Data Representation Synthesis *

Peter Hawkins
Computer Science Department, Stanford University
hawkinsp@cs.stanford.edu

Alex Aiken
Computer Science Department, Stanford University
aiken@cs.stanford.edu

Kathleen Fisher
Computer Science Department, Tufts University †
kfisher@eecs.tufts.edu

Martin Rinard
MIT Computer Science and Artificial Intelligence Laboratory
rinard@csail.mit.edu

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

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 permit the user to specify different concrete representations for relations, and show that operations on concrete representations soundly implement their relational specification. 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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types, Data types and structures; E.2 [*Data Storage Representations*]

General Terms Languages, Design, Algorithms, Performance, Verification

Keywords Synthesis, Composite Data Structures

1. Introduction

One of the first things a programmer must do when implementing a system is commit to particular choices of data structures. 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.

* This work was supported by NSF grants CCF-0702681 and CNS-050955.

† Kathleen was at AT&T Labs Research when this research was done.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

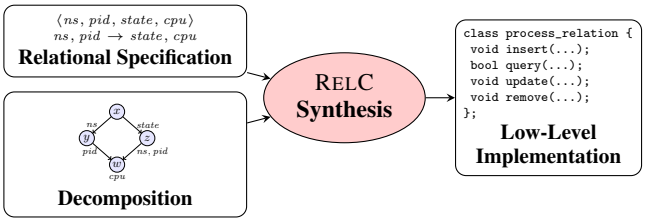

Figure 1. Schematic of Data Representation Synthesis.

Whatever our 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. 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 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.

We propose a new method termed *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* 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.

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.

We build on our previous work [12], which introduced the idea of synthesizing shared low-level data structures from a high-level relational description. Our theoretical framework is substantially simpler and more flexible. In particular, we can handle destructive updates to relations while still preserving all relation invariants. We have also implemented a compiler for relational specifications and an autotuner that finds the best decomposition for a relation automatically. Finally, we have integrated synthesized data representations into existing C++ code as a proof of concept.

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

- We describe a new scheme for synthesizing efficient low-level data representations from abstract relational descriptions of data (Section 2). We describe a relational interface that abstracts data from its concrete representation.
- The decomposition language (Section 3) 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 4). 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 5), which given a relational specification and a performance metric finds the best decomposition up to a user-specified bound.
- The compiler RELC (Section 6) 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.

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 *relational specification* is a set of column names C and functional dependencies Δ. In the scheduler example from Section 1 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 $ns, pid \to state, cpu$, 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:

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

Functional Dependencies A relation r has a *functional dependency* (FD) $C_1 \to C_2$ if any pair of tuples in r that are equal on columns C_1 are also equal on columns C_2. 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.

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; $\mathsf{ref}\ x$ 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, $\mathsf{empty}\ ()$ creates a new empty relation. The operation $\mathsf{insert}\ r\ t$ inserts tuple t into relation r, $\mathsf{remove}\ r\ s$ removes tuples matching tuple s from relation r, and $\mathsf{update}\ r\ s\ u$ applies the updates in tuple u to each tuple matching s in relation r. Finally $\mathsf{query}\ r\ s\ C$ 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 $\mathsf{empty}\ ()$ 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

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

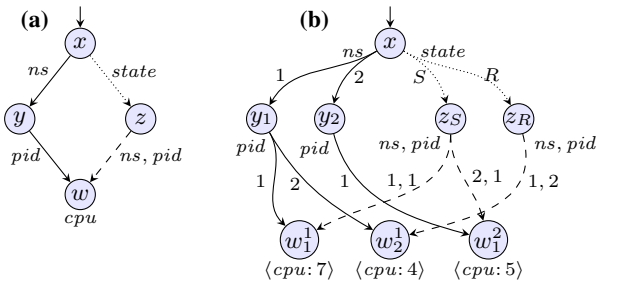

Figure 2. Data representation for a process scheduler: **(a)** a decomposition, **(b)** an instance of that decomposition. Solid edges represent hash tables, dotted edges represent vectors, and dashed edges represent doubly-linked lists.

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

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

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:

```
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);
  ...
};
```

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.

3. 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.

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.

$$\begin{aligned}
\hat{p} &::= C \mid C \overset{\psi}{\mapsto} v \mid \hat{p}_1 \bowtie \hat{p}_2 && \text{decomposition primitives}\\
\hat{d} &::= \mathsf{let}\ v\colon C_1 \triangleright C_2 = \hat{p}\ \mathsf{in}\ \hat{d} \mid v && \text{decompositions}\\
\psi &::= \mathsf{dlist} \mid \mathsf{htable} \mid \mathsf{vector} \mid \cdots && \text{data structures}
\end{aligned}$$

Figure 3. The decomposition language.

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

Figure 4. Decomposition instances.

We use the scheduler example to explain the features of the decomposition language. 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.

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. Note, 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.

To reason formally about decompositions and decomposition instances we encode graphs in a let-binding notation, using the language shown in Figure 3 for decompositions and Figure 4 for instances. We stress that this notation is isomorphic to the graph notation and only exists to aid formal reasoning.

The decomposition of Figure 2(a) written in let-notation is:

$$\begin{aligned}
&\mathsf{let}\ w\colon \{ns, pid, state\} \triangleright \{cpu\} = \{cpu\}\ \mathsf{in}\\
&\mathsf{let}\ y\colon \{ns\} \triangleright \{pid, cpu\} = \{pid\} \overset{\mathsf{htable}}{\longmapsto} w\ \mathsf{in}\\
&\mathsf{let}\ z\colon \{state\} \triangleright \{ns, pid, cpu\} = \{ns, pid\} \overset{\mathsf{dlist}}{\longmapsto} w\ \mathsf{in}\\
&\mathsf{let}\ x\colon \emptyset \triangleright \{ns, pid, cpu, state\} =\\
&\qquad (\{ns\} \overset{\mathsf{htable}}{\longmapsto} y) \bowtie (\{state\} \overset{\mathsf{vector}}{\longmapsto} z)\ \mathsf{in}\ x
\end{aligned} \tag{2}$$

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 $\mathsf{let}\ v\colon B \triangleright C =$

$\hat{p}$ 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 such instance represents a relation with columns C. Figure 2(b) written in the let-notation of instances is:

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

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 of columns B. For example, 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.

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 nodes labeled with columns C. For example, in Figure 2(a) node w is a unit decomposition containing a single cpu value.
- A *map* $C \overset{\psi}{\mapsto} v$ 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 the decomposition v. 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 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(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(a) node x has two outgoing map edges and hence is the join of two map decompositions.

3.2 Abstraction Function

The abstraction functions $\alpha(d, \Gamma)$ and $\alpha(p, \Gamma)$ 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.

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

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

$$\text{(WfJoin)}\ \frac{\Gamma, p_1 \models \hat{\Gamma}, \hat{p}_1 \qquad \Gamma, p_2 \models \hat{\Gamma}, \hat{p}_2 \qquad r_1 = \alpha(p_1, \Gamma) \qquad r_2 = \alpha(p_2, \Gamma) \qquad \pi_{\operatorname{dom} r_2}\, r_1 = \pi_{\operatorname{dom} r_1}\, r_2}{\Gamma, p_1 \bowtie p_2 \models \hat{\Gamma}, \hat{p}_1 \bowtie \hat{p}_2} \qquad \text{(WfVar)}\ \frac{\Gamma, \Gamma(v_t) \models \hat{\Gamma}, \hat{\Gamma}(v)}{\Gamma, v_t \models \hat{\Gamma}, v}$$

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

Figure 5. Well-formed instances: $\Gamma, d \models \hat{\Gamma}, \hat{d}$ and $\Gamma, p \models \hat{\Gamma}, \hat{p}$

3.3 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 $\hat{d}$ if $\cdot, d \models \cdot, \hat{d}$ follows from the rules given in Figure 5. 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.

3.4 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 $\hat{d}$ in Figure 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 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 6. The first argument to the judgement is an environment Σ that maps a variable v bound in the context to a pair $B \triangleright C$, where

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

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

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

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

Figure 6. Adequacy rules: $\Sigma; A \vdash_{a,\Delta} \hat{d}; B$ and $\Sigma; A \vdash_{a,\Delta} \hat{p}; B$

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 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 $\alpha(d, \cdot) = r$.

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 (2) 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.

4. Querying and Updating Decomposed Relations

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.

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

Figure 7. Query Plan Operators

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: *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 4.2. We describe the query planner implementation of the RelC compiler in Section 4.3.

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. 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.

We describe the semantics of query execution using a function $\mathsf{dqexec}\ q\ d\ t$ 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 7. 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 ψ. In general, we expect qlookup to have better time complexity than qscan.

Left/Right The $\mathsf{qlr}(q, lr)$ 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 $\mathsf{qjoin}(q_1, q_2, lr)$ 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.

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

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

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.

4.2 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 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 8 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.

$$\text{(QUNIT)}\ \frac{}{\hat{\Gamma}, C, A \vdash_{q,\Delta} \mathsf{qunit}, C} \qquad \text{(QSCAN)}\ \frac{\hat{\Gamma}, \hat{\Gamma}(v), (A \cup C) \vdash_{q,\Delta} q, B}{\hat{\Gamma}, C \mapsto v, A \vdash_{q,\Delta} \mathsf{qscan}(q), B \cup C}$$

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

$$\text{(QJOIN)}\ \frac{\hat{\Gamma}, \hat{p}_1, A \vdash_{q,\Delta} q_1, B_1 \qquad \hat{\Gamma}, \hat{p}_2, A \cup B_1 \vdash_{q,\Delta} q_2, B_2 \qquad \Delta \vdash_{\mathrm{fd}} A \cup B_1 \to B_2 \qquad \Delta \vdash_{\mathrm{fd}} A \cup B_2 \to B_1}{\hat{\Gamma}, \hat{p}_1 \bowtie \hat{p}_2, A \vdash_{q,\Delta} \mathsf{qjoin}(q_1, q_2, \mathsf{left}), B_1 \cup B_2 \quad \text{or } \hat{\Gamma}, \hat{p}_2 \bowtie \hat{p}_1, A \vdash_{q,\Delta} \mathsf{qjoin}(q_2, q_1, \mathsf{right}), B_1 \cup B_2}$$

$$\text{(QLR)}\ \frac{\hat{\Gamma}, \hat{p}_1, A \vdash_{q,\Delta} q, B}{\hat{\Gamma}, \hat{p}_1 \bowtie \hat{p}_2, A \vdash_{q,\Delta} \mathsf{qlr}(q, \mathsf{left}), B \quad \text{or } \hat{\Gamma}, \hat{p}_2 \bowtie \hat{p}_1, A \vdash_{q,\Delta} \mathsf{qlr}(q, \mathsf{right}), B}$$

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

Figure 8. Validity of query plans: $\hat{\Gamma}, \hat{d}, A \vdash_{q,\Delta} q, B$

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 (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 $\operatorname{dom} s = A$ we have

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

4.3 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

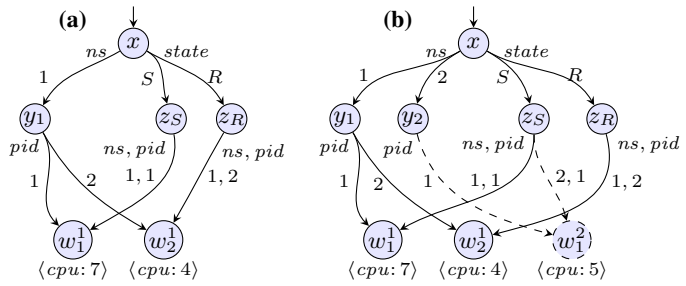

Figure 9. Example of insertion and removal. Inserting the tuple $t = \langle ns\colon 2, pid\colon 1, state\colon S, cpu\colon 5\rangle$ into instance (a) produces instance (b); conversely removing tuple t from (b) produces (a). Differences between the instances are shown using dashed lines.

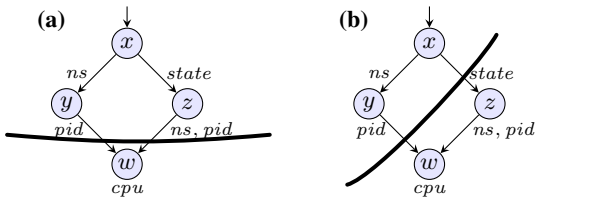

Figure 10. Two cuts of a decomposition: (a) the cut for columns $\{ns, pid\}$, and (b) the cut for columns $\{state\}$.

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:

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

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.

4.4 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.

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 9(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.

4.5 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 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, $\Delta \vdash_{\mathrm{fd}} A \to C$, and nodes $x_B \in X$ that may form part of the representation of tuples that do not match t, that is $\Delta \vdash_{\mathrm{fd}} B \not\to C$. Figure 10 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 9(b). Tuple t has the domain $C = \{ns, pid\}$; Figure 10(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 9(b). Once the dashed edges/nodes in Figure 9 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 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.

4.6 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 5) by induction using initialization and preservation lemmas.

Lemma 3 (Decomposition Initialization)**.** For any decomposition $\hat{d}$, if $d = \mathsf{dempty}\ \hat{d}$ then $\cdot, d \models \cdot, \hat{d}$ and $\alpha(d, \cdot) = \emptyset$.

Lemma 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 $\alpha(d', \cdot) = r'$.

Theorem 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 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 $\alpha(d, \cdot) = r$.

5. 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.

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 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 [4], namely both non-intrusive and intrusive doubly-linked lists (`std::list`, `boost::intrusive::list`), non-intrusive and intrusive binary trees (`std::map`, `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.

6.1 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. For space reasons, 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 rela-

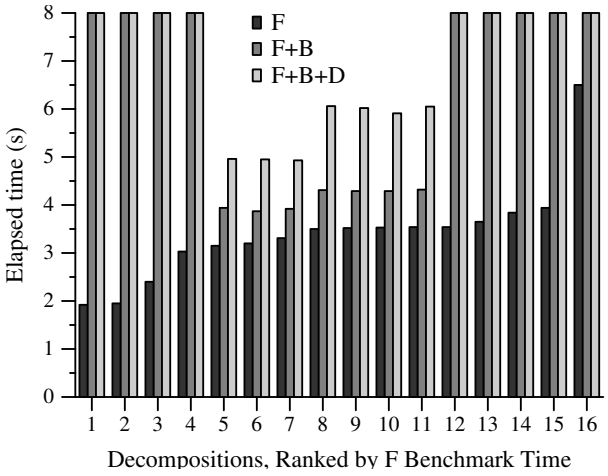

Figure 11. Elapsed times for directed graph benchmarks for decompositions up to size 4 with identical input. For each decomposition we show the times to traverse the graph forwards (F), to traverse both forwards and backwards (F+B), and to traverse forwards, backwards and delete each edge (F+B+D). We elide 68 decompositions which did not finish a benchmark within 8 seconds.

tional 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 autotuner 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 autotuner).

Timing results for decompositions that completed within an 8 second time limit are shown in Figure 11. Decompositions that

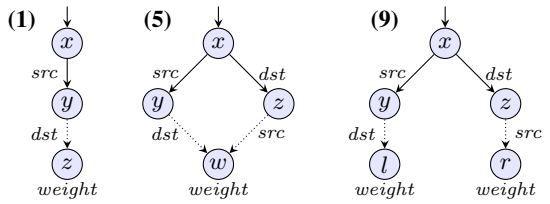

Figure 12. Decompositions 1, 5 and 9 from Figure 11. Solid edges represent instances of `boost::intrusive::map`, dotted edges represent instances of `boost::intrusive::list`.

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 12 shows three representative decompositions from those shown in Figure 11 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.

6.2 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 [21].

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

	Original		Synthesis	
System	**Everything**	**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.

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 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 13 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 6.1), 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 5) guarantees that the resulting data structures are correct by construction.

7. Discussion and Related Work

We build on our previous work [12], which introduced the idea of synthesizing shared low-level data structures from a high-level relational description. We decompose relations directly using graphs, rather than first decomposing relations into trees that are then fused into graphs. As a consequence our theoretical framework is much simpler. We describe a complete query planning implementation, and we show how to reuse the query planning infrastructure to perform efficient destructive updates using graph cuts. We present a compiler that can synthesize efficient C++ implementations of relational operations; previous work only described a proof-of-concept simulator. We present an autotuner that automatically infers the best decomposition for a relation. Finally, using three real examples we show that synthesis leads to code that is simpler, guaranteed to be correct, and comparable in performance to the code it replaces.

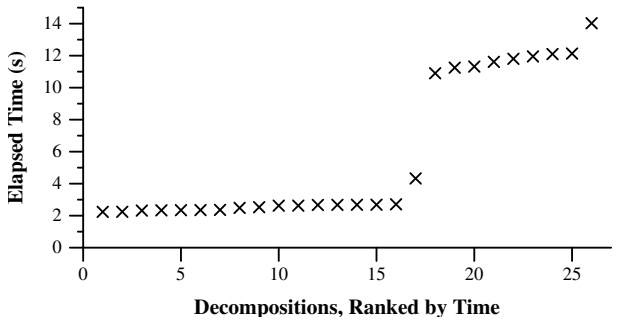

Figure 13. Elapsed time for IpCap to log 3×10^5 random packets for 26 decompositions up to size 4 generated by the auto-tuner, ranked by elapsed time. The 58 decompositions not shown did not complete within 30 seconds.

Relational Representations Many authors propose adding relations to both general- and special-purpose programming languages (e.g., [3, 22, 23, 26, 30]). We focus on the orthogonal problem of specifying and implementing the underlying representations for relational data. Relational representations are well-known from the database community; however, databases typically treat the relations as a black box. Many extensions of our system are possible, motivated by the extensive database literature. Data models such as E/R diagrams and UML rely heavily on relations. One application of our technique is to close the gap between modeling languages and implementations.

The autotuner 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.

Synthesizing Data Representations The problem of automatic data structure selection was explored in SETL [5, 24, 27] and has also been pursued for Java collection implementations [28]. The SETL representation sublanguage [9] 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.

Previous work has proposed using a programming model based on relations which is implemented in the backend using container data structures [8, 29]. A novel aspect of our approach is that our relations can have specified restrictions (specifically, functional dependencies) which enable a much wider range of possible implementations, including complex patterns of sharing. We also present the first formal results, including the notion of adequate decompositions and a proof that operations on adequate decompositions are sound with respect to their relational specifications. Unlike previous work, we propose 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.

Synthesizing specialized data representations has previously been considered in other domains. Ahmed et al. [1, 15] 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 [31]. 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 [17, 19]. Monotonic typestates enable aliased objects to monotonically change their typestates in the presence of sharing without violating type safety [11]. Researchers have developed systems that have mechanically verified data structures (for example, hash tables) that implement binary relational interfaces [7, 31, 32]. 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. Previous work investigated modular reasoning about data structures shared between different modules [13]. Graph types [14] extend tree-structured types with extra pointers that are functionally determined by the structure of the tree backbone, but cannot reason about overlapping structures. Separation logic allows elegant specifications of disjoint data structures [25], and mechanisms have been added to separation logic to express some types of sharing [2, 10]. Some static analysis algorithms infer some sharing between data structures in low level code [16, 18, 20, 21]; 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.

Acknowledgments

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.

References

[1] 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.

[2] Josh Berdine, Cristiano Calgano, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73368-3_22.

[3] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, volume 3586 of *LNCS*, pages 262–286. Springer Berlin / Heidelberg, 2005. doi: 10.1007/11531142_12.

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

[5] Jiazhen Cai and Robert A. Paige. "Look ma, no hashing, and no arrays neither". In *POPL*, pages 143–154, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99605.

[6] Surajit Chaudhuri and Vivek 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. ISBN 1-55860-470-7.

[7] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596565.

[8] Donald Cohen and Neil Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53–60, May 1993. ISSN 0740-7459. doi: 10.1109/52.210604.

[9] 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 Trans. Program. Lang. Syst.*, 1(1):27–49, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357064.

[10] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449782.

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

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data structure fusion. In *APLAS*, volume 6461 of *LNCS*, pages 204–221. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-17164-2_15.

[13] 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.

[14] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *POPL*, pages 196–205, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158628.

[15] 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 *LNCS*, pages 318–327. Springer Berlin / Heidelberg, 1997. doi: 10.1007/BFb0002751.

[16] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level C with overlapping structures. In *VMCAI*, volume 5044 of *LNCS*, pages 214–230. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-11319-2_17.

[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.

[18] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *POPL*, pages 17–32, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503276.

[19] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, volume 3385 of *LNCS*, pages 430–447. Springer Berlin / Heidelberg, 2005. doi: 10.1007/978-3-540-30579-8_28.

[20] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, LNCS, 2011. To appear.

[21] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Static Analysis*, volume 6337 of *LNCS*, pages 71–99. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-15769-1_6.

[22] 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.

[23] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726.

[24] 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.

[25] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817. Invited paper.

[26] Tom Rothamel and Yanhong A. Liu. Efficient implementation of tuple pattern based retrieval. In *PEPM*, pages 81–90, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: 10.1145/1244381.1244394.

[27] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *POPL*, pages 197–210, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567771.

[28] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *PLDI*, pages 408–418, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542522.

[29] Yannis Smaragdakis and Don Batory. DiSTiL: a transformation library for data structures. In *Conference on Domain-Specific Languages (DSL '97)*, pages 257–271. USENIX, October 1997.

[30] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP*, volume 4609 of *LNCS*, pages 54–78. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73589-2_4.

[31] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375624.

[32] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542514.