \alpha_t & \text{otherwise, where } e = \alpha\beta. \end{cases}$$

For simplicity, the diagram in Figure 4.2(c) does not show the striping of absent locks.

4.4 Query Planning and Lock Ordering

In Section 4.3 we introduced concurrent decompositions, which describe a relational specification using both concurrent and non-concurrent containers in combination with locks. In this section we show how to compile the relational operations of Section 4.1 into code tailored to a particular concurrent decomposition.

Chapter 2 described how to compile relational operations in a non-concurrent context. There are two additional complications we must deal with when generating concurrent implementations of relational operations—we must ensure that a transaction takes the locks that protect the decomposition edges it touches, and we must ensure that transactions are deadlock-free.

4.4.1 Deadlock-Freedom and Lock Ordering

A common strategy for ensuring that a set of concurrent transactions is deadlock-free is to impose a total order on locks. If all transactions acquire locks in ascending lock order, then we are guaranteed that concurrent transactions are deadlock-free.

We ensure deadlock-freedom for concurrent decomposition operations by imposing a total lock order on the physical locks of a decomposition; it is the responsibility of the query planner to generate code that respects this order.

All query plans must obey a single static order on all possible physical locks of a decomposition. The precise set of physical locks in existence may change as we allocate and deallocate node instances, but the relative order of any given pair of physical locks never changes during runtime. We order physical locks firstly on a topological sort of the decomposition nodes to which they belong. We order different instances of the

$$q ::= x \mid \mathsf{let}\ x = q_1\ \mathsf{in}\ q_2 \mid \mathsf{lock}(q, v) \mid \mathsf{unlock}(q, v) \mid \mathsf{scan}(q, uv) \mid \mathsf{lookup}(q, uv)$$

Figure 4.3: The concurrent query language. We only show the fragment necessary for implementing query operations.

same node lexicographically on the values of the key columns. Finally, we order the physical locks attached to each node instance by number.

For example, consider the decomposition of Figure 4.2(c). We fix a topological order of the nodes, say

$$\rho < x < y < z < w;$$

meaning that all locks attached to ρ are ordered before all locks attached to instances of x, and so on. We lift the topological order on nodes to a total order on node instances

$$\rho < x_{s_0} < x_{s_1} < \cdots < y_{t_0} < y_{t_1} < \cdots ,$$

where the tuple sequences (s_i) and (t_i) are in lexicographic order. Finally, since there may be more than one physical lock per node due to lock striping, we lift the total order on node instances to a total order on physical locks:

$$\rho^0 < \rho^1 < \cdots < x^0_{s_0} < x^1_{s_0} < \cdots < x^0_{s_1} < x^1_{s_1} \cdots .$$

As an aside, it is necessary that we totally order the physical locks of a decomposition, not the logical locks; a query only acquires physical locks directly and it is the order of those physical locks that is pertinent to deadlock.

4.4.2 Query Language

Once we have fixed a total order on the physical locks of a decomposition, the query planner must generate well-locked, two-phase code that respects the lock order for each possible query.

A key requirement of a query plan is that it must make explicit which locks to acquire and in which order. The query trees in Chapter 2 are not suitable for reasoning about locks since they have no notion of sequencing of expressions.

In the concurrent setting, we extend query trees to a fragment of a strict, impure functional language, shown in Figure 4.3. The let-binding construct of the concurrent query language can describe the order of execution of operations with side-effects, in particular lock and unlock operations. A query expression, denoted using the metavariable q, is one of: a variable reference x, let-binding $\mathsf{let}\ x = q_1\ \mathsf{in}\ q_2$, a lock acquisition $\mathsf{lock}(q, v)$, a lock release $\mathsf{unlock}(q, v)$, an edge lookup $\mathsf{lookup}(q, v)$, or an edge iteration $\mathsf{scan}(q, v)$. We discuss the semantics of expressions shortly.

Query States Evaluating any expression in the query language yields a set of *query states*. A *query state* is a pair (t, m) of a tuple t containing a subset of the relation's columns, together with a mapping m from decomposition nodes v to the corresponding node instance v_t. If a vertex v with type $v\colon A \triangleright B$ appears in the domain of m, tuple t must contain sufficient columns such that v_t is uniquely defined, that is, $A \subseteq \operatorname{dom} t$.

For example, a query state that might occur when evaluating a query over the decomposition instance of Figure 4.1(b) is

$$(\langle \mathit{parent}\colon 1,\ \mathit{name}\colon \text{‘a’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\}).$$

In this example, column parent is bound to value 1 and column name is bound to the string ‘a’. The query state also identifies a set of node instances, namely the instance ρ_0 of node ρ, and the instance y_3 of node y.

Query Expressions We now describe the semantics of each query expression. Variable lookup and variable binding are standard. Let bindings also allow us to sequence operations with side effects, such as locks; we use a don't-care variable $\mathsf{let}\ \_ = q_1\ \mathsf{in}\ q_2$ to denote executing q_1 just for its side effects, discarding its return value, and then executing q_2.

A lock acquisition $\mathsf{lock}(q, v)$ acquires the physical locks associated with the instances of node v in the query states given by q. Like all expressions in the query language, lock acts on a set of query states produced by the evaluation of a query expression q, locking the instance of physical lock v named by each state. For example, if evaluating the query expression q yields the set of query states

$$\begin{aligned}\{&(\langle \mathit{parent}\colon 1,\ \mathit{name}\colon \text{‘a’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\})\\ &(\langle \mathit{parent}\colon 2,\ \mathit{name}\colon \text{‘b’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_2\})\}\end{aligned}$$

then the query expression $\mathsf{lock}(q, y)$ acquires locks on node instances y_2 and y_3. The lock operation must acquire locks in accordance with the lock order. While the query planner always produces the query plans with lock expressions in correct node order, the lock operator must sort node instances into the correct lexicographic order before acquiring locks. In our example, if y_2 precedes y_3 in the lock order, then the query planner must acquire the lock on y_2 before the lock on y_3. The counterpart $\mathsf{unlock}(q, v)$ unlocks the instances of node v in the set q; unlike the lock operation the unlock operation does not need to enforce sorted order on its arguments.

Recall that for each node instance u_t an edge uv in a decomposition corresponds to a container data structure that maps a set of key columns $\mathsf{cols}(uv)$ to a set of node instances of $v_{t'}$. The operation $\mathsf{scan}(q, uv)$ iterates over the contents of the container, returning the natural join of the input query states q together with the entries of the map. If the query states in q contain a superset of the key columns $\mathsf{cols}(uv)$, we can instead use the more efficient operation $\mathsf{lookup}(q, uv)$, which looks up the particular entry v_t in the container. Both the lookup and scan operations require that the input query states contain an instance of the source vertex u.

For example, suppose we wanted to iterate over all of the tuples of a directory entry relation represented using the decomposition of Figure 4.1(a) under a coarse lock placement which places all locks at the root node

($\psi(e) = \rho$ for all e). One possible query plan is:

$$\begin{aligned}
&1\text{: let } \_ = \mathsf{lock}(a, \rho) \text{ in}\\
&2\text{: let } b = \mathsf{scan}(\mathsf{scan}(a, \rho y), yy) \text{ in}\\
&3\text{: let } \_ = \mathsf{unlock}(a, \rho) \text{ in}\\
&4\text{: } b
\end{aligned} \tag{4.2}$$

Variable a is free in the query plan, and represents the input to the plan. When evaluating the query plan, a is bound to a singleton query state containing the location of the decomposition root ρ. The query plan first locks the unique instance of the root vertex ρ in set a (line 1), and then iterates over instances of the edge ρy from the root vertex in set a (line 2, $\mathsf{scan}(a, \rho y)$); the iteration yields a set of query states that contain instances of node y together with valuations of the $parent$ and $name$ fields. For each such query state, we then iterate over the singleton instances of edge yz (line 2, $\mathsf{scan}(\cdots, yz)$), yielding a valuation for the $child$ field; we store the resulting set of query states as a set b. We release the acquired locks (line 3), and return our final query states b (line 4).

To make the execution concrete, suppose we execute the query plan (4.2) on the decomposition instance of Figure 4.1(b). The query plan receives the input query state $a = \{(\langle\rangle, \{\rho \mapsto \rho_0\})\}$ as input, which specifies the location of the decomposition root but does not specify any valuations for relation columns. The lock statement acquires the lock attached to ρ_0, which is the unique instance of ρ in set a. Evaluation of the expression $\mathsf{scan}(a, \rho y)$ in line 2 yields states

$$\begin{aligned}
\big\{&(\langle parent\colon 1,\ name\colon \text{‘a’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\})\\
&(\langle parent\colon 2,\ name\colon \text{‘b’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_2\})\\
&(\langle parent\colon 2,\ name\colon \text{‘c’}\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_1\})\big\}.
\end{aligned}$$

The expression $\mathsf{scan}(\cdots, yy)$ in line 2 scans the unit edge attached each y node instance, meaning that the query reads the column values from the unit node, in this case just the $child$ field. Applying the scan expression yields the states

$$\begin{aligned}
\big\{&(\langle parent\colon 1,\ name\colon \text{‘a’},\ child\colon 2\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\})\\
&(\langle parent\colon 2,\ name\colon \text{‘b’},\ child\colon 3\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_2\})\\
&(\langle parent\colon 2,\ name\colon \text{‘c’},\ child\colon 4\rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_1\})\big\},
\end{aligned}$$

which we store as set b. Finally, we unlock the lock at ρ_0 and return the entries of b as the query result.

Query plan (4.2) was not the only possible query plan, even under the same decomposition and lock placement. Another possible query plan uses edges ρx and xy instead of the edge ρy.

$$\begin{aligned}
&1\text{: let } \_ = \mathsf{lock}(a, \rho) \text{ in}\\
&2\text{: let } b = \mathsf{scan}(\mathsf{scan}(\mathsf{scan}(a, \rho x), xy), yy) \text{ in}\\
&3\text{: let } \_ = \mathsf{unlock}(a, \rho) \text{ in}\\
&4\text{: } b
\end{aligned} \tag{4.3}$$

Now suppose we want to make the same query on the same decomposition, under the lock placement shown in Figure 4.1(a), in which a lock on every node protects the edges with their source at that node. The equivalent of query plan (4.3) under the new finer-grained lock placement is:

$$\begin{aligned}
&1\text{: let } \_ = \mathsf{lock}(a, \rho) \text{ in}\\
&2\text{: let } b = \mathsf{scan}(a, \rho x) \text{ in}\\
&3\text{: let } \_ = \mathsf{lock}(b, x) \text{ in}\\
&4\text{: let } c = \mathsf{scan}(b, xy) \text{ in}\\
&5\text{: let } \_ = \mathsf{lock}(c, y) \text{ in}\\
&6\text{: let } d = \mathsf{scan}(c, yy) \text{ in}\\
&7\text{: let } \_ = \mathsf{unlock}(c, y) \text{ in}\\
&8\text{: let } \_ = \mathsf{unlock}(b, x) \text{ in}\\
&9\text{: let } \_ = \mathsf{unlock}(a, \rho) \text{ in}\\
&10\text{: } d
\end{aligned} \tag{4.4}$$

Query Planner To pick a good implementation for each query, the compiler uses a query planner that finds the query plan with the lowest cost as measured by a heuristic cost estimation function. The concurrent query planner is based on the non-concurrent query planner of Chapter 2; like the non-concurrent query planner, the concurrent query planner enumerates valid query plans and chooses the plan with the lowest cost estimate.

The main extension for concurrency is the query planner must only permit queries that acquire and hold the right locks in the right order. Internally the query planner only considers plans with two phases, a growing phase consisting of a sequence of lock, scan, and lookup statements, and a shrinking phase containing a matching sequence of unlock statements in reverse order; such plans are trivially two-phase. To ensure that queries acquire the correct locks in the correct order, we extend the definition of query validity to require that lock statements in a query plan appear in the decomposition node lock order, and that lookup and scan operations must be preceded by a lock of the corresponding physical lock.

As in the non-concurrent case, we reuse the query planning infrastructure to compile mutation operations. Code for mutations is generated by first constructing a concurrent query plan that locates and locks all of the edges that require updating; the code generator then emits code that uses the query results to perform the required updates, just as in the non-concurrent case, sandwiched between the growing and shrinking phases of the query plan.

Query Expression Compilation Each query expression evaluates to a set of query states. Internally we compile each query expression into an iterator over query states. We compile let-bindings by evaluating the right-hand side of the binding and storing the results into a temporary set of query states; subsequent references to a bound variable compile to iterations over the stored state set.

In general the `lock` statement must sort the locks that it acquires. However, in some cases the set of locks may already be in the correct order, so it is superfluous to sort them. For example, consider the acquisition of the locks on node b in query 4.4 (line 3). Edge ρx is represented by a `TreeMap` in the decomposition, which stores its entries in sorted order; a scan over the edge will therefore yield entries in sorted order, which coincides with the correct lock order. Conversely, if edge were represented using a `HashMap` then iteration would return entries in an unpredictable order, so the code would have to sort the locks before acquiring them. The compiler uses a simple static analysis to detect lock statements where it can avoid sorting.

4.5 Experimental Evaluation

We have developed a prototype implementation of concurrent data representation synthesis, targeted at the Java virtual machine. The prototype is implemented as a Scala [Odersky et al., 2006] compiler plugin; relations and relational operations are translated into Scala ASTs, which the Scala compiler backend converts to JVM bytecode. In this section we evaluate the performance of the resulting implementation using two benchmarks—a synthetic benchmark based on directed graph operations, and a cache benchmark motivated by the real-world non-concurrent examples from Section 2.5.

4.5.1 Autotuner

A programmer may not know the best possible representation for a concurrent relation. To help find an optimal decomposition for a particular relational specification, we have implemented an autotuner which, given a concurrent benchmark, automatically discovers the best combination of decomposition structure, container data structures, and choice of lock placement.

Chapter 2 described an autotuner capable of identifying a good decomposition in the absence of concurrency. We extend the idea of autotuning to a concurrent setting.

To enumerate possible decompositions, the autotuner first chooses an adequate decomposition structure, exactly as for the non-concurrent case. Next, the autotuner chooses a well-formed lock placement; every edge of a decomposition needs a corresponding physical lock. Finally the autotuner chooses a data structure implementation for each edge. If the chosen lock placement serializes access to an edge, the autotuner picks a non-concurrent container, whereas if concurrent access to a container is permitted by the lock placement then the autotuner chooses a concurrency-safe container.

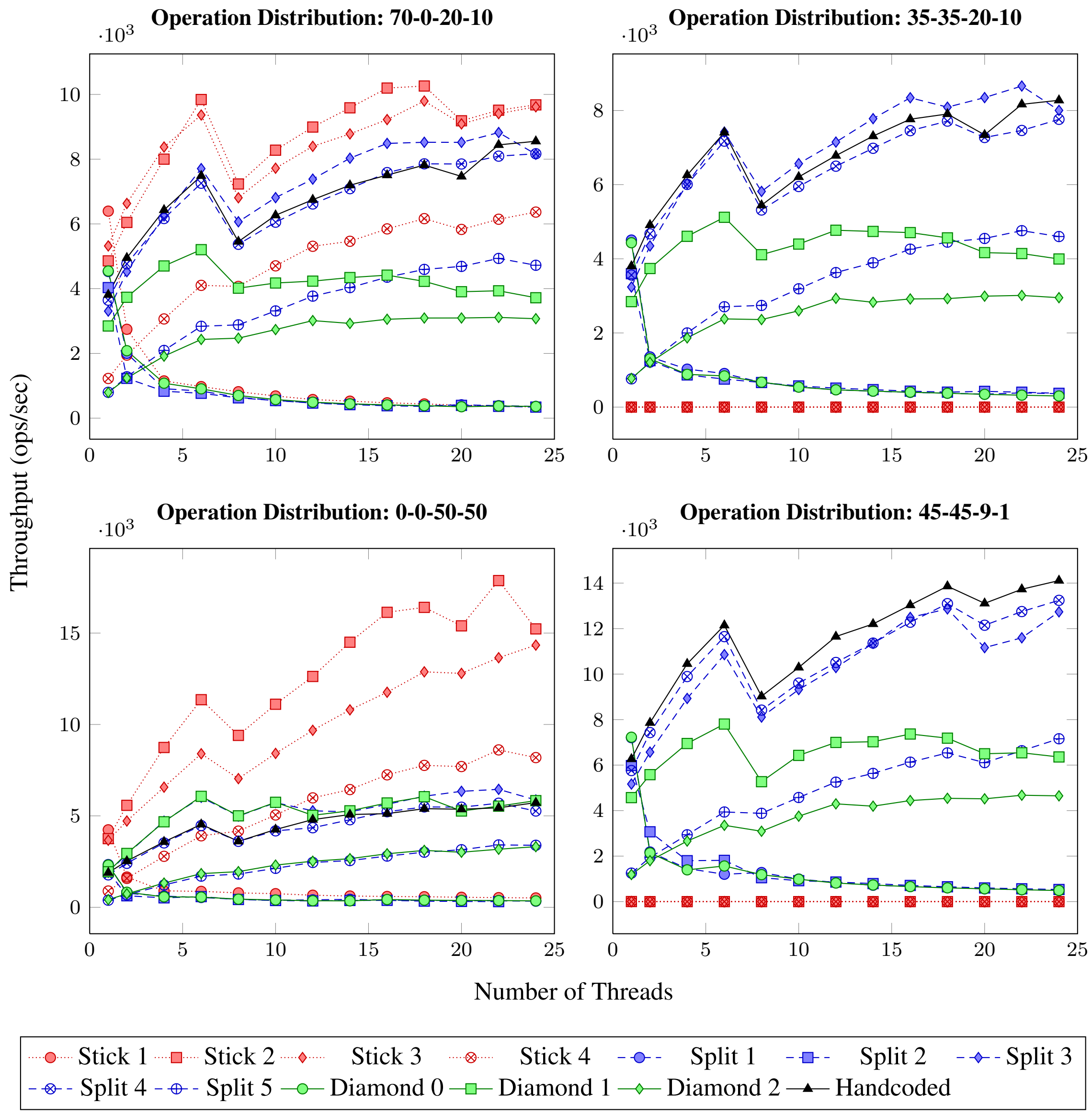


Figure 4.4: Throughput/scalability curves for a selection of decompositions. Each thread performs 5×10^5 random graph operations. Each graph is labeled x-y-z-w, denoting a distribution of $x\%$ successors, $y\%$ predecessors, $z\%$ inserts, and $w\%$ removes. "Stick" decompositions are structurally isomorphic to Figure 4.2(a) but have different choices of data structures and lock placements, similarly "split" to Figure 4.2(b), and "diamond" to Figure 4.2(c).

4.5.2 Graph Benchmark

We first evaluate the generated code using a synthetic benchmark modeled after the methodology of Herlihy et al. [2006] for comparing concurrent map implementations, extended to the more general context of a relation. We fix a particular relational specification, together with a set of relational operations. For any given choice of decomposition, the benchmark uses k identical threads that operate on a single shared relation. Starting from an initially empty relation, each thread executes 5×10^5 randomly chosen operations. We plot the total throughput of all threads in operations per second against the number of threads to obtain a throughput-scalability curve. By varying the distribution of relational operations we can evaluate the performance of the relation under different workloads.

For our benchmarks we use the directed graph relation described in Section 4.3.2, together with four relational operations, namely find successors, find predecessors, insert edge, and remove edge. The find successor operation chooses a random src value and queries the relation for the set of all $dst, weight$ pairs corresponding to that src. The find predecessor operation is similar but chooses a random dst and queries for $src, weight$ pairs. The insert edge operation chooses a random $src, dst, weight$ triple to insert into the relation; to ensure that the relation's functional dependency is not violated we use the compare-and-set functionality of the insert operation to check that no existing edge shares the same src, dst parameters. Finally the remove operation chooses a random (src, dst) tuple and removes the corresponding edge, if present.

We performed our experiments on a machine with two six-core 3.33Ghz Intel X5680 Xeon CPUs, each with 12Mb of L3 cache, and 48Gb memory in total. Hyperthreading was enabled for a total of 24 hardware thread contexts. All benchmarks were run on a OpenJDK 6b20 Java virtual machine in server mode, with a 4Gb initial and maximum heap size. We repeated each experiment 8 times within the same process, with a full garbage collection between runs. We discarded the results of the first 3 runs to allow the JIT compiler time to warm up; the reported values are the average of the last 5 runs.

Figure 4.4 presents throughput-scalability curves for a selection of decompositions. We generated 448 variants of the three decomposition structures shown in Figure 4.2 using the autotuner, varying the choice of lock placement, lock striping factor (chosen for simplicity to be either 1 or 1024), and selection of containers from the options `ConcurrentHashMap`, `ConcurrentSkipListMap`, `HashMap`, and `TreeMap`. For clarity of presentation we selected 12 representative decompositions that cover a spectrum of different performance levels across the 4 benchmarks; we compare the performance of both the automatically generated implementations and a hand-written implementation.

One obvious feature of the results is that the "stick" decompositions, which are variants of the decomposition shown in Figure 4.2(a), perform relatively well for the two workloads (70-0-20-10 and 0-0-50-50) that consist only of successor, insert, and remove operations. For the workloads that include finding predecessors (35-35-20-10 and 45-45-9-1), "split" (Figure 4.2(b)) and "diamond" (Figure 4.2(c)) perform far better. Finding successors in a stick decomposition is much more efficient than finding predecessors, which requires iterating over all edges in the graph.

Coarsely-locked data structures scale poorly; three of the decompositions shown in the graph (Stick 1, Split 1, Diamond 1) use a single coarse lock to protect the entire decomposition; each container uses a coarsely locked `HashMap` to represent the top level of edges in the decomposition, and a `TreeMap` to represent the second level of edges. Another decomposition (Split 2) uses striped locks and concurrent maps on the left side of the decomposition (ρu, uw, wx), but uses a single coarse lock to protect the other edges of the graph, leading to similarly poor performance.

Sticks 2, 3, 4 use a striped lock at the root to protect a `ConcurrentHashMap` of `HashMap` containers, a `ConcurrentHashMap` of `TreeMap` containers, and a `ConcurrentSkipListMap` of `HashMap` containers, respectively; all scale much better than the coarsely-locked data structures.

Decompositions which do not share nodes between the two sides of the decomposition outperform decompositions that do. For example, Split 3 and Diamond 1 both use `ConcurrentHashMap` containers to represent the top-level edges and `HashMap` containers to represent the second level edges, differing only in the sharing structure of the decomposition; the split decomposition performs better in most cases. Split 4 is a variant of Split 3 with `TreeMap` containers in place of the `HashMap` containers. Interestingly, there is a small but consistent effect where Split 3 is the best choice for the 35-35-20-10 workload and Split 4 is better for the 45-45-9-1 workload. Split 5 and Diamond 2 are also similar to Split 3 and Diamond 2, except with `ConcurrentSkipListMap` containers in place of `ConcurrentHashMap` containers; once again, the split decomposition outperforms the diamond decomposition.

The handcoded implementation (which was written before the automated experiments) is essentially Split 4, and produces almost identical results; the difference in performance between the two is probably due to extra boxing in the generated code that could be eliminated with improvements to the code generator. But clearly the automatically generated code is competitive with the hand-written code but requires much less programmer effort, and unless one was willing to write many different hand-coded versions, the autotuner will be able to find variants that outperform any single hand-written code for particular workload characteristics.

It is interesting to note that diamond decompositions outperformed split decompositions in the non-concurrent case (Section 2.5.1); the result here is reversed for two reasons. The split decomposition produces less lock contention, since a pair of transactions may query for successors and predecessors in parallel without interfering with one another. Much of the benefit for sharing in the non-concurrent case came from the fact that it is possible to remove an object from the middle of an intrusive doubly-linked list in constant time. Since it is impossible to write such intrusive containers in a generic fashion in the Java type system, we do not gain the advantage of more efficient removals from shared nodes.

The prominent decrease in throughput evident in Figure 4.4 when increasing from 6 to 8 threads is an artifact of the thread scheduler and the memory hierarchy of the test machine. The test machine has two six-core CPUs, each with two hardware contexts per core. The benchmark harness schedules up to the first six threads on different cores of the same CPU, sharing a common on-chip L3 cache. The harness schedules the next six threads on the second CPU; when threads are split across two CPUs they must communicate via the processor interconnect, rather than via a shared on-chip cache. Communication off-chip is substantially slower than on-chip communication, producing the “notch” in the graph.

Overall the experiments show the benefits of automatic synthesis of both data structures and synchronization: sophisticated implementations competitive with hand written code can be produced at much lower cost in programmer effort, while at the same time providing guarantees about the correctness of the implementation of the high-level concurrent relational program.

4.5.3 Concurrent Cache Benchmark

In this section we apply concurrent data representation synthesis to a concurrent cache benchmark. The cache benchmark is based on the non-concurrent examples described in Section 2.5.2, all of which use relations to implement variants on a cache.

The benchmark models a cache as a relation with three integer-valued columns key, $value$ and $refCount$. The relation satisfies a functional dependency $key \to value, refCount$. Each tuple in the relation is a mapping from a distinct integer key to an integer $value$. Every tuple also has an associated reference count $refCount$, which is the number of active users of a particular entry in the cache. Entries with non-zero reference counts are still in use and cannot be evicted from the cache; conversely entries with zero reference counts have no active users and are candidates for eviction.

The cache provides three operations, implemented by the code in Figure 4.5:

- Operation `getReference`(`key`, `makeValue`) looks up the cache entry associated a particular key. If the key is already present, then its reference count is incremented, and the existing value associated with the key is returned. If the key is absent, then function `makeValue`(`key`) is invoked to construct a new value for the key, which is then inserted into the cache with an initial reference count of 1.
- Operation `putReference`(`key`) decrements the reference count associated with `key`. If the key is not present or already has a zero reference count the operation leaves the cache unchanged.
- Operation `evict`() evicts entries with a zero reference count from the cache. Eviction is performed in Least-Recently-Used (LRU) order.

The implementation in Figure 4.5 uses the relational interface generated by the compiler, which is shown in Figure 4.6. While each of the relational operations generated by the compiler is individually atomic, each cache operation performs multiple relational operations, so the code must guard against concurrent modifications from other threads between relational operations. Both the `getReference` and `putReference` operations are structured as loops containing a pair of relational operations: a query to find the existing data associated with a key, and an insert or an update to update that data. The insert or update double-checks the data returned by the query operator; if an inconsistency is detected between the query and the subsequent update the cache operation is restarted. The `evict` operation uses a query operation to find all keys with a zero reference count. The subsequent removal operation double-checks that the reference count of each key is still zero; if not, then the key is not evicted from the cache.

We evaluate the effect of different concurrent decompositions on the performance of the cache using a methodology similar to that of Section 4.5.2. For any given choice of decomposition, the benchmark uses k

```
class ConcurrentCache {
 val r : CacheRelation

 def getReference(key: Int, makeValue: Int => Int): Int = {
  lazy val newValue = makeValue(key)

  while (true) {
    r.query_Key_RefCountValue(key) match {
     case None =>
       if (!r.insert_Key(key, 1, newValue))
         return newValue // Insertion succeeded
     case Some((refCount, existingValue)) =>
       if (r.update_KeyRefCount_RefCount(key, refCount, refCount + 1))
         return existingValue // Update succeeded
    }
    // We raced with another thread on the same key, so we must retry.
  }
  value
 }

 def putReference(key: Int) {
  while (true) {
    r.query_Key_RefCountValue(key) match {
     case None => return // Key doesn't exist in the cache
     case Some((refCount, value)) =>
      if (refCount == 0) return // Key already had zero reference count
      if (r.update_KeyRefCount_RefCount(key, refCount, refCount - 1))
       return // Successfully decremented the reference count
     }
     // We raced with another thread, so we must retry.
   }
 }

  def evict() {
    var victimKeys = List[Int]()
    // Find all keys with zero reference count.
    r.query_RefCount_KeyValue(0, { (key, value) =>
      victimKeys = key :: victimKeys
    })
    for (victimKey <- victimKeys) {
      // Remove a key only if its reference count is still zero.
      r.remove_KeyRefCount(victimKey, 0)
    }
  }
}
```

Figure 4.5: Scala code implementing a concurrent cache using the relational interface generated by the compiler (Figure 4.6).

```
trait CacheRelation {
  // Insert a new tuple into the relation, if no tuple with the same
  // key already exists. Returns true if the tuple already exists.
  def insert_Key(key: Int, refCount: Int, value: Int): Boolean

  // Return the (refCount, value) tuple associated with a key, if any.
  def query_Key_RefCountValue(key: Int): Option[(Int, Int)]

  // Return the (key, value) fields of all tuples with a given refCount,
  // in Least-Recently-Used order.
  def query_RefCount_KeyValue(refCount: Int,
    yieldFn: (Int, Int) => Unit): Unit

  // Update the tuple matching key and refCount, setting the reference
  // count to updRefCount. Returns true if the update was succesful.
  def update_KeyRefCount_RefCount(key: Int, refCount: Int,
    updRefCount: Int): Boolean

  // Remove the tuple with a given key and refCount, if any.
  def remove_KeyRefCount(key: Int, refCount: Int): Unit
}
```

Figure 4.6: The interface to the concurrent cache relation generated by the compiler, edited for clarity and with comments added.

identical threads that operate on a single shared relation. Starting from an initially empty relation, each thread executes 5×10^4 randomly chosen cache operations. We plot the total throughput of all threads in operations per second against the number of threads to obtain a throughput-scalability curve.

Figure 4.7(a) shows the decomposition used for the concurrent cache experiment. Each instance of node w corresponds to an entry in the cache. There are two paths to any instance of w. Edge ρw maps a key to the corresponding instance of node w. Alternatively, edge ρu maps a $refCount$ to an instance of node u corresponding to the subrelation containing entries with that reference count. For any given reference count edge uw maps each key to the corresponding instance of node w. The edge uw is annotated [LRU], which denotes that a scan over instances edge uw must return its results in Least-Recently-Used order.

Figure 4.8(a) presents throughput-scalability curves for the 44 variants of the decomposition in Figure 4.7(a) generated by the autotuner. The autotuner varies the choice of lock placement, lock striping factor (chosen for simplicity to be either 1 or 1024), and selection of containers from the options `ConcurrentHashMap`, `ConcurrentSkipListMap`, `HashMap`, `TreeMap`, and a Scala `DoubleLinkedList`.

We describe two of the more interesting decompositions in detail. Both decompositions represent edge uw using a `DoubleLinkedList` since, of the available containers, it is the only container capable of maintaining an LRU order. The decomposition labeled "—🟩—" uses a `HashMap` for edges ρw and ρu, and protects everything with a single coarse-grained lock at the root. This decomposition performs best in the single-threaded case;

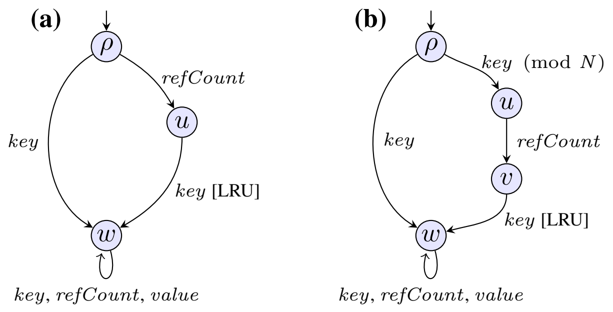


Figure 4.7: Concurrent decompositions for the cache benchmark. Decomposition (a) uses a single list of keys in Least-Recently-Used order for each reference count value, whereas decomposition (b) partitions the key space into N partitions and uses a separate LRU list for each partition/reference count pair.

the addition of more threads only reduces performance due to lock contention. Decomposition “–●–” uses a `ConcurrentHashMap` for edges ρw and ρu, together with a lock striped by key at the root to protect instances of edge ρw, and another striped lock striped by $refCount$ at the root to protect instances of edges ρu. Edge uw is protected by a lock at u, and each instance of node w has its own lock to protect the associated singleton fields. This decomposition has relatively good performance under high contention, but mediocre performance in the single-threaded case.

No variant of the decomposition shown in Figure 4.7(a) scales well, largely because every operation contends on the locks that protect the LRU list of keys associated with each reference count. Maintaining a strict Least-Recently-Used order requires that threads agree on the observed order of cache operations, limiting parallelism.

To allow more concurrency between transactions, we altered the benchmark to relax the LRU ordering requirement. Figure 4.7(b) shows a revised decomposition, which approximates a per-reference count LRU list by splitting the key space into a N stripes and using a separate LRU list for each (here we choose $N = 1024$). Although the revised decomposition no longer maintains a precise LRU order, splitting the list allows operations on the same reference count value but on different key stripes to proceed in parallel without contending on the same lock. Figure 4.8(b) shows the results for the 162 variants of the decomposition in Figure 4.7(b) generated by the autotuner; we describe just three in detail.

Once again, all decompositions use a `DoubleLinkedList` for edge vw due to the need to maintain an LRU order. Decomposition “-●-” uses `HashMaps` to represent all of the other edges, and protects the entire decomposition with a single coarse-grained lock at the root. This decomposition performs relatively poorly even in the single-threaded case, probably due to the extra overhead of maintaining additional data structures as compared to decomposition “–■–”.

Decomposition “-▲-” uses `ConcurrentSkipListMaps` to implement edges ρw and ρu, and a `TreeMap` to implement edge uv. A striped lock at the root indexed by key protects edge ρw, whereas another striped lock at the root indexed by $key \pmod N$ protects edges ρu, uv, and vw. Node w has its own lock to protect

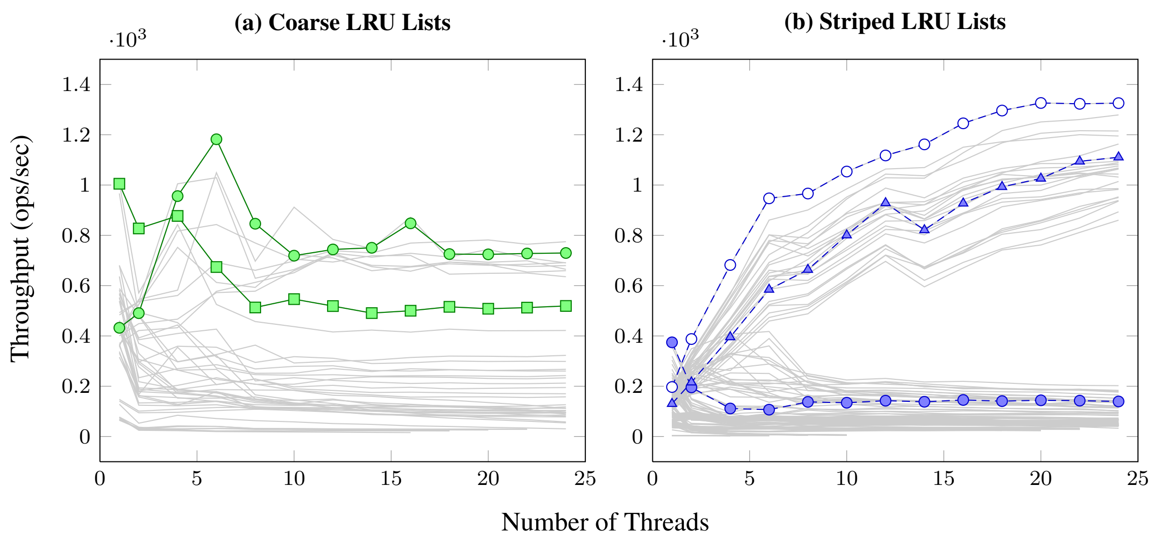


Figure 4.8: Throughput/scalability curves for a selection of concurrent cache decompositions. Each thread performs 5×10^4 random graph operations, with a distribution of 70% get reference operations, 28% put reference operations, and 2% eviction operations. The decompositions in the left graph use a single Least-Recently-Used list per reference count value, whereas the decompositions in the right graph approximate LRU order by using separate lists for different parts of the key space within each reference count value.

the associated singleton fields. Decomposition "-○-" is similar, but uses `ConcurrentHashMap` in place of `ConcurrentSkipListMap`, and `HashMap` in place of `TreeMap`. Both decompositions have good scalability characteristics; in particular both perform better under contention than any of the decompositions with a single LRU list.

There are several ways to optimize the generated code which we hope to explore in future work. Firstly, all of the cache operations add or remove nodes to LRU lists. However, since Scala lacks generic intrusive data structures, removing a node from the middle of an LRU list takes time linear in the size of the list. If we were to extend the compiler to emit some form of intrusive linked list, we could remove linear time algorithms from the `getReference` and `putReference` implementations. Secondly, the decomposition maintains a LRU list for each reference count; it is however only necessary to maintain a LRU list just for those nodes with reference count zero. If we were to extend the compiler to support such partial data structures, then we could avoid many unnecessary list updates for nodes with non-zero reference counts.

Finally, each of the cache operations performs multiple relational operations in succession. Since the compiler compiles each relational operation in isolation, invoking sequences of relational operations often leads to redundant work between operations. In future work we hope to extend our compiler to handle a larger transaction language, rather than just a compiling a single operation at a time, allowing this redundant work to be optimized away.

4.6 Discussion and Related Work

Simultaneously with and independently from our work, Manevich et al. [2012] have investigated synthesizing concurrent graph representations from a relational description. Their approach is similar in many ways, but differs particularly in how abstract and concrete synchronization are used to ensure safety and serializability of transactions. The logical locks of our concurrent decomposition language correspond to a class of abstract locks (in their terminology). Our lock placements, which describe how to map logical locks onto a particular concrete implementation, do not have a counterpart in their proposal. We implement concrete synchronization both by leveraging the safety and consistency properties of concurrent containers, and by placing restrictions on the choice of lock placement that only permit concurrent access to a container if such access is safe and linearizable. In their proposal abstract synchronization is implemented as a separate table of abstract locks, it is the responsibility of individual containers to ensure concrete safety and linearizability.

The closest previous literature to our work in spirit is Paraglider [Vechev and Yahav, 2008], which provides semi-automatic assistance in synthesizing low-level concurrent algorithms and data structures. Paraglider focuses on the correct implementation of a single concurrent data structure, while our work is about assembling multiple concurrent and non-concurrent data structures into more complex abstractions. Thus, Paraglider is complementary to our approach, and we could extend our menu of concurrent building blocks with Paraglider-generated components.

Our system can be viewed as implementing a pessimistic software transactional memory [Shavit and Touitou, 1997]. Future extensions of our work could synthesize optimistic concurrency control primitives in addition to pessimistic locks. Unlike traditional software transactional memory systems, which perform on-line dynamic analysis to determine the read and write sets of transactions, our system performs much of the same analysis statically, resulting in run-time code with considerably lower overhead. Furthermore, our approach is able to automatically change the data structures and granularity of locking used to improve overall performance. It is also worth noting that speculative locking was first introduced in the context of advanced software transactional memory systems [Bronson et al., 2010a].

Chapter 5

Conclusions

This dissertation proposes a new approach for specifying combinations of data structures with complex sharing in a manner that is both declarative and results in provably correct code.

We describe a scheme for synthesizing efficient low-level data representations from abstract relational descriptions of data. In our approach programmers describe and manipulate data at a high level as relations, and it is the task of the compiler to generate implementations of the various relational operations specialized to a particular representation.

We propose a novel decomposition language, which specifies how relations should be mapped to low-level physical implementations, assembled from a library of primitive container data structures. The decomposition language provides a new way to specify high-level heap invariants that are difficult or impossible to express using standard data abstraction or heap-analysis techniques. We describe adequacy conditions that ensure a decomposition faithfully represents a relation.

Decompositions can be combined with locks to describe a wide range of concurrent representations of data. We introduce the idea of a lock placement: such diverse concepts as lock granularity, speculative locks, lock splitting and merging, and dynamically changing lock assignments can all be understood as examples of a lock placement that maps each heap field to a lock that guards it. We have also identified the key concept of a stable set of mutually supporting locks and heap facts, where the set of locks protect the heap facts and the set of heap facts preserve the lock placement. We have used these two concepts to develop a series of proof systems for showing that transactions are well-locked and therefore serializable, applying these technique to flat heaps, tree-structured heaps, and finally decompositions.

We describe how to synthesize efficient low-level implementations of relational queries and updates, specialized to a particular decomposition. Key to our approach is a query planner that chooses an efficient execution plan for each query or mutation. We show that queries and updates are sound, that is, the implementation generated by our compiler for each query or mutation faithfully implements its relational specification. We extend query planning to concurrent relations; the concurrent query planner ensures that queries take the correct locks in the correct order to guarantee serializability and deadlock-freedom.

A programmer may not know the best decomposition for a particular relation. We describe an autotuner, which given a relational specification and a performance metric, such as a benchmark, finds the decomposition with the best combination of data structures for that query load. We extend the autotuner to support concurrency; the concurrent autotuner finds the decomposition with the combination of data structures and lock placement that produces the best performance. Exploring different choices of data structures and locking strategies is extremely difficult in hand-written code, since writing concurrent data structure code is a difficult and error-prone process.

Finally we describe our experience with two different implementations of data representation synthesis. We describe a non-concurrent compiler which takes an input a relation and its decomposition and generates C++ code implementing the relation, which is easily incorporated into existing systems. We also describe a separate concurrent implementation, which converts a decomposition incorporating a lock placement into a Scala implementation. We demonstrate that different choices of decomposition have very different performance characteristics. We incorporate synthesis into a variety of real systems, in each case leading to code that is simpler, correct by construction, and comparable in performance.

Bibliography

Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: `10.1145/1142473.1142548`.

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 967–980, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: `10.1145/1376616.1376712`.

Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Supercomputing*, page 58. IEEE Computer Society, November 2000. doi: `10.1109/SC.2000.10033`.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. thesis, DIKU, University of Copenhagen, Denmark, 1994.

Hagit Attiya, G. Ramalingam, and Noam Rinetzky. Sequential verification of serializability. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–42, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: `10.1145/1706299.1706305`.

Don Batory and Jeff Thomas. P2: A lightweight DBMS generator. *Journal of Intelligent Information Systems*, 9:107–123, 1997. ISSN 0925-9902. doi: `10.1023/A:1008617930959`.

Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5):441–452, May 2000. ISSN 0098-5589. doi: `10.1109/32.846301`.

J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar. DataBlitz storage manager: Main-memory database performance for critical applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 519–520, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8. doi: `10.1145/304182.304239`.

Catriel Beeri, Ronald Fagin, and John H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–61, New York, NY, USA, 1977. ACM. doi: `10.1145/509404.509414`.

Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-29735-2. doi: `10.1007/11575467_5`.

Josh Berdine, Cristiano Calgano, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer Berlin / Heidelberg, 2007. doi: `10.1007/978-3-540-73368-3_22`.

Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 262–286. Springer Berlin / Heidelberg, 2005. doi: `10.1007/11531142_12`.

Cui Bin. *Indexing for Efficient Main Memory Processing*. Ph.D. thesis, National University of Singapore, 2003.

Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 220–231. ACM, 2004. doi: `10.1145/964001.964020`.

Peter Boncz, Martin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 225–237. VLDB, 2005. URL `http://www.cidrdb.org/cidr2005/papers/P19.pdf`.

Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8:101–119, 1999. ISSN 1066-8888. doi: `10.1007/s007780050076`.

Boost. Boost C++ libraries, 2010. URL `http://www.boost.org/`.

Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation, and aliasing. In *Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Compute Environments for Memory Management (SPACE)*, 2004. URL `http://www.diku.dk/topps/space2004/space_final/bornat-calcagno-ohearn.pdf`.

Nathan G. Bronson. *Composable Operations on High-Performance Concurrent Collections*. Ph.D. thesis, Stanford University, 2011.

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: High-performance concurrent sets and maps for STM. In *Proceeding of the 29th ACM SIGACT-SIGOPS*

Symposium on Principles of Distributed Computing (PODC), pages 6–15, New York, NY, USA, 2010a. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835703.

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, New York, NY, USA, 2010b. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693488.

Jiazhen Cai and Robert A. Paige. “Look ma, no hashing, and no arrays neither”. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 143–154, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99605.

Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2009. doi: 10.1145/1480881.1480917.

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310. ACM, 1990. ISBN 0-89791-364-7. doi: 10.1145/93542.93585.

Shaunak Chatterjee, Shuvendu Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_4.

Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers. ISBN 1-55860-470-7.

Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–315, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375619.

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596565.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6): 377–387, 1970. ISSN 0001-0782. doi: 10.1145/362384.362685.

Donald Cohen and Neil Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3): 53–60, May 1993. doi: 10.1109/52.210604.

Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In Laurie Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78790-7. doi: `10.1007/978-3-540-78791-4_19`.

Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: `10.1145/349299.349309`.

Jyotirmoy Deshmukh, G. Ramalingam, Venkatesh-Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In *Programming Languages and Systems*, volume 6012 of *LNCS*, pages 226–245. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-11956-9. doi: `10.1007/978-3-642-11957-6_13`.

Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the International Conference of Computer Languages*, pages 2–13, Apr 1992. doi: `10.1109/ICCL.1992.185463`.

Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):27–49, January 1979. ISSN 0164-0925. doi: `10.1145/357062.357064`.

David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 1–8, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. doi: `10.1145/602259.602261`.

Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998. ISSN 0743-7315. doi: `10.1006/jpdc.1998.1441`.

Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-33056-1. doi: `10.1007/11691372_19`.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: `10.1145/178243.178264`.

Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 291–296, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: `10.1145/1190216.1190260`.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19:624–633, November 1976. ISSN 0001-0782. doi: `10.1145/360363.360369`.

Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *Proceedings of the International Workshop on Alias Confinement and Ownership*, July 2003.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 253–263, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: `10.1145/349299.349332`.

Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, December 1992. ISSN 1041-4347. doi: `10.1109/69.180602`.

Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic fine-grain locking using shape properties. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37756-6. doi: `10.1007/11823230_16`.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76636-0. doi: `10.1007/978-3-540-76637-7_3`.

Sumit Gulwani and Ashish Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In Werner Damm and Holger Hermanns, editors, *Proceedings of Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 379–392. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73367-6. doi: `10.1007/978-3-540-73368-3_42`.

Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 256–265, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: `10.1145/1250734.1250764`.

Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 310–323, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040331.

Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 353–364, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: 10.1109/PACT.2007.23.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data structure fusion. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 6461 of *Lecture Notes in Computer Science*, pages 204–221. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-17164-2_15.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993504.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Reasoning about lock placements. In *Programming Languages and Systems*, volume 7211 of *LNCS*, pages 336–356. Springer Berlin / Heidelberg, 2012a. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_17.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2012b. ACM.

David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781150.

Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 249–260, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143138.

Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Proceedings of the Conference On Principles of Distributed Systems (OPODIS)*, 2006.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, 2006.

C.B. Jones. *Development methods for computer programs including a notion of interference*. Ph.D. thesis, Oxford University, 1981.

Uri Juhasz, Noam Rinetzk, Arnd Poetzsch-Heffter, Mooly Sagiv, and Eran Yahav. Modular verification with shared abstractions. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 196–205, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158628.

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 318–327. Springer Berlin / Heidelberg, 1997. doi: 10.1007/BFb0002751.

Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level C with overlapping structures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5044 of *Lecture Notes in Computer Science*, pages 214–230. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-11319-2_17.

Victor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.125.

Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–32, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503276.

Patrick Lam. *The Hob system for verifying software design properties*. Ph.D. thesis, MIT, Cambridge, MA, USA, 2007.

Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 430–447. Springer Berlin / Heidelberg, 2005. doi: 10.1007/978-3-540-30579-8_28.

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 235–248, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143137.

William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 56–67, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: `10.1145/155090.155096`.

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 140–140. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-25435-5. doi: `10.1007/978-3-540-31987-0_10`.

Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *Proceedings of Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 592–608. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22109-5. doi: `10.1007/978-3-642-22110-1_48`.

Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *Proceedings of the 12th International Conference on Very Large Databases (VLDB)*, pages 294–303. Morgan Kaufmann, 1986. ISBN 0-934613-18-4.

Tobin J. Lehman, Eugene J. Shekita, and Luis-Felipe Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, dec 1992. ISSN 1041-4347. doi: `10.1109/69.180606`.

Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or B-tree: Main memory database index structure revisited. In *Proceedings of the 11th Australasian Database Conference (ADC)*, pages 65–73, 2000. doi: `10.1109/ADC.2000.819815`.

Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring invariants in separation logic for imperative list-processing programs. In *3rd SPACE Workshop*, 2006.

Roman Manevich, Rashid Kaleem, and Keshav Pingali. Synthesizing concurrent relational data structures. Technical Report TR-2012-17, The University of Texas at Austin, 2012.

Mark Marron, Darko Stefanovic, Manuel Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 31–36, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3. doi: `10.1145/1251535.1251541`.

Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 346–358, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: `10.1145/1111037.1111068`.

Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 71–99. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-15768-4. doi: `10.1007/978-3-642-15769-1_6`.

Scott McPeak and George Necula. Data structure specifications via local equality axioms. In *Proceedings of Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-27231-1. doi: `10.1007/11513988_47`.

Eric Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *SIGMOD*, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: `10.1145/1142473.1142552`.

Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: `10.1145/378795.378851`.

Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338. ACM, 2007. ISBN 1-59593-575-4. doi: `10.1145/1190216.1190265`.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An Overview of the Scala Programming Language, second edition. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2006.

Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3): 271–307, 2007.

John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, January 2010. ISSN 0163-5980. doi: `10.1145/1713254.1713276`.

Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code — a case study. *Journal of Symbolic Computation*, 4(2):207–232, 1987. ISSN 0747-7171. doi: `10.1016/S0747-7171(87)80066-4`.

Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 78–89, 1999.

Jun Rao and Kenneth A. Ross. Making B+- trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: `10.1145/342009.335449`.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002. doi: `10.1109/LICS.2002.1029817`. Invited paper.

Tom Rothamel and Yanhong A. Liu. Efficient implementation of tuple pattern based retrieval. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 81–90, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: `10.1145/1244381.1244394`.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. ISSN 0164-0925. doi: `10.1145/514188.514190`.

Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–210, New York, NY, USA, 1979. ACM. doi: `10.1145/567752.567771`.

Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: `10.1145/1542476.1542522`.

Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2011.

Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. ISSN 0178-2770. doi: `10.1007/s004460050028`.

Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Conference on Domain-Specific Languages*, pages 257–271, October 1997.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: `10.1145/237721.237727`.

Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One size fits all? — Part 2: Benchmarking results. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 173–184. VLDB, 2007a. URL `http://www.cidrdb.org/cidr2007/papers/cidr07p20.pdf`.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (It's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Databases (VLDB)*, pages 1150–1160, 2007b. ISBN 978-1-59593-649-3.

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*, pages 553–564, 2005. ISBN 1-59593-154-6.

TimesTen. In-memory data management for consumer transactions: The TimesTen approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 528–529, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8. doi: `10.1145/304182.304244`.

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 188–201, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: `10.1145/174675.177855`.

Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 — Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74406-1. doi: `10.1007/978-3-540-74407-8_18`.

Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78. Springer Berlin / Heidelberg, 2007. doi: `10.1007/978-3-540-73589-2_4`.

Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 125–135, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: `10.1145/1375581.1375598`.

William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 83–94, New York, NY, USA, 1980. ACM. ISBN 0-89791-011-7. doi: `10.1145/567446.567455`.

John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: `10.1145/996841.996859`.

John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 610–629. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-11956-9. doi: `10.1007/978-3-642-11957-6_32`.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: 10.1145/207110.207111.

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-70543-7. doi: 10.1007/978-3-540-70545-1_36.

Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. *Journal of Logic and Algebraic Programming*, 73(1-2):111–142, 2007. ISSN 1567-8326. doi: 10.1016/j.jlap.2006.12.001.

Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 349–361, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375624.

Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–351, 2009.

Yuan Zhang, Vugranam Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-89739-2. doi: 10.1007/978-3-540-89740-8_10.