An Introduction to Data Representation Synthesis

Peter Hawkins
Stanford University
hawkinsp@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Kathleen Fisher
Tufts University
kfisher@eecs.tufts.edu

Martin Rinard
MIT
rinard@csail.mit.edu

Mooly Sagiv
Tel-Aviv University
msagiv@post.tau.ac.il

ABSTRACT

We consider the problem of 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 permits the user to specify different concrete representations for relations, and show that operations on concrete representations soundly implement their relational specification. We also describe an auto-tuner that automatically identifies the best decomposition for a particular workload. 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.

1. INTRODUCTION

One of the first things a programmer must do when implementing a system is commit to particular data structure choices. For example, consider a simple operating system process scheduler. Each process has an ID *pid*, a *state* (running or sleeping), and a variety of statistics such as the *cpu* time consumed. Since we need to find and update processes by ID, we store processes in a hash table indexed by *pid*; as we also need to enumerate processes in each state, we simultaneously maintain a linked list of running processes and a separate list of sleeping processes.

Whatever our choice of data structures, it has a pervasive influence on the subsequent code. Moreover, as requirements evolve it is difficult and tedious to change the data structures to match. For example, suppose we add virtualization support by allowing processes with the same *pid* number to exist in different namespaces *ns*, together with the ability to enumerate processes in a namespace. Extending the existing data structures to support the new requirement may require many changes distributed throughout the code.

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 the scheduler, we require that each process appears in both the hash-table indexed by process ID and exactly one of the running or sleeping lists. Such invariants must be enforced by every piece of code that manipulates the scheduler's data structures. It is easy to forget a case, say by failing to add a process to the appropriate list when it changes state or by failing to delete a hash table entry when a process terminates. Invariants of this nature require deep knowledge about the heap's structure, and are difficult to enforce through existing static analysis or verification techniques.

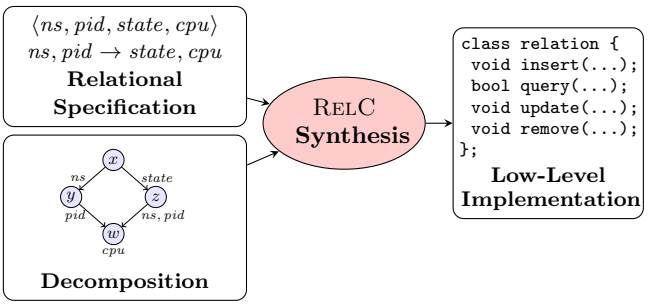

Figure 1: Data Representation Synthesis.

We propose to use *data representation synthesis*, depicted in Figure 1. 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* that 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.

Synthesis allows programmers to describe and manipulate data at a high level as relations, while enabling control of how relations are represented physically in memory. By abstracting data from its representation, programmers no longer prematurely commit to a particular data representation. 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 obeys the constraints listed in the relational specification, invariants on the synthesized data structures are automatically maintained.

Due to space constraints we omit many details from this article. See the full paper [11] for a more comprehensive

The original version of this paper was published in the Proceedings of Programming Language Design and Implementation (PLDI), 2011, ACM.

treatment.

2. 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 3.

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 v. We write $t(c)$ for the value of column c in tuple t. A *relation* r is a set of tuples $\{t_1, t_2, \dots\}$ over identical columns C. A relation r has a *functional dependency* $C_1 \to C_2$ if any pair of tuples in r that are equal on columns C_1 are also equal on columns C_2.

A *relational specification* is a set of column names C and functional dependencies Δ. For the scheduler example from Section 1, processes may be modeled 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. The other columns have integer values. For example, the scheduler might represent three processes as the relation:

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

Not every relation represents a valid set of processes; all meaningful sets of processes satisfy a functional dependency $ns, pid \to state, cpu$, which allows at most one *state* or *cpu* value for any given process.

Relational Operations.

We provide five operations for creating and manipulating relations. Operation 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:

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

The operation

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

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

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

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

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

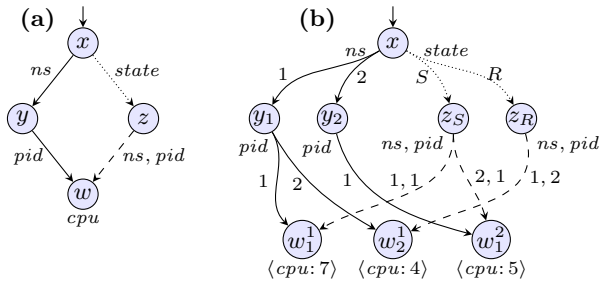

Figure 2: Data representation for a process scheduler: (a) a decomposition, (b) an instance of that decomposition. Solid edges represent hash tables, dotted edges represent vectors, and dashed edges represent doubly-linked lists.

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.

3. DECOMPOSITIONS

Decompositions describe how to represent relations as a combination of primitive container data structures. Not every decomposition can correctly implement every relation. Our goal in this section is to define *adequacy conditions* which are sufficient conditions for a decomposition to faithfully represent a relation.

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

Figure 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.

We now describe the three decomposition primitives.

- A *unit* decomposition, depicted as a node labeled with a set of columns C, represents a single tuple t with columns C. A unit node cannot have any outgoing edges. For example, in Figure 2(a) node w is a unit

decomposition containing a single *cpu* value.

- A *map* decomposition, depicted as an edge labeled with a set of columns C, represents a relation as a mapping $\{t \mapsto v_{t'}, \dots\}$ from a set of columns C, called *key columns*, to a set of *residual relations* $r_{t'}$, one for each valuation t of the key columns. Each residual relation $r_{t'}$ is in turn represented by decomposition v. For example, in Figure 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. Any container that implements a key-value associative map interface can be used to implement the map; the example uses unordered doubly-linked lists of key-value pairs, hash tables, and arrays mapping keys to values. The set of containers is extensible; any container implementing a common interface may be used.

- Multiple edges that exit the same node denote a *join*, which represents a relation as the natural join of two sub-relations r_1 and r_2. Each sub-relation has its own sub-decomposition, allowing a join to represent the same data but with different cost models. In diagrams, join decompositions exist wherever multiple map edges exit the same node. For example, in Figure 2(a) node x has two outgoing edges and hence is the join of two map decompositions.

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(b) shows an instance of the decomposition representing the relation r_s defined in Equation (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. For example, node $z_{\langle state\colon S\rangle}$ 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.

Each decomposition node has an associated "type," consisting of a pair of column sets $B \triangleright C$; every instance of node v in a decomposition instance has a distinct valuation of columns B, and each such instance represents a relation with columns C. In the decomposition of Figure 2(a), the root node x has type $\emptyset \triangleright \{ns, pid, cpu, state\}$. Since there is only one valuation for the empty set of columns $\emptyset$ there is exactly one instance of variable x in any instance of the decomposition; further the subgraph of a decomposition instance rooted at x represents a relation with all columns. Similarly, node y has type $\{ns\} \triangleright \{pid, cpu\}$, implying that there is a distinct instance of node y for each value of the ns field in the relation, and that the subgraph rooted at each instance of y represents a relation with columns pid, cpu. Node z has type $\{state\} \triangleright \{ns, pid, cpu\}$. Finally node w has type $\{ns, pid, state\} \triangleright \{cpu\}$.

The structure of a decomposition instance parallels the structure of the decomposition; for each node $v\colon B \triangleright C$ in the decomposition the instance contains a corresponding set of node instances $\{v_t, v_{t'}, \dots\}$, each for different valuations of columns B. For example, in Figure 2, decomposition node z has two different instances $z_{\langle state\colon S\rangle}$ and $z_{\langle state\colon R\rangle}$, one for each *state* value in the relation.

3.2 Adequacy of Decompositions

We define an *abstraction function* α, defined formally in the full paper [11], which maps decomposition instances to the relation they represent. The abstraction function is defined recursively on the structure of the decomposition instance. Informally, a unit node represents its associated tuple t, a map decomposition represents the union of t joined with $\alpha(v_{t'})$ for each edge $t \mapsto v_{t'}$ that comprises the map, and a join decomposition represents the natural join of all of its subdecompositions.

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 $\hat{d}$ in Figure 2(a) cannot represent the relation

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

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 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 define *adequacy conditions*, that characterize when a decomposition $\hat{d}$ is a suitable representation for a relation with columns C satisfying functional dependencies (FDs) Δ. The adequacy conditions are analogous to a type system; if a decomposition $\hat{d}$ is adequate, then it can represent every possible relation with columns C satisfying FDs Δ:

LEMMA 1. *If decomposition $\hat{d}$ is adequate for relations with columns C and FDs Δ, then for each relation r with columns C that satisfies Δ there is some instance d such that $\alpha(d) = r$.*

4. QUERIES AND UPDATES

In Section 3 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 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.

4.1 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, similar to a database: *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.

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 plan answers the query correctly.

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. 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. 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.

A *query plan* is a tree of query plan operators. 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 $\mathsf{qscan}(q)$ 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 $\mathsf{qlookup}(q)$ 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. In general, we expect qlookup to have better time complexity than qscan.

Join The $\mathsf{qjoin}(q_1, q_2, lr)$ 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. The $\mathsf{qlr}(q, lr)$ operator is a special case of the join operator, which 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.

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. The two plans described above that implement the query are

$$q_1 = \mathsf{qlr}\big(\mathsf{qlookup}(\mathsf{qscan}(\mathsf{qunit})), \mathsf{right}\big)$$

$$q_2 = \mathsf{qjoin}\big(\mathsf{qlookup}(\mathsf{qscan}(\mathsf{qunit})),\ \mathsf{qlookup}(\mathsf{qlookup}(\mathsf{qunit})), \mathsf{left}\big).$$

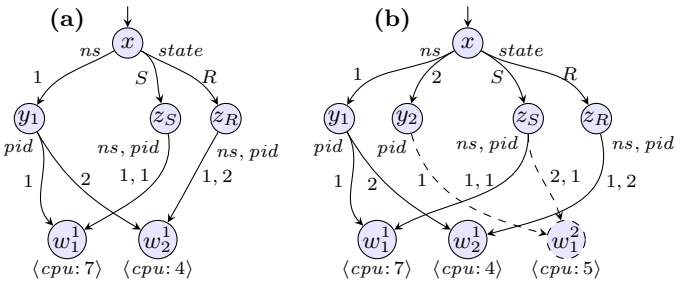

Figure 3: Example of insertion and removal. Inserting the tuple $t = \langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle$ into instance (a) produces instance (b); conversely removing tuple t from (b) produces (a). Differences between the instances are shown using dashed lines.

An important property of the query operators is that they all require only constant space; there is no need to allocate intermediate data structures to execute a query. 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.

Not every query plan is a correct strategy for evaluating a query. We say a query plan q is *valid* if q correctly answers queries over a decomposition $\hat{d}$ given a set of input columns A and yielding output columns B. Query validity is defined formally in the full paper [11].

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 simple 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.

4.2 Mutations

Next we turn our attention to operations dempty, which creates an empty instance of a decomposition $\hat{d}$, and dinsert, 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.

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 a node instance v_s in the decomposition instance, where s is tuple t restricted to columns B. 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, inserting the tuple $\langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle$ into

the decomposition instance shown in Figure 3(a) yields the state shown in Figure 3(b).

We next consider the dremove and dupdate operations. Operation dremove removes tuples matching tuple s from an instance d of decomposition $\hat{d}$.

To remove tuples matching a tuple t, we compute a cut of the decomposition between those nodes that can only be part of the representation of tuples matching t, and those nodes that may form part of the representation of tuples that do not match t. We then break any edge instances crossing the cut. To find the edge instances to break we can reuse the query planner. Once all such references are removed, the instances of nodes in Y are unreachable from the root of the instance and can be deallocated. We can also clean up any map nodes in X that are now devoid of children.

Operation dupdate 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.

4.3 Soundness of Relational Operations

Finally we show that the operations on decompositions faithfully implement their relational specifications.

THEOREM 2. *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 each intermediate relation satisfies FDs Δ. Then the corresponding sequence of* dinsert, dupdate, *and* dremove *operators given* dempty $\hat{d}$ *as input produce an instance d such that $\alpha(d) = r$.*

5. AUTO-TUNER

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 *auto-tuner* that, given a program written to the relational interface, attempts to infer the best possible decomposition for that program.

The auto-tuner 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 auto-tuner 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.

6. 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 6.1) 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

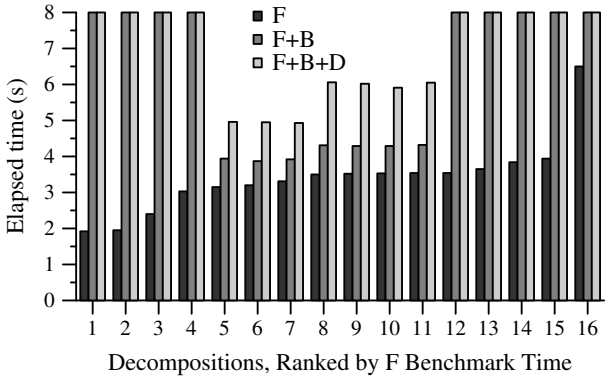

Figure 4: Elapsed times for directed graph benchmarks for decompositions up to size 4 with identical input. For each decomposition we show the times to traverse the graph forwards (F), to traverse both forwards and backwards (F+B), and to traverse forwards, backwards and delete each edge (F+B+D). We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

systems. We apply synthesis to three existing systems (Section 6.2), 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, namely doubly-linked lists (`std::list`, `boost::intrusive::list`), binary trees (`std::map`, `boost::intrusive::set` hash-tables (`boost::unordered_map`), and vectors (`std::vector`). The library includes both non-intrusive containers, in which elements are represented out-of-line as pointers, and intrusive containers, in which the pointer structure of a container is threaded through the elements of the container with no additional indirection. Since the C++ compiler expands templates, the time and space overheads introduced by the wrappers is minimal.

6.1 Microbenchmarks

We implemented a selection of small benchmarks; here we focus on just one based on directed graphs.

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 $src, dst \rightarrow weight$. 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`

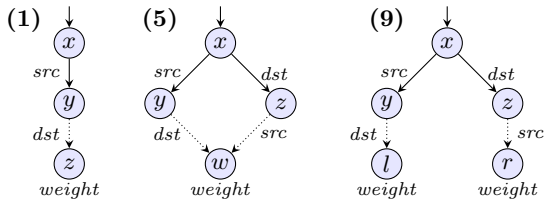

Figure 5: Decompositions 1, 5 and 9 from Figure 4. Solid edges represent instances of `boost::intrusive::map`, dotted edges represent instances of `boost::intrusive::list`.

with methods corresponding to each relational operation. A typical client of the relational interface is the algorithm to perform a depth-first search:

```
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();
    }
  }
}
```

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 auto-tuner framework to evaluate three variants of the graph benchmark under different decompositions. We used a single input graph 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 auto-tuner).

Timing results for decompositions that completed within an 8 second time limit are shown in Figure 4. 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 auto-tuner 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 5 shows three representative decompositions from those shown in Figure 4 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.

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

Table 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.

6.2 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.

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.

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, 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

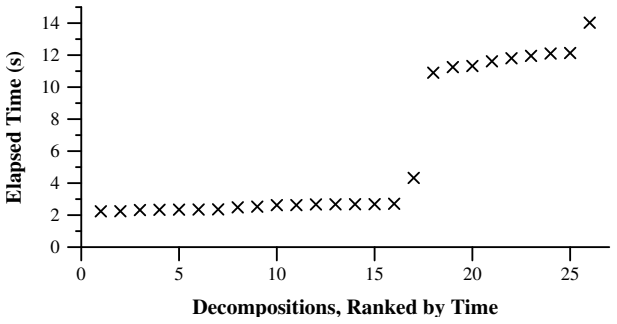

Figure 6: Elapsed time for IpCap to log 3×10^5 random packets for 26 decompositions up to size 4 generated by the auto-tuner, ranked by elapsed time. The 58 decompositions not shown did not complete within 30 seconds.

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 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 auto-tuner on the Ipcap benchmark to generate all decompositions up to size 4; Figure 6 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× 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-written code.

Our experiments show that different choices of decomposition lead to significant changes in performance (Section 6.1), that we can tune the representation based on evolving workloads, and that the best performance is comparable to existing hand-written implementations (Section 6.2). The resulting code is concise (Sections 6.1 and 6.2), and the soundness of the compiler (Theorem 2) guarantees that the resulting data structures are correct by construction.

7. DISCUSSION AND RELATED WORK

Databases and Relational Programming.

Literature on lightweight databases also advocates a programming model based on relations, which are implemented in the backend using container data structures [7, 22, 3, 2]. A novel aspect of our approach is that our relations can have specified restrictions (specifically, functional dependencies). These restrictions, together with the fact that in our compilation context the set of possible queries is known in advance, enable a wider range of possible implementations than a traditional database, particularly representations using complex patterns of sharing. Also new in our work is the notion of adequate decompositions and a proof that operations on adequate decompositions are sound with respect to their relational specifications. Our definition of adequacy is informed in particular by the classic Boyce-Codd Normal Form in relational schema design.

Unlike previous lightweight database work, we describe a dynamic auto-tuner 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 auto-tuner framework has a similar goal to AutoAdmin [6]. 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.

Many authors propose adding relations to both general- and special-purpose programming languages (e.g., [5, 17, 19, 23]). We focus on the orthogonal problem of specifying and implementing the underlying representations for relational data. Data models such as E/R diagrams and UML also rely heavily on relations. One potential application of our technique is to close the gap between modeling languages and implementations.

Synthesizing Data Representations.

The problem of automatic data structure selection was explored in SETL [20] and has also been pursued for Java collection implementations [21]. The SETL representation sublanguage [8] 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 complex 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.

Synthesizing specialized data representations has previously been considered in other domains. Ahmed et al. [1] proposed transforming dense matrix computations into implementations tailored to specific sparse representations as a technique for handling the proliferation of complicated sparse representations.

Synthesis Versus Verification Approaches.

A key advantage of data representation synthesis over hand-written implementations is the synthesized operations are correct by construction, subject to the correctness of

the compiler. We assume the existence of a library of data structures; the data structures themselves can be proved correct using existing techniques [24]. Our system provides a modular way to assemble individually correct data structures into a complete and correct representation of a program's data.

The Hob system uses abstract sets of objects to specify and verify end-to-end properties of systems using multiple data structures that share objects [14]. Monotonic typestates enable aliased objects to monotonically change their typestates in the presence of sharing without violating type safety [10]. Researchers have developed systems that have mechanically verified data structures (for example, hash tables) that implement binary relational interfaces (e.g., [24]). The relation implementation presented in this paper is more general (it can implement relations of arbitrary arity) and solves problems orthogonal to those addressed in previous research.

Specifying And Inferring Shared Representations.

The decomposition language provides a "functional" description of the heap that separates the problem of modeling individual data structures from the problem of modeling the heap as a whole. Unlike previous work, decompositions allow us to state and reason about complex sharing invariants that are difficult to state and impossible to verify using previous techniques. Separation logic allows elegant specifications of disjoint data structures [18], and mechanisms have been added to separation logic to express some types of sharing [4, 9]. Some static analysis algorithms infer some sharing between data structures in low level code [13, 12, 16, 15]; however verifying overlapping shared data structures in general remains an open problem for such approaches. The combination of relations and functional dependencies allows us to reason about sharing that is beyond current static analysis techniques.

8. CONCLUSION

We have presented a system for specifying and operating on data at a high level as relations while correctly compiling those relations into a composition of low-level data structures. Most unusual is our ability to express, and prove correct, the use of complex sharing in the low-level representation. We show using three real-world systems that data representation synthesis leads to code that is simpler, correct by construction, and comparable in performance to existing hand-written code.

9. ACKNOWLEDGEMENTS

This work was supported by NSF grants CCF-0702681 and CNS-050955. The fourth author thanks Viktor Kuncak, Patrick Lam, Darko Marinov, Alex Sălcianu, and Karen Zee for discussions in 2001–2 on programming with relations, with the relations implemented by automatically-generated linked data structures. The second author thanks Daniel S. Wilkerson and Simon F. Goldsmith for discussions in 2005 on Wilkerson's language proposal Orth, which includes relational specifications of data structures, the generation of functions for querying and maintaining them, and was further envisioned by Goldsmith to perform automatic data structure selection via profiling.

10. REFERENCES

[1] N. Ahmed, N. Mateev, K. Pingali, and P. Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Supercomputing*, page 58. IEEE Computer Society, Nov. 2000.

[2] D. Batory, G. Chen, E. Robertson, and T. Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5):441–452, May 2000.

[3] D. Batory and J. Thomas. P2: A lightweight DBMS generator. *Journal of Intelligent Information Systems*, 9:107–123, 1997.

[4] J. Berdine, C. Calgano, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer Berlin / Heidelberg, 2007.

[5] G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *ECOOP*, volume 3586 of *LNCS*, pages 262–286. Springer Berlin / Heidelberg, 2005.

[6] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers.

[7] D. Cohen and N. Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53–60, May 1993.

[8] R. B. K. Dewar, A. Grand, S.-C. Liu, J. T. Schwartz, and E. Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Program. Lang. Syst.*, 1(1):27–49, January 1979.

[9] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, New York, NY, USA, 2008. ACM.

[10] M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Alias Confinement and Ownership*, July 2003.

[11] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49, 2011.

[12] J. Kreiker, H. Seidl, and V. Vojdani. Shape analysis of low-level C with overlapping structures. In *VMCAI*, volume 5044 of *LNCS*, pages 214–230. Springer Berlin / Heidelberg, 2010.

[13] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL*, pages 17–32, New York, NY, USA, 2002. ACM.

[14] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, volume 3385 of *LNCS*, pages 430–447. Springer Berlin / Heidelberg, 2005.

[15] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer Berlin / Heidelberg, 2011.

[16] B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Static Analysis*, volume 6337 of *LNCS*, pages 71–99. Springer Berlin / Heidelberg, 2011.

[17] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *SIGMOD*, pages 706–706, New York, NY, USA, 2006. ACM.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. Invited paper.

[19] T. Rothamel and Y. A. Liu. Efficient implementation of tuple pattern based retrieval. In *PEPM*, pages 81–90, New York, NY, USA, 2007. ACM.

[20] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in SETL. In *POPL*, pages 197–210, New York, NY, USA, 1979. ACM.

[21] O. Shacham, M. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. In *PLDI*, pages 408–418, New York, NY, USA, 2009. ACM.

[22] Y. Smaragdakis and D. Batory. DiSTiL: a transformation library for data structures. In *Conference on Domain-Specific Languages (DSL '97)*, pages 257–271.

USENIX, Oct. 1997.

[23] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *ECOOP*, volume 4609 of *LNCS*, pages 54–78. Springer Berlin / Heidelberg, 2007.

[24] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, New York, NY, USA, 2008. ACM.