Data Structure Fusion

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv

Stanford University, AT&T Labs Research, MIT, Tel Aviv University

Abstract. We consider the problem of specifying 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; a novel *fuse* operation on relational indexes specifies where the underlying physical data structure representation has sharing. We permit the user to specify different concrete shared representations for relations, and show that the semantics of the relational specification are preserved.

1 Introduction

Consider the data structure used in an operating system kernel to represent the set of available file systems. There are two kinds of objects: *file systems* and *files*. Each file system has a list of its files, and each file may be in one of two states, either currently *in use* or currently *unused*. Figure 1 sketches the data structure typically used:[1] each file system is the head of a linked list of its files, and two other linked lists maintain the set of files in use and files not in use. Thus, every file participates in two lists: the list of files in its file system, and one of the in-use or not-in-use lists. A characteristic feature of this example is the sharing: the files participate in multiple data structures. Sharing usually implies that there are non-trivial high-level invariants to be maintained when the structure is updated. For example, in Figure 1, if a file is removed from a file system, it should be removed from the in-use or not-in-use list as well. A second characteristic is that the structure is highly optimized for a particular expected usage pattern. In Figure 1, it is easy to enumerate all of the files in a file system, but without adding a parent pointer to the file objects we have only a very slow way to discover which file system owns a particular file.

We are interested in the problem of how to support high-level, declarative specification of complex data structures with sharing while also achieving efficient and safe low-level implementations. Existing languages provide at most one or the other. Modern functional languages provide excellent support for inductive data structures, which are all essentially trees of some flavor. When multiple such data structures overlap (i.e., when there is more than one inductive structure and they are not separate), functional languages do not provide any support beyond what is available in conventional object-oriented and procedural languages. All of these languages require the programmer to build and maintain mutable structures with sharing by using explicit pointers or reference

[1] This example is a simplified version of the file system representation in Linux, where file systems are called *superblocks* and files are *inodes*.

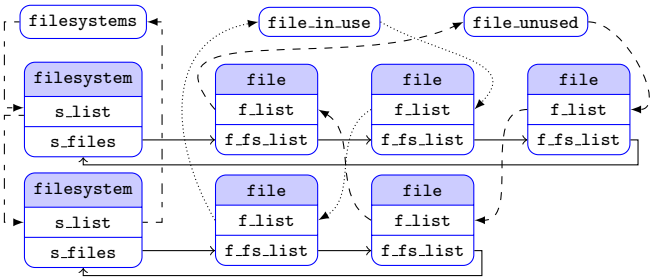

Fig. 1. `File` objects simultaneously participate in multiple circular lists. Different line types denote different lists.

cells. While the programmer can get exactly the desired representation, there is no support for maintaining or even describing invariants of the data structure.

Languages built on relations, such as SQL and logic programming languages, provide much higher-level support. We could encode the example above using the relation:

$$\text{file}(\mathit{filesystem} : \mathsf{int}, \mathit{fileid} : \mathsf{int}, \mathit{inuse} : \mathsf{bool})$$

Here integers suffice as unique identifiers for file systems and files, and a boolean records whether or not the file is in use. Using standard query facilities we can conveniently find for a file system fs all of its files $\text{file}(fs, \_, \_)$ as well as all of the files not in use $\text{file}(\_, \_, \mathsf{false})$. Even better, using *functional dependencies* we can specify important high-level invariants, such as that every file is part of exactly one file system, and every file is either in use or not; i.e., the *fileid* functionally determines the *filesystem* and *inuse* fields. Thus, there is only one tuple in the relation per *fileid*, and when the tuple with a *fileid* is deleted all trace of that file is provably removed from the relation. Finally, relations are general; since pointers are just relationships between objects, any pointer data structure can be described by a set of relations. Adding relations to general-purpose programming languages is a well-accepted idea. Missing from existing proposals is the ability to provide highly specialized implementations of relations, and in particular to take advantage of the potential for mutable data structures with sharing.

Our vision is a programming language where low-level pointer data structures are specified using high-level relations. Furthermore, because of the high-level specification, the language system can produce code that is correct by construction; even in cases where the implementation has complex sharing and destructive update, the implementation is guaranteed to be a faithful representation of the relational specification. In this paper, we take only the first step in realizing this plan, focusing on the core problem of what it means to represent a given high-level relation by a low-level representation (possibly with sharing) that is provably correct. We do not address in this paper the design of a surface syntax for integrating relational operations into a full programming language (there are many existing proposals).

This paper is organized into several parts, each of which highlights a separate contribution of our work:

– We begin by describing three examples of data structure specification. Our approach separates the semantic content of a data structure from details of

its implementation, while allowing the programmer to control the low-level physical representation (Section 2).
- A key contribution is the design of a language for specifying *indices*, which are a mapping between a relational specification and concrete data structures. (Section 3). This language allows us to define *cross-linking* and *fusion* constructs which, although common in practice, express sharing that is difficult or impossible to express using standard data abstraction techniques.
- We describe adequacy conditions that ensure that the low-level representation of a relation is capable of implementing its higher-level specification.
- We describe the implementations of the core relation primitives, and we prove that the low-level implementations are sound with respect to the higher-level specifications (Section 4 and Section 5).

Due to space limitations we have not included all supporting lemmas or any proofs in this paper. All lemmas and proofs are in the on-line tech report [10].

2 Relation Representations and Indices

In this section we motivate and describe three different representations for relations, at different levels of abstraction, using three examples: directed graphs, a process scheduler, and a Minesweeper game. The highest level is the *logical representation* of a relation, which is the usual mathematical description of a finite relation as a set of tuples. The lowest level is the *physical representation* of a relation, which represents a relation in a program's heap using pointer-based data structures. Bridging the gap we have an intermediate *tree decomposition* of a relation, which decomposes the relation into a tree form corresponding to an index without yet committing to a specific physical representation.

First, we need to fix notation.

Values, Tuples, Relations For our formal development we assume a universe of untyped values $\mathbb{V}$, which includes the integers, that is, $\mathbb{Z} \subseteq \mathbb{V}$. We write v to denote one value, $\mathbf{v}$ for a sequence of values, and V to denote a set of values.

A *tuple* $t = \langle c_1 \mapsto v_1, c_2 \mapsto v_2, \dots \rangle$ is a mapping from a set of *columns* $\{c_1, c_2, \dots\}$ to values drawn from $\mathbb{V}$. We write $t(c)$ to denote the value of column c in tuple t, and we write $t(\mathbf{c})$ to denote the sequence of values corresponding to a sequence of columns. We write $s \subseteq t$ if the tuple t is an extension of tuple s, that is we have $t(c) = s(c)$ for all c in the domain of s. In an abuse of notation we sometimes use a sequence of columns $\mathbf{c}$ as a set. A *relation* r is a set of tuples $\{x, y, z, \dots\}$ over the same set of column names C.

Relational Algebra We use the standard notation of relational algebra [6]: union ($\cup$), intersection ($\cap$), set difference ($\setminus$), selection $\sigma_f\, r$, projection $\pi_C\, r$, projection onto the complement of a set of columns C: $\pi_{\overline{C}}\, r$, and natural join $r_1 \bowtie r_2$; we also allow tuples in place of relations as arguments to relation operators.

2.1 Logical Representation of Relations

We begin with the problem of representing the edges of a weighted directed graph (V, E) where $E \subseteq V \times \mathbb{Z} \times V$. We return to this example throughout the paper. One popular way to represent sparse graphs is as an adjacency list, which

$$\mathsf{empty}_d : \mathsf{unit} \to (\alpha_1, \ldots, \alpha_k)\ \mathsf{relation}_d$$
$$\mathsf{insert}_d : \alpha_1 * \cdots * \alpha_k \to (\alpha_1, \ldots, \alpha_k)\ \mathsf{relation}_d \to \mathsf{unit}$$
$$\mathsf{remove}_d : \alpha_1 * \cdots * \alpha_k \to (\alpha_1, \ldots, \alpha_k)\ \mathsf{relation}_d \to \mathsf{unit}$$
$$\mathsf{query}_d : (\alpha_1, \ldots, \alpha_k)\ \mathsf{relation}_d \to \alpha_1\ \mathsf{option} * \cdots * \alpha_k\ \mathsf{option} \to (\alpha_1 * \cdots * \alpha_k)\ \mathsf{list}$$

Fig. 2. Primitive operations on logical relations

records the list of successors and predecessors of each vertex $v \in V$. In ML, we might represent a graph via adjacency lists as the type

$$\mathsf{type}\ g = (v, (v * \mathsf{int})\ \mathsf{list})\ \mathsf{btree} * (v, (v * \mathsf{int})\ \mathsf{list})\ \mathsf{btree},$$

assuming v is the type of vertices, and (α, β) btree is a binary tree mapping keys of type α to values of type β. Here the graph is represented as two collaborating data structures, namely a binary tree mapping each vertex to a list of its successors, together with the corresponding edge weights, and a binary tree mapping each vertex to a list of its predecessors, and the corresponding edge weights.

One problem with our proposed ML representation is that the successor and predecessor data structures represent the same set of edges; however it is the programmer's responsibility to ensure that the two data structure representations remain consistent. Another problem is that with only tree-like data structures there is no natural place to put the edge weight—we can place it in either the successor data structure or the predecessor data structure, increasing the time complexity of certain queries, or we can duplicate the weight, as we have here, which increases the space cost and introduces the possibility of inconsistencies.

Instead, we can use a relation. We represent the edges of our directed graph as a relation g with three columns $(src, dst, weight)$, in which each tuple represents the source, destination, and weight of an edge. The graph shown in Figure 5(a) can be represented as the relation $\{\langle 1, 2, 17\rangle, \langle 1, 3, 42\rangle\}$. We call the usual mathematical view of a relation as a set of tuples the *logical representation*.

We extend ML with a new type constructor $(\alpha_1, \ldots, \alpha_k)$ relation which represents relations of arity k, together with a set of primitive operations to manipulate relations. Relations are mutable data structures conceptually similar to $(\alpha_1 * \cdots * \alpha_k)$ list ref, with a very different representation. The primitives with which the client programmer manipulates relations, shown in Figure 2, are creating an empty relation, operations to insert and remove tuples from a relation, and query, which returns the list of tuples matching a *tuple pattern*, a tuple in which some fields are missing. We describe a minimal interface to make proofs easier; a practical implementation should provide a richer set of primitives, such as an interface along the lines of LINQ [15].

2.2 Indices and Tree Decompositions

The data structure designer describes how to represent a logical relation using an *index*, which specifies how to decompose the relation into a collection of nested map and join operations over unit relations containing individual tuples. Different decompositions lead to different operations being particularly efficient. We do not maintain an underlying list of tuples; the only representation of a

$$
\begin{aligned}
d &::= \mathsf{unit}(\mathbf{c}) \mid \mathsf{map}(\psi, \mathbf{c}, d') \mid \mathsf{join}(d_1, d_2, L) && \text{indices}\\
\psi &::= \mathsf{option} \mid \mathsf{slist} \mid \mathsf{dlist} \mid \mathsf{btree} && \text{data struct.}\\
l \in L &::= (\mathsf{fuse}, \mathbf{z}_1, \mathbf{z}_2) \mid (\mathsf{link}, \mathbf{z}_1, \mathbf{z}_2) && \text{cross-links}\\
\mathbf{z} \in \mathit{contour} &::= \{\mathtt{m}, \mathtt{l}, \mathtt{r}\}^* && \text{stat. contours}\\
\mathbf{y} \in \mathit{dcontour} &::= \{\mathtt{m}_{\mathbf{v}}, \mathtt{l}, \mathtt{r}\}^* && \text{dyn. contours}
\end{aligned}
$$

Fig. 3. Syntax of indices

relation is that described by an index. Beyond the index definition programmers can remain oblivious of details of how relations are represented.

Every relation r has an associated *index* d describing how to decompose the relation into a tree and how to lay that tree out in memory; Figure 3 shows the syntax of indices. Given an index d and a relation r we can form a *tree decomposition* ρ whose structure is governed by d; Figure 4 defines the syntax of tree decompositions. There are three kinds of index that we can use to decompose a relation, each of which has a corresponding kind of tree-decomposition node:

- *Joins* allow the data-structure designer to specify how to divide the relation into pieces. These pieces can have different structures, each supporting different access patterns efficiently. We require that the natural join of the pieces be equal to the original relation. Formally, a $\mathsf{join}(d_1, d_2, L)$ index represents a relation as the natural join of two different sub-relations (ρ_1, ρ_2), where d_1 is an index that describes how to represent ρ_1 and d_2 is an index that describes how to represent ρ_2. The set L consists of *cross-linking* and *fusion* declarations, which we will describe shortly.
- *Maps* allow the data-structure designer to specify that certain columns of the relation can be used to lookup other columns. The map operator allows the programmer to specify the data structure ψ that should be used for this mapping, with options including singly- and doubly-linked lists and binary trees. Formally, a $\mathsf{map}(\psi, \mathbf{c}, d')$ index represents a relation as a mapping $\{\mathbf{v}_i \mapsto \rho_i\}_{i \in I}$ from a sequence of *key* columns $\mathbf{c}$ to a set of *residual relations* ρ_i, one for each valuation $\mathbf{v}_i$ of the key columns. We further decompose each residual relation ρ_i using an index d'.
- *Unit* indices are the base case, and represent individual tuples. Formally, a $\mathsf{unit}(\mathbf{c})$ index represents a relation over a sequence of columns $\mathbf{c}$ with cardinality either 0 or 1; such a relation can either be the empty set $\{\}$, or contain a single sequence of values $\{\mathbf{v}\}$.

We assume we are given correct implementations of a set of primitive data structures such as singly- and doubly-linked lists and trees. Our focus is on assembling such building blocks into nested and overlapping data structures.

Static Contours We annotate each term in the index with a unique name called a *static contour*. Formally, a static contour $\mathbf{z}$ is a path in an index d which identifies a specific sub-index d'. A static contour $\mathbf{z}$ is drawn from the set $\{\mathtt{m}, \mathtt{l}, \mathtt{r}\}^*$, where `m` means "move to the child index of a map index", `l` means "move to the left sub-index of a join index", and `r` means "move to the right sub-index of a join index". We write $d.\mathbf{z}$ to denote the sub-index of d identified by a contour $\mathbf{z}$.

In our directed graph we want to find the set of successors and find the set of predecessors of a vertex efficiently. One index that satisfies this constraint is

$$\rho ::= \{\} \mid \{\mathbf{v}\} \mid \{\mathbf{v}_i \mapsto \rho_i\}_{i \in I} \mid (\rho_1, \rho_2)$$

Fig. 4. Tree decompositions

$$d_g = \mathsf{join}^{\cdot}(\mathsf{map}^{\mathtt{l}}(\mathsf{btree}, [src], \mathsf{map}^{\mathtt{lm}}(\mathsf{slist}, [dst], \mathsf{unit}^{\mathtt{lmm}}([weight]))),\\ \mathsf{map}^{\mathtt{r}}(\mathsf{btree}, [dst], \mathsf{map}^{\mathtt{rm}}(\mathsf{slist}, [src], \mathsf{unit}^{\mathtt{rmm}}([]))), \{(\mathsf{fuse}, \mathtt{rmm}, \mathtt{lmm})\})$$

The index d_g states that we should represent the relation as the natural join of two sub-indices. The left sub-index is a binary tree mapping each value of the *src* column to a distinct singly-linked list, which in turn maps each *dst* column value (for the given *src*) to its corresponding *weight*. The right sub-index is a binary tree mapping each value of the *dst* column to a linked list of *src* values.

Tree Decompositions An index determines a useful intermediate representation of the associated relation, decomposing it into a tree according to the operations in the index. We call this representation the *tree decomposition* of a relation. As an example, Figure 5(b) depicts the tree decomposition ρ of the graph relation g given index d_g. We write ρ mathematically as

$$\Big(^{\cdot} \Big\{^{\mathtt{l}} 1 \mapsto \{^{\mathtt{lm}_1} 2 \mapsto \{^{\mathtt{lm}_1\mathtt{m}_2}\langle 17\rangle\}, 3 \mapsto \{^{\mathtt{lm}_1\mathtt{m}_3}\langle 42\rangle\}\}\Big\}, \\ \Big\{^{\mathtt{r}} 2 \mapsto \{^{\mathtt{rm}_2} 1 \mapsto \{^{\mathtt{rm}_2\mathtt{m}_1}\langle\rangle\}\}, 3 \mapsto \{^{\mathtt{rm}_3} 1 \mapsto \{^{\mathtt{rm}_3\mathtt{m}_1}\langle\rangle\}\}\Big\}\Big), \tag{1}$$

Dynamic Contours We assign each term of a tree decomposition a unique label, called a *dynamic contour*. A *dynamic contour* $\mathbf{y}$ is a path in a tree decomposition ρ under index d that identifies a specific subtree of ρ. Each dynamic contour in a tree decomposition corresponds to an instance of a static contour in an index. In a dynamic contour we annotate the `m` operator with a sequence of key values $\mathbf{v}$; a tree decomposition via a map index has one subtree for each sequence of key values, and hence when navigating to a subtree we must specify which subtree we mean. We do not need any extra dynamic information for a join index, so we leave the `l` and `r` operators unannotated. For example, the part of the tree labeled `r` corresponds with the sub-index of d_g labeled `r`, and maps *dst* values of the relation to a list of tree decompositions corresponding to index `rm`.

2.3 Physical Representations, Cross-Linking, and Fusion

In Section 2.2 we showed how to represent logical relations as tree-decompositions. Given a relation and an accompanying index, our implementation generates a *physical representation* with the structure given by the index. This representation is the concrete realization of the tree-decomposed relation in memory. Each term in the tree-decomposition becomes an object in memory, and we use the data structures specified in the index to lay out and link those objects together.

Sharing declarations allow the programmer to specify connections between objects in different parts of the index. Such sharing declarations come in two flavors: *fusion* and *cross-linking*. Fuse declarations indicate that the objects should be merged, with each structure containing a pointer to the shared object, while link operations indicate that one structure should contain a pointer to an object in another structure. Effectively these constructs collapse the tree decomposition into a directed acyclic graph.

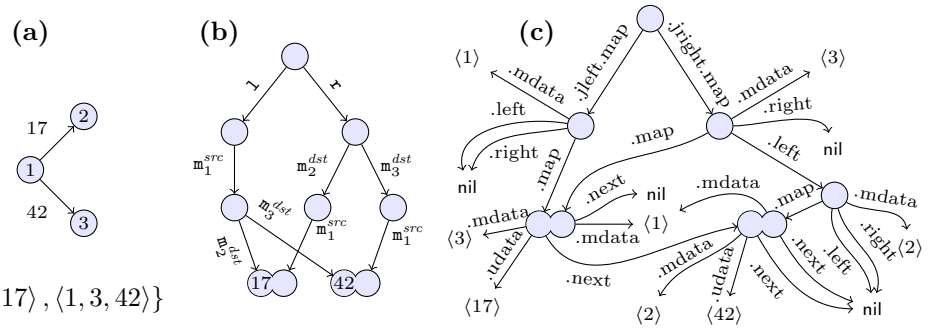

Fig. 5. Representations of a weighted directed graph: **(a)** An example graph, and its representation as a relation, **(b)** A tree decomposition of the relation in (a), with fused data structures shown as conjoined nodes, and **(c)** a diagram of the memory state that represents (b).

In the graph example, we would like to share the weight of each edge between the two representations. Observe that given a (src, dst) pair, the weight is the same whether we traverse the links in the left or the right tree. That is, there is a functional dependency: any (src, dst) pair determines a unique weight, and it does not matter whether we visit the *src* or the *dst* first. Hence instead of replicating the weight, we can share it between the two trees, specified here by the *fuse* declaration. The declaration says that the data structure we get after looking up a *src* and then a *dst* in the left tree should be fused with the data structure we get by looking up a *dst* and then a *src* in the right tree.

Each join index takes an argument L which is a set of cross-linking declarations $(\mathsf{link}, \mathbf{z}_1, \mathbf{z}_2)$ and fusion declarations $(\mathsf{fuse}, \mathbf{z}_1, \mathbf{z}_2)$. A cross-linking declaration $(\mathsf{link}, \mathbf{z}_1, \mathbf{z}_2)$ states that a pointer should be maintained from each object with static contour $\mathbf{z}_1$ to the corresponding object with static contour $\mathbf{z}_2$. Similarly, a fusion declaration $(\mathsf{fuse}, \mathbf{z}_1, \mathbf{z}_2)$ states that objects with static contour $\mathbf{z}_1$ should be placed adjacent to the corresponding object with static contour $\mathbf{z}_2$. By "corresponding" object we mean the object with static contour $\mathbf{z}_2$, whose column values are drawn from the set bound by following static contour $\mathbf{z}_1$.

In the graph example, the contour `rmm` names the data structure we get by looking in the right component of the join (`r`) and then navigating down two map indices (`mm`), *i.e.*, looking in the right tree and then following first the *dst* and then the *src* links. The contour `lmm` names the corresponding location in the left tree. The fuse declaration indicates these two nodes should be merged, with the *weight* data structure from the left tree being fused with the empty data structure from the right tree. Figure 5(b) depicts the index structure after fusion. Figure 5(c) graphically depicts the resulting physical memory state that represents the graph of Figure 5(b). The conjoined nodes in the figure are placed at a constant field offset from one another on the heap.

2.4 Process Scheduler

As another example, suppose we want to represent the data for a simple operating system process scheduler (as in [13]). The scheduler maintains a list of live processes. A live process can be in any one of a number of states, e.g. running or

$$(\text{TWFEMP})\ \{\} \models_T \mathsf{unit}(\mathbf{c}) \qquad (\text{TWFUNIT})\ \frac{|\mathbf{v}| = |\mathbf{c}|}{\{\mathbf{v}\} \models_T \mathsf{unit}(\mathbf{c})}$$

$$(\text{TWFMAP})\ \frac{\forall i \in I.\, |\mathbf{v}_i| = |\mathbf{c}| \qquad \forall i \in I.\, \rho_i \models_T d \qquad \forall i \in I.\, \alpha_t(\rho_i, d) \neq \emptyset}{\{\mathbf{v}_i \mapsto \rho_i\}_{i \in I} \models_T \mathsf{map}(\psi, \mathbf{c}, d)}$$

$$(\text{TWFJOIN})\ \frac{\begin{array}{cc}\rho_1 \models_T d_1 & \alpha_t(\rho_1, d_1) \models \operatorname{dom} d_1 \cap \operatorname{dom} d_2 \to \operatorname{dom} d_1 \setminus \operatorname{dom} d_2 \\ \rho_2 \models_T d_2 & \pi_{\operatorname{dom} d_1 \cap \operatorname{dom} d_2}\, \alpha_t(\rho_1, d_1) = \pi_{\operatorname{dom} d_1 \cap \operatorname{dom} d_2}\, \alpha_t(\rho_2, d_2)\end{array}}{(\rho_1, \rho_2) \models_T \mathsf{join}(d_1, d_2, L)}$$

Fig. 6. Well-formed tree decompositions: $\rho \models_T d$

sleeping. The scheduler also maintains a list of possible process states; for each state we maintain a tree of processes with that state. We represent the scheduler's data by a relation live(pid, $state$, uid, $walltime$, $cputime$), and the index

$$\mathsf{join}^{\cdot}\big(\mathsf{map}^{\mathtt{l}}\big(\mathsf{btree}, [pid], \mathsf{unit}^{\mathtt{lm}}([uid, walltime, cputime])\big),\\ \quad \mathsf{map}^{\mathtt{r}}\big(\mathsf{dlist}, [state], \mathsf{map}^{\mathtt{rm}}(\mathsf{btree}, [pid], \mathsf{unit}^{\mathtt{rmm}}([]))\big), \{(\mathsf{fuse}, \mathtt{rmm}, \mathtt{lm})\}\big)$$

2.5 Minesweeper

Another example is motivated by the game of Minesweeper. A Minesweeper board consists of a 2-dimensional matrix of cells. Each cell may or may not have a mine; each cell may also be concealed or exposed. Every cell starts off in the unexposed state; the goal of the game is to expose all of the cells that do not have mines without exposing a cell containing a mine. Some implementations of Minesweeper also implement a "peek" cheat code that iterates over the set of unexposed cells, temporarily displaying them as exposed. We represent a board by the relation board(x, y, $ismined$, $isexposed$), with the index:

$$\mathsf{join}^{\cdot}\big(\mathsf{map}^{\mathtt{l}}\big(\mathsf{btree}, [x], \mathsf{map}^{\mathtt{lm}}(\mathsf{btree}, [y], \mathsf{unit}^{\mathtt{lmm}}([ismined, isexposed]))\big),\\ \quad \mathsf{map}^{\mathtt{r}}\big(\mathsf{slist}, [isexposed], \mathsf{map}^{\mathtt{rm}}(\mathsf{btree}, [x, y], \mathsf{unit}^{\mathtt{rmm}}([]))\big), \{(\mathsf{link}, \mathtt{rmm}, \mathtt{lmm})\}\big)$$

In this example, the index specifies a *cross-link* rather than a fusion. Cross-linking adds a pointer from one object in a tree decomposition to another object, providing a "short-cut" from one data structure to another.

3 Abstraction, Well-formedness, and Adequacy

In this and subsequent sections we give the details of how we can specify data structures with sharing at a high-level using relations and then faithfully translate those specifications into efficient low-level representations. There are two main complications. First, not every index can represent every relation; we introduce a notion of *adequacy* to characterize which relations an index can represent. Second, our proof strategy requires two steps: first showing that the intermediate *tree decomposition* of a relation is correct with respect to the logical relation, and second showing that the *physical representation* is correct with respect to the tree decomposition (Sections 4 and 5).

$$\text{(LAUnit)}\ \frac{\Delta \vdash_{\text{fd}} \emptyset \to \mathbf{c}}{\mathbf{c}; \Delta \vdash_l \mathsf{unit}(\mathbf{c})} \qquad \text{(LAMap)}\ \frac{C_2; \Delta/\mathbf{c}_1 \vdash_l d}{\mathbf{c}_1 \uplus C_2; \Delta \vdash_l \mathsf{map}(\psi, \mathbf{c}_1, d)}$$

$$\text{(LAJoin)}\ \frac{\Delta \vdash_{\text{fd}} C_1 \to C_2 \qquad C_1 \cup C_2; \Delta \vdash_l d_1 \qquad C_1 \cup C_3; \Delta \vdash_l d_2}{C_1 \uplus C_2 \uplus C_3; \Delta \vdash_l \mathsf{join}(d_1, d_2, L)}$$

$$\text{where } \Delta/C = \left\{(A \setminus C) \to (B \setminus C) \mid (A \to B) \in \Delta\right\}$$

Fig. 7. Rules for logical adequacy $C; \Delta \vdash_l d$

3.1 Tree Decompositions

Abstraction Function Finally, we can relate the pieces we have defined so far. The abstraction function $\alpha_t(\rho, d)$ maps a tree decomposition ρ according to some index d to the corresponding high-level logical relation, showing what relation the tree decomposition represents:

$$\alpha_t(V, \mathsf{unit}(\mathbf{c})) = \{\langle \mathbf{c} \mapsto \mathbf{v} \rangle \mid \mathbf{v} \in V\}$$
$$\alpha_t(\{\mathbf{v}_i \mapsto \rho_i\}_{i \in I}, \mathsf{map}(\psi, \mathbf{c}, d)) = \bigcup\nolimits_{i \in I} \left(\langle \mathbf{c} \mapsto \mathbf{v}_i \rangle \times \alpha(\rho_i, d)\right)$$
$$\alpha_t((\rho_1, \rho_2), \mathsf{join}(d_1, d_2, L)) = \alpha_t(\rho_1, d_1) \bowtie \alpha_t(\rho_2, d_2)$$

Functional Dependencies A relation r has a *functional dependency* (FD) $B \to C$, if any pair of tuples in r that are equal on the set of columns B are also equal on columns C. We write Δ to denote a set of functional dependencies; we write $r \models \Delta$ if a relation r has the set of FDs Δ. If a FD $A \to B$ is a consequence of set of FDs Δ we write $\Delta \vdash_{\text{fd}} A \to B$; sound and complete inference rules for functional dependencies are standard [1].

Well-Formed Decompositions We define a class of well-formed tree decompositions ρ for an index d with a judgment $\rho \models_T d$ shown in Figure 6. The (TWfEmp) and (TWfUnit) check that a unit node is either the empty set or a sequence of values of the right length. The (TWfMap) rule checks that each sequence of key values has the right length, and that there are no key values that map to empty subtrees. The (TWfJoin) rule ensures the relation actually has the functional dependency promised by the adequacy judgment, and that we do not have "dangling" tuples on one side of a join without a matching tuple on the other side. Note that rule (TWfJoin) does not place any restrictions on the fusion declaration L; valid fusions are the subject of the physical adequacy rules of Figure 9. We write $\operatorname{dom} d$ for the set of columns that appear in an index.

3.2 Logical Adequacy

Digressing briefly, we observe that we cannot decompose every relation with every index. In general an index can only represent a class of relations satisfying particular functional dependencies.

For our running graph example the index d_g is not capable of representing every possible relation of three columns. For example, the relation $r' = \{\langle 1, 2, 3\rangle, \langle 1, 2, 4\rangle\}$ cannot be represented, because d_g can only represent a single

$$
\begin{array}{rl|rl}
f \in & \{\mathrm{link}_{(\mathbf{z}_1,\mathbf{z}_2)}, \mathrm{fuse}_{(\mathbf{z}_1,\mathbf{z}_2)}, \ldots\} & \text{field names} & \mathbb{A} = \mathbb{Z} \times f^* & \text{addresses} \\
\mu : & \mathbb{A} \to \mathbb{A} \cup \mathbb{V} & \text{memory} & \Lambda : \mathit{dcontour} \to \mathbb{A} & \text{layout}
\end{array}
$$

Fig. 8. Heaps

weight for each pair of *src* and *dst* vertices. However r' does not correspond to a well-formed graph; all well-formed graphs satisfy a functional dependency $\mathit{src}, \mathit{dst} \to \mathit{weight}$, which allows at most one weight for any pair of vertices.

We say that an index d is *adequate* for a class of relations R if for every relation $r \in R$ there is some tree decomposition ρ such that $\alpha_t(\rho, d) = r$. Figure 7 lists inference rules for a judgment $C; \Delta \vdash_l d$ that is a sufficient condition for an index to be adequate for the class of relations with columns C that satisfy a set of FDs Δ. The inference rules enforce two properties. Firstly, the (LAUnit) and (LAMap) rules ensure that every column of a relation must be represented by the index; every column must appear in a unit or map index. Secondly, in order to split a relation into two parts using a join index, the (LAJoin) rule requires a functional dependency to prevent anomalies such as spurious tuples.

We have the following lemma:

Lemma 1 (Soundness of Adequacy Judgement). *If $C; \Delta \vdash_l d$ then for each relation r with columns C such that $r \models \Delta$ there is some ρ such that $\rho \models_T d$ and $\alpha_t(\rho, d) = r$.*

3.3 Physical Representation

Heaps. Figure 8 defines the syntax for our model of memory. We represent the heap as function μ from a set of heap locations to a set of heap values. Our model of a heap location is based on C structs, except that we abstract away the layout of fields within each heap object. Heap locations are drawn from an infinite set $\mathbb{A}$, and consist of a pair $(n, \mathbf{f})$ of a integer address identifying a heap object, together with a string of field offsets; each integer location notionally has a infinite number of field slots, although we only ever use a small and bounded number, which can then be laid out in consecutive memory locations. The contents of each heap cell can either be a value drawn from $\mathbb{V}$ or an address drawn from $\mathbb{A}$; we assume that the two sets are disjoint.

The set of columns that are bound by following a static contour $\mathbf{z}$ is given by the function $\mathsf{bound}(\mathbf{z}, d)$, defined as

$$
\begin{array}{rr}
\mathsf{bound}(\cdot, d) = \emptyset & \mathsf{bound}(\mathtt{m}\mathbf{z}, \mathsf{map}(\psi, \mathbf{c}, d)) = \mathbf{c} \cup \mathsf{bound}(\mathbf{z}, d) \\
\mathsf{bound}(\mathtt{l}\mathbf{z}, \mathsf{join}(d_1, d_2, L)) = \mathsf{bound}(\mathbf{z}, d_1) & \mathsf{bound}(\mathtt{r}\mathbf{z}, \mathsf{join}(d_1, d_2, L)) = \mathsf{bound}(\mathbf{z}, d_2)
\end{array}
$$

Layouts. We use dynamic contours to name positions in a tree. A *layout* function Λ is a mapping from the dynamic contours of a tree to addresses from $\mathbb{A}$. Layout functions allow us to translate from semantic names for memory locations to a more machine-level description of the heap; the extra layer of indirection allows us to ignore details of memory managers and layout policies, and to describe fusion and cross-linking succinctly. All layouts must be injective; that is, different tree locations must map to different physical locations. We define operators that strip and add prefixes to the domain of a layout

$$\Lambda/x = \{\mathbf{y} \mapsto a \mid (x\mathbf{y} \mapsto a) \in \Lambda\}, \text{ and } \Lambda \times x = \{x\mathbf{y} \mapsto a \mid (\mathbf{y} \mapsto a) \in \Lambda\}.$$

$$\text{(PAUnit)}\quad \Delta;\Phi \vdash_p \mathsf{unit}(\mathbf{c}) \qquad \text{(PAMap)}\quad \frac{\Delta/\mathbf{c}_1;\{\mathbf{x} \mid \mathtt{m}\mathbf{x} \in \Phi\} \vdash_p d}{\Delta;\Phi \vdash_p \mathsf{map}(\psi, \mathbf{c}_1, d)}$$

$$\text{(PAJoin)}\quad \frac{\forall l \in L.\, \Delta;\Phi \vdash_p d; l \qquad \Phi' = \Phi \cup \{\mathbf{z} \mid (\mathsf{fuse}, \mathbf{z}, \mathbf{z}') \in L\} \qquad \Delta;\{\mathbf{x} \mid \mathtt{l}\mathbf{x} \in \Phi'\} \vdash_p d_1 \qquad \Delta;\{\mathbf{x} \mid \mathtt{r}\mathbf{x} \in \Phi'\} \vdash_p d_1}{\Delta;\Phi \vdash_p \mathsf{join}(d_1, d_2, L)}$$

$$\text{(PALink)}\quad \frac{\mathsf{bound}(\mathtt{r}\mathbf{z}_1\mathtt{m}, d) \supseteq \mathsf{bound}(\mathtt{l}\mathbf{z}_2, d)}{\Delta;\Phi \vdash_p d; (\mathsf{link}, \mathtt{r}\mathbf{z}_1\mathtt{m}, \mathtt{l}\mathbf{z}_2)} \qquad \text{(PAFuse)}\quad \frac{\mathtt{r}\mathbf{z}_1\mathtt{m} \notin \Phi \qquad \mathsf{bound}(\mathtt{r}\mathbf{z}_1\mathtt{m}, d) = \mathsf{bound}(\mathtt{l}\mathbf{z}_2, d)}{\Delta;\Phi \vdash_p d; (\mathsf{fuse}, \mathtt{r}\mathbf{z}_1\mathtt{m}, \mathtt{l}\mathbf{z}_2)}$$

Fig. 9. Rules for physical adequacy $\Delta;\Phi \vdash_p d\ [;\ l]$

Data Structures. In our present implementation, a map index can be represented by an option type (option), a singly-linked list (slist), a doubly-linked list (dlist), or a binary tree (btree). It is straightforward to extend the set of data structures by implementing a common data structure interface—we present this particular selection merely for concreteness. Each data structure must provide functions: $\mathsf{pempty}_\psi\ a$ which creates a new structure rooted at a, $\mathsf{pisempty}_\psi\ a$ which tests emptiness, $\mathsf{plookup}_\psi\ a\ \mathbf{v}$ which returns the address a' of the entry with value $\mathbf{v}$, if any, $\mathsf{pscan}_\psi\ a$ which returns the set of all $(a', \mathbf{v})$ pairs of a value $\mathbf{v}$ and its address a', $\mathsf{pinsert}_\psi\ a\ \mathbf{v}\ a'$ which inserts a new value $\mathbf{v}$ at address a', and $\mathsf{premove}_\psi\ a\ \mathbf{v}\ a'$ which removes a value $\mathbf{v}$ at address a' from a data structure.

For cross-linking and fusion to be well-defined in an index d, we need d to be physically adequate. This condition ensures that for cross-linking and fusion operations between static contours $\mathbf{z}_1$ and $\mathbf{z}_2$, the mapping from $\mathbf{z}_1$ to $\mathbf{z}_2$ is a function for each cross-link declaration and an injective function for each fusion declaration. Further, as fusions constrain the location of an object in memory, we require any object is fused at most once for feasibility. We use the judgment form $\Delta;\Phi \vdash_p d$ and the associated rules in Figure 9 to indicate that index d is physically adequate for functional dependencies Δ where Φ denotes the set of static contours that have already been fused. The (PALink) and (PAFuse) rules ensure a suitable mapping by requiring the set of fields bound by the target contour of a link be a subset of the set of fields bound by the source contour; in the case of a fusion we require equality. The rule (PAFuse) ensures that no contour is fused twice. We assume that all indices are physically adequate.

Abstraction Function We define a second abstraction function $\alpha_m(\mu, a, d) = \rho$, which given a memory state μ, root address a, and an index d constructs the corresponding tree decomposition ρ:

$\alpha_m(\mu, a, \mathsf{unit}(\mathbf{c})) = \mathsf{if}\ !a.\mathrm{ulen} = 0\ \mathsf{then}\ \{\}\ \mathsf{else}\ \{!a.\mathrm{udata}\}$
$\alpha_m(\mu, a, \mathsf{map}(\psi, \mathbf{c}, d)) = \{\mathbf{v} \mapsto \alpha_m(\mu, a', d) \mid (\mathbf{v}, a') \in \mathsf{pscan}_\psi\ a.\mathrm{map}\}$
$\alpha_m(\mu, a, \mathsf{join}(d_1, d_2, L)) = (\alpha_m(\mu, a.\mathrm{jleft}, d_1),\ \alpha_m(\mu, a.\mathrm{jright}, d_2))$

4 Queries

Up to this point we have focused on defining how relations are represented as data structures; now we turn to describing how high-level queries on relations

correspond to low-level sequences of operations traversing those data structures. Recall that we define a query operation that extracts the set of tuples in a relation whose fields match a tuple pattern, i.e., query $r\ t = r \bowtie t$, where $\operatorname{dom} t \subseteq \operatorname{dom} r$. We define *query plans* on the data structure representation, and establish sufficient conditions for a query plan to be *valid*, meaning that the query plan correctly implements a particular query on both the tree decomposition and physical representations.

One problem we do not address is selecting an efficient query plan from all possible valid query plans, but we can make a few observations. First, there is always a trivial valid query plan that uses the entire index; more efficient plans avoid traversing parts of data structures unneeded for a particular query. Second, all possible query plans can be enumerated and checked for validity; there are only so many ways to traverse an index. Finally, we expect that profile-directed database methods for selecting good query plans can be adapted to our setting; we leave that as future work.

4.1 Query Plans

A query plan is a tree of query plan operators, which take as input a *query state*, a pair $(t, \mathbf{y})$ of a tuple pattern t and a dynamic contour $\mathbf{y}$, and produce as output a set of tuples. The input tuple t maps previously bound variables to their values, whereas the dynamic contour represents the position in the index tree to which the query operator applies. Query plans are defined inductively:

None The qnone operator determines whether an index is empty or non-empty, and returns either the empty set $\{\}$ or the singleton set $\{\langle\rangle\}$ respectively.

Unit The qunit operator returns the tuple represented by a unit index, if any.

Scan The $\mathsf{qscan}(q')$ operator retrieves the list of key values that match t in a map index and invokes query operator q' for each sub-tree. Since the qscan operator iterates over the contents of a map data structure, it typically takes time linear in the number of entries in the map.

Lookup The $\mathsf{qlookup}(q')$ operator looks up a particular sequence of key values in a $\mathsf{map}(\psi, \mathbf{c}, d)$ index; each of the key columns must be bound in the input tuple t. Query operator q' is invoked on the relevant subtree, 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 Join The $\mathsf{qljoin}(q_1, q_2)$ operator first executes query q_1 in the left subtree of a join index, then executes query q_2 in the right subtree, and returns the natural join of the two results. The $\mathsf{qrjoin}(q_1, q_2)$ operator is similar, but executes the two queries in the opposite order. Both joins produce identical results, however the computational complexity may differ.

Fuse Join The $\mathsf{qfusejoin}(\mathbf{z}_0, l, q_1, q_2)$ operator switches the current index data structure by following a fuse or cross-link l and executes query q_2; it then switches back to the original location and executes q_1. The result is the natural join of the two sub-queries. Parameter $\mathbf{z}_0$ identifies the join index that contains l; position $\mathbf{y}$ must be an instantiation of the source of l.

For example, suppose in the directed graph example of Section 2.1 we want to find the set of successors of graph vertex 1, together with their edge weights. Figure 10 depicts one possible query plan q consisting of the operations

$$q = \mathsf{qrjoin}(\mathsf{qnone}, \mathsf{qscan}(\mathsf{qlookup}(\mathsf{qfusejoin}(\cdot, (\mathsf{fuse}, \mathtt{rmm}, \mathtt{lmm}), \mathsf{qunit}, \mathsf{qunit})))).$$

To find successors using query plan q, we again start with the state $(\langle src \mapsto 1\rangle, \cdot)$. Since the left branch of the join is qnone, the join reduces to a recursive execution of the query qscan$(\cdots)$ with input $(\langle src \mapsto 1\rangle, \mathtt{r})$. The qscan recursively invokes qlookup on each of the states $(\langle src \mapsto 1, dst \mapsto 2\rangle, \mathtt{rm}_2)$ and $(\langle src \mapsto 1, dst \mapsto 3\rangle, \mathtt{rm}_3)$. The qlookup operator in turn recursively invokes the qfusejoin operator on $(\langle src \mapsto 1, dst \mapsto 2\rangle, \mathtt{rm}_2\mathtt{m}_1)$ and the state $(\langle src \mapsto 1, dst \mapsto 3\rangle, \mathtt{rm}_3\mathtt{m}_1)$. To execute its second query argument the fuse join maps each instantiation of contour `rmm` to the corresponding instantiation of contour `lmm`; we are guaranteed that exactly one such contour instantiation exists by index adequacy. The fuse join produces the states $(\langle src \mapsto 1, dst \mapsto 2\rangle, \mathtt{lm}_1\mathtt{m}_2)$ and $(\langle src \mapsto 1, dst \mapsto 3\rangle, \mathtt{lm}_1\mathtt{m}_3)$. Finally the invocations of qunit on each state produces the tuples

$$\{\langle src \mapsto 1, dst \mapsto 2, weight \mapsto 17\rangle, \\ \langle src \mapsto 1, dst \mapsto 3, weight \mapsto 17\rangle\}.$$

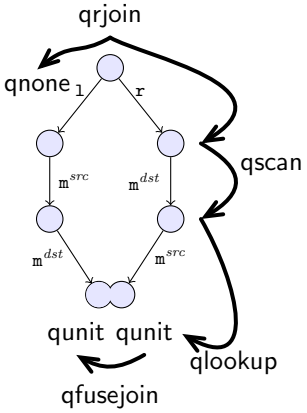

Fig. 10. One possible query plan for the graph example of Section 2.1

We need a criteria for determining whether a particular query plan does in fact return all of the tuples that match a pattern. We say a query plan is *valid*, written $d, \mathbf{z}, X \vdash_q q, Y$ if q correctly answers queries in index d at dynamic instantiations of contour $\mathbf{z}$, where X is the set of columns bound in the input tuple pattern t and Y is the set of columns bound in the output tuples (see the tech report [10]).

5 Relational Operations

In this section we describe implementations for the primitive relation operators for the tree-decomposition and physical representations of a relation, and we prove our main result: that these primitive operators are sound with respect to their higher-level specification. Complete code is given in the tech report [10].

5.1 Operators on the Tree Decomposition

We implement queries over tree decompositions by a function tquery $d\ t\ \rho$, which finds tuples matching pattern t over tree decompositions ρ under index d. The core routine is a function tqexec $\rho\ d\ q\ t\ \mathbf{y}$ which, given a tree decomposition ρ, index d, and a tuple t, executes plan q at the position of the dynamic contour $\mathbf{y}$.

Creation/update are handled by tempty d, which constructs a new empty relation with index d, tinsert $d\ t\ \rho$, which inserts a tuple t into a tree-decomposed relation ρ with index d, and tremove $d\ t\ \rho$ which removes a tuple t from a tree-decomposed relation ρ with index d. It is the client's responsibility to ensure that functional dependencies are not violated; the implementation contains dynamic checks that abort if the client fails to comply. These checks can be removed if there is an external guarantee that the client will never violate the dependencies.

To show that the primitive operations on tree decompositions faithfully implement the corresponding primitive operations on logical relations, we first show

executing valid queries over tree decompositions soundly implements logical tuple pattern queries. We then prove a soundness result by induction.

Lemma 2 (Tree Decomposition Query Soundness). *For all ρ, r, d such that $\rho \models_T d$ and $\alpha_t(\rho, d) = r$, if $d, \cdot, \operatorname{dom} t \vdash_q q, \operatorname{dom} d$ for a tuple pattern t and query plan q we have* tqexec $\rho\ d\ q\ t\ \cdot =$ query $r\ t$.

Theorem 1 (Tree Decomposition Soundness). *Suppose a sequence of* insert *and* remove *operators starting from the empty relation produce a relation r. The corresponding sequence of* tinsert *and* tremove *operators given* tempty d *as input either produce ρ such that $\rho \models_T d$ and $\alpha_t(\rho, d) = r$, or abort with an error.*

5.2 Physical Representation Operators

In this section we describe implementations of each of the primitive relation operations that operate over the physical representation of a relation. We prove soundness of the physical implementation with respect to the tree decomposition. For space reasons we omit the code for physical operators but we give a brief synopsis of each function; for a complete definition see the full paper [10].

We implement physical queries via a query execution function pqexec $d\ q\ y\ a\ \mathbf{y}$. Function pqexec is structurally very similar to the query execution function tqexec over tree decompositions. Instead of a tree decomposition ρ the physical function accesses the heap, and in place of a dynamic contour $\mathbf{y}$ the physical function represents a position in the data structure by a pair $(\mathbf{z}, a)$ of a static contour $\mathbf{z}$ and an address a. The main difference in implementation is that the qfusejoin case follows a fusion or cross-link simply by performing pointer arithmetic or a pointer dereference, respectively, rather than traversing the index.

Creation/update are handled by pempty $d\ a$ (creates an empty relation with index d rooted at address a), pinsert $d\ t\ a$ (inserts tuple t into a relation with index d rooted at a), and premove $d\ t\ a$ (removes tuple t). The main difference with corresponding operations on the tree decomposition is in pinsert, which needs to create fusions and cross-links. To fuse two nodes we simply place the data of a node being fused in a subfield of the node into which it is fused. To create a cross-link, we first construct the tree structure and then add pointers between each pair of linked nodes.

Analogous to the soundness proof for tree decompositions, we prove soundness by proving a set of commutative diagrams relating physical representations of relations with their tree decomposition counterparts. We need a well-formedness invariant for physical states. A memory state μ is *well-formed* for index d with layout Λ if there exists an injective function Λ such that the judgment $\mu; \Lambda \models_p d$ holds, defined by the inference rules in [10].

We show that valid queries over physical memory states are sound with respect to the tree decomposition. We then show soundness by induction.

Lemma 3 (Physical Query Soundness). *Suppose we have $\mu; \Lambda \models_p d$ and $\alpha_m(\mu, \Lambda(\cdot), d) = \rho$ for some μ, Λ, d. Then for all queries q and tuples t such that $d, \cdot, \operatorname{dom} t \vdash_q q, \operatorname{dom} d$ we have* pqexec $d\ q\ t\ a\ \cdot =$ tqexec $\rho\ d\ q\ t\ \cdot$, *where* pqexec *is executed in memory state μ.*

Theorem 2 (Physical Soundness). *Let d be an index, and suppose a sequence of* insert *and* tremove *operators starting from the $\rho =$ tempty d produce a relation ρ'. Let μ be the heap produced by* pempty $d\ a$ *where a is a location initially present*

in the heap. Then the corresponding sequence of pinsert *and* premove *operators given μ as input either produce a memory state μ' such that $\mu'; \Lambda \models_p d$ for some Λ and $\alpha_m(\mu, a, d) = \rho'$, or abort with an error.*

6 Related Work

Relations Many authors propose adding relations to both general- and special-purpose programming languages (e.g., [3; 15; 19; 16]). We focus on the orthogonal problem of specifying and implementing shared representations for data. Our approach can benefit from much of this past work; in particular, database techniques for query planning are likely to prove useful.

Automatic Data Structure Selection Automatic data structure selection was studied in SETL [20; 4; 17] and has also been pursued for Java collection implementations [21]. Our index language describes a mapping between abstract data and its concrete implementations with a similar goal to [7]. We focus on composing and expressing sharing between data structures which is important in many practical situations. Our work can be combined with static and dynamic techniques to infer suitable data structures.

Specifying Shared Representations Graph types [11] extend tree-structured types with extra pointers functionally determined by the structure of the tree backbone. One way to view our cross-linking and fusion constructs is adding extra pointers determined by the semantics of data and not by its structure. Separation Logic allows elegant specifications of disjoint data structures [18]. Various extensions of separation logic enable proofs about some types of sharing [2; 8].

Inferring Shared Representations Some static analysis algorithms infer some sharing between data structures in low level code [13; 12]. In contrast we allow the programmer to specify sharing in a concise way and guarantee consistency only assuming that functional dependencies are maintained. Functional dependencies or their equivalent are an essential invariant for any shared data structure.

Verification Approaches The Hob system uses abstract sets of objects to specify and verify properties that characterize how multiple data structures share objects [14]. Monotonic typestates enable aliased objects to monotonically change their typestates in the presence of sharing without violating type safety [9]. Researchers have developed systems to mechanically verify data structures (e.g., hash tables) that implement binary relational interfaces [22; 5]. The relation implementation presented here is more general, allowing relations of arbitrary arity and substantially more sophisticated data structures than previous research.

7 Conclusion

We have presented a system for specifying and operating on data structures at a high level as relations while implementing those relations as the composition of low-level pointer data structures. Most unusually we can express, and prove correct, the use of complex sharing in the low-level representation, allowing us to express many practical examples beyond the capabilities of previous techniques.

Bibliography

[1] C. Beeri, R. Fagin, and J. H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In *SIGMOD*, pages 47–61. ACM, 1977.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

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

[4] J. Cai and R. Paige. "Look ma, no hashing, and no arrays neither". In *POPL*, pages 143–154, 1991.

[5] A. J. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, pages 79–90, 2009.

[6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[7] 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, 1979.

[8] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[9] M. Fahndrich and R. Leino. Heap monotonic typestates. In *Int. Work. on Alias Confinement and Ownership*, July 2003.

[10] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data structure fusion (full), 2010. URL http://theory.stanford.edu/~hawkinsp/papers/rel-full.pdf.

[11] N. Klarlund and M. I. Schwartzbach. Graph types. In *POPL*, pages 196–205, Charleston, South Carolina, 1993. ACM.

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

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

[14] P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, pages 430–447, 2005.

[15] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *SIGMOD*, page 706. ACM, 2006.

[16] C. Olston et al. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, June 2008.

[17] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code. *J. Sym. Com.*, 4(2):207–232, 1987.

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

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

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

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

[22] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.