Verifying the Correctness of Disjoint-Set Forests with Kleene Relation Algebras

Walter Guttmann

Department of Computer Science and Software Engineering,
University of Canterbury, Christchurch, New Zealand
walter.guttmann@canterbury.ac.nz

Abstract. We give a simple relation-algebraic semantics of read and write operations on associative arrays. The array operations seamlessly integrate with assignments in computation models supporting while-programs. As a result, relation algebras can be used for verifying programs with associative arrays. We verify the correctness of an array-based implementation of disjoint-set forests with a naive union operation and a find operation with path compression. All results are formally proved in Isabelle/HOL.

1 Introduction

Relations, relation algebras, Kleene algebras and similar structures have been used for various aspects of program semantics, in particular, to model control flow, refinement and data structures [1, 18, 23, 32]. For example, the control-flow of while-programs can be modelled in Kleene algebras with tests, where the Kleene star is used to define the semantics of while-loops [2, 21]. Program transformations and refinements can be carried out algebraically; for example, see [2, 20]. On the data side, relations are intimately connected with graphs through their adjacency matrices, whence the data-flow of graph algorithms can be modelled using relation algebras, frequently extended by a Kleene star to describe transitive closure [3–6, 14, 19, 25]. Relations as a generalisation of functions are also useful for the specification and derivation of functional programs [7].

Hoare logic [16] is commonly used for verifying programs. A verification condition generator automatically derives from the structure of the program a collection of statements whose proof implies correctness of the program. When applied to graph algorithms using a relation algebra to represent graphs, the verification conditions are simply relation-algebraic formulas. They can be discharged by a combination of manual and automated reasoning in relation algebras [6].

When modelling graphs, the operations of relation algebras work on entire relations. This abstract view is useful for specification and verification, but typically not intended directly for implementation. Efficient algorithms are often expressed at a lower level, in particular, using arrays. For example, the pseudocode for disjoint-set forests in [10] uses two arrays: one for the rank of a node and one for its parent. A difference between these arrays is that the rank of a node is a natural number while the parent of a node is also a node.

An associative array is just a finite mapping from a set of indices to a set of values, hence a relation. The term 'array' often implies that the set of indices is an interval of integer numbers, but it can be an arbitrary finite set for associative arrays. The rank array of a disjoint-set forest maps nodes to natural numbers, making it a heterogeneous relation. The parent array maps nodes to nodes, which gives a homogeneous relation.

In the present paper, we focus on associative arrays with the same index and value sets. We do not assume any specific structure on the index/value set. In this context we give a simple relation-algebraic semantics of reading from and writing to an array. These access operations can occur in assignments in while-programs, and are therefore amenable to the usual verification techniques. The generated verification conditions are relation-algebraic formulas using the semantics of the array operations.

As a case study, we implement disjoint-set forests in a way that is close to the pseudo-code in [10] and verify their correctness in Isabelle/HOL. This facilitates the use of relation-algebraic reasoning about algorithms expressed at a low level.

The contributions of this paper are:

- A simple relation-algebraic semantics of selective read and write in associative arrays.
- Verification of the correctness of disjoint-set forests in Kleene relation algebras.
- Constructive proof of a theorem of Kleene relation algebras using an imperative program.
- Formalisation of the above and all other results in Isabelle/HOL.

Proofs are omitted in this paper and can be found in the Isabelle/HOL theory file available at http://www.csse.canterbury.ac.nz/walter.guttmann/algebra/.

In Section 2 we discuss related approaches. Section 3 introduces the algebraic framework for the remainder of this paper including relation algebras and Kleene algebras. We give a simple semantics of read and write access to associative arrays in Section 4 and discuss basic properties. The semantics of disjoint-set forests is provided in Section 5. Forming the main part of this paper, Section 6 describes our Isabelle/HOL verification of the total correctness of the make-set, find-set and union-sets operations on disjoint-set forests.

2 Related Work

The semantics of array or general state access is well understood and has been described in many different formalisms. We discuss a selection of these related works. An early example are the *a* and *c* functions in [24] for updating and reading state vectors, which map variables to values. Arrays are modelled as mappings in [17] and selective array updates are defined as updates of mappings. Such updates are more formally defined in [31]. A relational definition of functional overriding is given in [33] and extended to override relations in the second edition of this book. Overwriting one relation with another also appears in [26] where it is used for pointer structures. Axioms for state attributes and array access are given in [1]; some of these are used for lenses [12]. A definition of general updates in Kleene algebras with domain is given in [11]. The relation-algebraic semantics given in the present paper specialises definitions given in the last five references to selective array updates studied in the first three references.

Relation-algebraic methods have been used for the description and verification of numerous algorithms, in particular, on graphs as mentioned in the introduction. Especially relevant to the present work on disjoint sets are relational formalisations of forests and reachability; for example, see [4, 25, 32]. Also relevant are relational models of stores modelling pointer structures and using relational overwrite operations [25–28].

There are several formally verified implementations of disjoint-set forests. A persistent version of the data structure is verified in Coq by [9]. The specification is in terms of predicate logic and the implementation is based on a mathematical model of ML including references. See [22] for a verification using separation logic in Isabelle/HOL also based on a logical specification. Program complexity and correctness of an OCaml implementation is proved in [8] using separation logic in Coq based on a predicate-logic specification. See the latter paper for an overview of other formal verifications and further related works. The present paper gives a relation-algebraic specification and proof, which does not cover complexity of the union and find operations.

3 Relation Algebras and Kleene Algebras

This section presents the algebraic structures used in this development including relation algebras and Kleene algebras and basic properties [20, 32, 34].

A *semilattice* $(S, \sqcup)$ is a set S with a binary operation $\sqcup$ that is associative, commutative and idempotent. In a semilattice the binary relation $\sqsubseteq$ defined by $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$ is a partial order called the *semilattice order*. The operation $\sqcup$ is $\sqsubseteq$-isotone and gives the $\sqsubseteq$-least upper bound or join of two elements.

A *bounded semilattice* $(S, \sqcup, \bot)$ is a semilattice $(S, \sqcup)$ with a constant $\bot$ that is a unit of $\sqcup$. It follows that $\bot$ is the $\sqsubseteq$-least element of S.

A *lattice* $(S, \sqcup, \sqcap)$ comprises two semilattices $(S, \sqcup)$ and $(S, \sqcap)$ such that the absorption laws $x \sqcup (x \sqcap y) = x = x \sqcap (x \sqcup y)$ hold. The operation $\sqcap$ is $\sqsubseteq$-isotone and gives the $\sqsubseteq$-greatest lower bound or meet of two elements.

A *bounded lattice* $(S, \sqcup, \sqcap, \bot, \top)$ comprises two bounded semilattices $(S, \sqcup, \bot)$ and $(S, \sqcap, \top)$ such that $(S, \sqcup, \sqcap)$ is a lattice. It follows that $\top$ is the $\sqsubseteq$-greatest element of S and a zero of $\sqcup$, and that $\bot$ is a zero of $\sqcap$.

A lattice is *distributive* if the law $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ holds. In a lattice this law is equivalent to its dual $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.

A *Boolean algebra* $(S, \sqcup, \sqcap, \overline{}, \bot, \top)$ is a bounded lattice $(S, \sqcup, \sqcap, \bot, \top)$ that is distributive with a unary operation $\overline{}$ satisfying the laws $x \sqcup \overline{x} = \top$ and $x \sqcap \overline{x} = \bot$. The operation $\overline{}$ is $\sqsubseteq$-antitone.

A *monoid* $(S, \cdot, 1)$ is a set S with a binary composition operation $\cdot$ that is associative and a constant 1 that is a left unit and a right unit of $\cdot$.

An *idempotent semiring* $(S, \sqcup, \cdot, \bot, 1)$ is a bounded semilattice $(S, \sqcup, \bot)$ and a monoid $(S, \cdot, 1)$ such that $\cdot$ distributes over $\sqcup$ and $\bot$ is a left zero and a right zero of $\cdot$. The operation $\cdot$ is $\sqsubseteq$-isotone.

A *relation algebra* $(S, \sqcup, \sqcap, \cdot, \overline{}, ^\mathsf{T}, \bot, \top, 1)$ is a Boolean algebra $(S, \sqcup, \sqcap, \overline{}, \bot, \top)$ and an idempotent semiring $(S, \sqcup, \cdot, \bot, 1)$ with a unary transposition operation $^\mathsf{T}$ satisfying the laws:

$$(x \sqcup y)^\mathsf{T} = x^\mathsf{T} \sqcup y^\mathsf{T} \qquad\qquad x^{\mathsf{T}\mathsf{T}} = x$$
$$(x \cdot y)^\mathsf{T} = y^\mathsf{T} \cdot x^\mathsf{T} \qquad\qquad (x \cdot y) \sqcap z \sqsubseteq x \cdot (y \sqcap (x^\mathsf{T} \cdot z))$$

It follows that the operation $^\mathsf{T}$ is $\sqsubseteq$-isotone. A relation algebra satisfies the *Tarski rule* if $\top \cdot x \cdot \top = \top$ for each $x \neq \bot$.

A *Kleene algebra* $(S, \sqcup, \cdot, ^*, \bot, 1)$ is an idempotent semiring $(S, \sqcup, \cdot, \bot, 1)$ with a unary iteration operation * satisfying the laws:

$$1 \sqcup (y \cdot y^*) = y^* \qquad\qquad z \sqcup (y \cdot x) \sqsubseteq x \Rightarrow y^* \cdot z \sqsubseteq x$$
$$1 \sqcup (y^* \cdot y) = y^* \qquad\qquad z \sqcup (x \cdot y) \sqsubseteq x \Rightarrow z \cdot y^* \sqsubseteq x$$

The operation * is $\sqsubseteq$-isotone. It describes iterations with zero or more steps; the related operation $x^+ = x \cdot x^*$ describes iterations with one or more steps.

A *Kleene relation algebra* $(S, \sqcup, \sqcap, \cdot, \overline{}, ^\mathsf{T}, ^*, \bot, \top, 1)$ comprises a relation algebra $(S, \sqcup, \sqcap, \cdot, \overline{}, ^\mathsf{T}, \bot, \top, 1)$ and a Kleene algebra $(S, \sqcup, \cdot, ^*, \bot, 1)$.

An element $x \in S$ of a relation algebra S is called *reflexive* if $1 \sqsubseteq x$, *transitive* if $x \cdot x \sqsubseteq x$, *symmetric* if $x^\mathsf{T} = x$, an *equivalence* if x is reflexive and transitive and symmetric, *total* if $1 \sqsubseteq x \cdot x^\mathsf{T}$, *surjective* if $1 \sqsubseteq x^\mathsf{T} \cdot x$, *univalent* if $x^\mathsf{T} \cdot x \sqsubseteq 1$, *injective* if $x \cdot x^\mathsf{T} \sqsubseteq 1$, *bijective* if x is injective and surjective, a *mapping* if x is univalent and total, a *vector* if $x \cdot \top = x$, a *point* if x is a vector and bijective, and an *arc* if $x \cdot \top$ and $x^\mathsf{T} \cdot \top$ are bijective.

An element $x \in S$ of a Kleene relation algebra S is called *acyclic* if $x^+ \sqsubseteq \overline{1}$.

In this paper we work in a Kleene relation algebra S that satisfies the Tarski rule. For proving termination of programs we assume that S is finite.

The main model of Kleene relation algebras are binary relations over a set A, that is, subsets of $A \times A$. In this model $\sqcup$ is union, $\sqcap$ is intersection, $\overline{}$ is complement, $\sqsubseteq$ is subset, $\bot$ is the empty set, $\top$ is $A \times A$, $\cdot$ is relational composition, $^\mathsf{T}$ is relational transposition, 1 is the identity relation, * is reflexive transitive closure, $^+$ is transitive closure, and the Tarski rule holds.

We finally characterise vectors, points and arcs among the binary relations over A. A vector is a relation $B \times A$ for a subset $B \subseteq A$; hence vectors represent subsets of the base set such as a set of nodes in a graph. A point is a relation $\{a\} \times A$ for an element $a \in A$; hence points represent elements of the base set such as nodes in a graph. An arc is a relation $\{(a, b)\}$ for elements $a, b \in A$; hence arcs represent pairs of elements from the base set such as edges in a graph.

4 Associative Array Access

An array maps indices to values and therefore can be modelled as a binary relation between two sets. Under our assumption that indices and values come from the same set A, we can use binary relations on A and work with these using relation algebra. Because an array associates exactly one value to every index, the relation is a mapping in the relation-algebraic sense, that is, univalent and total. A relation that is just univalent corresponds to a partially defined array which associates at most one value to every index. An index or a value is an element of A, which can be modelled in relation algebras as a point. A relation that is just a vector corresponds to a set of indices or values.

These observations underlie the following simple semantics of array access. Let x, y and z be elements of a relation algebra such that y and z are points. The element x models the associative array, y corresponds to an index and z corresponds to a value. The array $x[y \mapsto z]$ obtained by updating array x at index y to new value z is:

$$x[y \mapsto z] = (y \sqcap z^{\mathsf{T}}) \sqcup (\overline{y} \sqcap x)$$

To understand this definition it is helpful to consider the matrix representation of the relation modelling the array x. A vector describes a set of rows of the matrix and a point describes a single row. The point y refers to the row at the corresponding index. Its complement $\overline{y}$ refers to all the other rows. The formula $\overline{y} \sqcap x$ specifies that in all other rows x is left unchanged. The formula $y \sqcap z^{\mathsf{T}}$ specifies that row y is updated to value z. Since z is a point, which refers to row, we take its transposition z^{T}, which refers to the column of the matrix at the corresponding value. In terms of binary relations, $y \sqcap z^{\mathsf{T}}$ constructs a relation containing a single pair of the index y and the value z. In relation algebras $y \sqcap z^{\mathsf{T}} = y \cdot z^{\mathsf{T}}$ is an arc for points y and z.

For example, consider the following relations x, y and z on $A = \{1, 2, 3\}$ given as Boolean matrices:

$$x = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad y = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad z = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Relation x represents a partially defined array that maps index 1 to value 3 and index 2 to value 2, point y represents index 2 and point z represents value 1. The updated array still maps index 1 to value 3, but maps index 2 to value 1:

$$y \sqcap z^{\mathsf{T}} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \overline{y} \sqcap x = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad x[y \mapsto z] = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Reading the value $x[y]$ of the associative array x at index y is done by:

$$x[y] = x^{\mathsf{T}} \cdot y$$

The composition of a relation with a vector always gives a vector. If x is interpreted as a transition relation, $x^{\mathsf{T}} \cdot y$ is a vector corresponding to the successors of the point y under a transition step according to x. In the matrix representation of an array, this is just the value of x at row y. If the array associates exactly one value to every index, the result is the unique value associated with index y, represented as a point.

Continuing the previous example, the value of x at index y is 2 and the value of x at index z is 3:

$$x^{\mathsf{T}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad x[y] = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad x[z] = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

The following result shows basic preservation properties of these write and read operations on arrays. It uses the above equational definitions without implicitly assuming that y and z are points. Part 1 is similar to [26, Lemma 2.7].

Theorem 1.
1. $x[y \mapsto z]$ is univalent if x is univalent, y is a vector and z is injective.
2. $x[y \mapsto z]$ is total if x is total, y is a vector and z is surjective.
3. $x[y \mapsto z]$ is a mapping if x is a mapping, y is a vector and z is bijective.
4. $x[y]$ is injective if x is univalent and y is injective.
5. $x[y]$ is surjective if x is total and y is surjective.
6. $x[y]$ is bijective if x is a mapping and y is bijective.
7. $x[y]$ is a point if x is a mapping and y is a point.
8. $x[y] = z \Leftrightarrow y \sqcap x = y \cdot z^{\mathsf{T}}$ if y and z are points.

5 Disjoint Sets

A disjoint-set data structure keeps track of a set of elements that is partitioned into disjoint sets [13]. The basic operations are to initialise elements to be in their own singleton sets, to form the union of two sets and to look up which set an element belongs to.

The semantics of a disjoint-set data structure with elements from A is an equivalence relation on A. The disjoint sets are just the equivalence classes of the relation. A particular representative from each class identifies a set.

An element of a relation algebra is an equivalence if it is reflexive, transitive and symmetric. The $\sqsubseteq$-least equivalence is the identity relation 1. The $\sqsubseteq$-greatest equivalence is the universal relation $\top$. Equivalences are closed under the operations $\sqcap$ and $^{\mathsf{T}}$ and, in Kleene relation algebras, under * and $^+$.

Following [10] we implement the data structure as a disjoint-set forest. Each equivalence class corresponds to a tree in the forest. Singleton sets correspond to empty trees, which contain one node. Each tree in the forest has a root and is directed. Each node in a tree has a unique parent node; the root is its own parent. The root of a tree represents the corresponding equivalence class. An edge from a node to its parent points towards the root of the tree, which can be reached by successively following parents.

Disjoint-set forests can be modelled in Kleene relation algebras as follows. An element $x \in S$ of a Kleene relation algebra S is called a *forest* if x is a mapping and $x \sqcap \overline{1}$ is acyclic. Requiring x to be a mapping ensures that each node has a unique parent. It remains to ensure that there are no cycles. We cannot require that x is acyclic because every root has itself as its parent, which corresponds to a loop in the graph. However, $x \sqcap \overline{1}$ removes all loops, so we require that the result is acyclic. Related helpful lemmas are $x^* = (x \sqcap \overline{1})^*$ and $x^* \sqcap \overline{1} = x^+ \sqcap \overline{1}$.

In a forest x, it is possible to reach from a node every other node in the same component tree by going towards its root and then back to the desired node. This defines a relation $\mathrm{fc}(x)$ on the nodes of the forest, namely the relation of being in the same component:

$$\mathrm{fc}(x) = x^* \cdot x^{\mathsf{T}*}$$

Properties of this construction are given in the following result.

Theorem 2.
1. $\mathrm{fc}(x)$ is an equivalence if x is univalent.
2. fc is $\sqsubseteq$-increasing, that is, $x \sqsubseteq \mathrm{fc}(x)$.
3. fc is $\sqsubseteq$-isotone.
4. $\mathrm{fc}(\mathrm{fc}(x)) = \mathrm{fc}(x)$ if x is univalent.
5. $\mathrm{fc}(x)^* = \mathrm{fc}(x)^+ = \mathrm{fc}(x)$ if x is univalent.
6. $\mathrm{fc}(\bot) = \mathrm{fc}(1) = 1$.
7. $\mathrm{fc}(\top) = \top$.

6 Verifying Disjoint-Set Forests in Isabelle/HOL

For implementing the operations on disjoint-set forests and verifying their correctness we use a Hoare-logic library of Isabelle/HOL [29, 30], which we have extended from partial correctness to total correctness [15]. The library supports while-programs, which have to be annotated with a precondition, a postcondition, and an invariant and a variant for each while-loop. From this, verification conditions are automatically generated.

Program variables can range over arbitrary HOL types. We write programs in the context of a class specifying the axioms of Kleene relation algebras, the Tarski rule and a finite universe for total correctness. Hence program variables range over elements from the universe of the class, which models the corresponding algebraic structure. Reasoning about these variables to discharge verification conditions is performed in the same context using existing libraries for Kleene algebras and relation algebras and newly derived theorems.

While-programs supported by the Hoare-logic library feature while-loops, conditionals, sequential composition and assignments as basic statements. We introduce new notation for array read and write operations, which are automatically translated to basic relation-algebraic expressions according to Section 4. The assignment $x[y] := z$ is translated to the assignment $x := x[y \mapsto z]$. The read expression $x[y]$ can be used directly on the right-hand side of assignments and in conditions, except we modify its syntax to $x[[y]]$ to avoid ambiguity with list syntax. This paper uses $x[y]$ except in Isabelle/HOL code which uses $x[[y]]$.

6.1 The make-set Operation

As a warm-up we implement the make-set operation of disjoint-set forests and prove its correctness. It is usually applied to each element when the data structure is initialised. Until the initialisation is complete, the underlying associative array is partial. Make-set puts an element x into its own singleton equivalence class by setting the parent of x to itself which creates an empty tree:

```
1  theorem make_set:
2    "VARS $p$
3      [ point $x \wedge p_0 = p$ ]
4      $p[x] := x$
5      [ make_set_postcondition $p\ x\ p_0$ ]"
6    apply vcg_tc_simp
7    by (simp add: ...) – names of four lemmas omitted
```

Line 2 declares variables that are changed by the program and therefore need to be part of the state, in this case only p which contains the parent array. The variables x and p_0 are universally quantified variables of the theorem; because they are not changed they do not need to be part of the state. The variable p_0 transports the initial value of p to the postcondition, where it can related to the final value of p. Line 3 gives the precondition, which requires x to be a point, representing an element of the set partitioned by the data structure. Line 4 updates the parent array to make x the root of a tree. Line 5 gives the postcondition, which is discussed below. Line 6 generates the verification condition, which for this small program is a single goal, and applies some simplifications to it:

$$\text{point } x \wedge p_0 = p \Rightarrow \text{make\_set\_postcondition } p[x \mapsto x]\ x\ p$$

Line 7 proves this goal by invoking the simplifier with additional lemmas. The postcondition has two parts:

$$\text{make\_set\_postcondition } p\ x\ p_0 \Leftrightarrow x \sqcap p = x \cdot x^{\mathsf{T}} \wedge \overline{x} \sqcap p = \overline{x} \sqcap p_0$$

The first condition $x \sqcap p = x \cdot x^{\mathsf{T}}$ states that the parent array contains x at index x. It is equivalent to $p[x] = x$ by Theorem 1.8. The second condition $\overline{x} \sqcap p = \overline{x} \sqcap p_0$ states that the parent array remains unchanged at all indices different from x.

The precondition and postcondition can be strengthened by adding $p \sqsubseteq 1$. As a consequence, when a disjoint-set forest is initialised each equivalence class constructed by make-set is a singleton.

The method vcg_tc_simp generates conditions that prove total correctness. Since the above program does not contain any while-loops, there are no conditions related to its termination.

We use a basic Hoare-logic library which does not support the definition of procedures. So that other programs can use a disjoint-set operation such as make_set, we extract an Isabelle/HOL function from the above proof using a technique of [15]. Specifically, the above total-correctness theorem implies:

```
lemma make_set_exists: "point $x \Rightarrow \exists p'$ . make_set_postcondition $p'\ x\ p$"
  using tc_extract_function make_set by blast
```

This is a consequence of how total correctness is defined on the underlying operational semantics. Hence we can introduce the following Isabelle/HOL function:

definition "make_set p x = (SOME p' . make_set_postcondition p' x p)"

The construct SOME y . $P(y)$ yields some element y that satisfies $P(y)$. In order to reason about this function in other programs we derive the following property:

```
lemma make_set_function:
  assumes "point $x$" and "$p'$ = make_set $p$ $x$"
  shows "make_set_postcondition $p'$ $x$ $p$"
  – proof omitted
```

6.2 The find-set Operation

We next implement the find-set operation of disjoint-set forests and verify its correctness. The find-set operation computes the representative of the equivalence class an element belongs to. We first demonstrate a basic implementation of find-set and then extend it by path compression. The pseudo-code in [10] uses recursion whereas we use a while-loop. The find-set operation follows the chain of parents from a node x to the root of its tree:

```
1   theorem find_set:
2     "VARS $y$
3       [ find_set_precondition $p$ $x$ ]
4       $y := x$;
5       WHILE $y \neq p[[y]]$
6         INV { find_set_invariant $p$ $x$ $y$ }
7         VAR { card $\{z \,.\, z \sqsubseteq p^{\mathsf{T}*} \cdot y\}$ }
8         DO $y := p[[y]]$
9         OD
10      [ find_set_postcondition $p$ $x$ $y$ ]"
11    apply vcg_tc_simp
12    – proof of three verification conditions omitted
```

In line 4, variable y is initialised with the start node x. The while-loop stops when it finds a node that is its own parent in line 5. Otherwise it continues with the parent of the current node in line 8.

The precondition requires that p is a forest (representing the disjoint sets) and x is a point (representing a node in the forest):

$$\text{find\_set\_precondition } p\ x \Leftrightarrow \text{forest } p \wedge \text{point } x$$

Every while-loop in the program needs to be annotated with an invariant. In this case, the invariant requires the precondition and that y is a point reachable from x along a chain of parents:

$$\text{find\_set\_invariant } p\ x\ y \Leftrightarrow \text{find\_set\_precondition } p\ x \wedge \text{point } y \wedge y \sqsubseteq p^{\mathsf{T}*} \cdot x$$

Vector $p^{\mathsf{T}*} \cdot x$ contains all successors of x under zero or more transitions of p. The postcondition states that y is a point and the root of the tree containing x:

$$\text{find\_set\_postcondition } p\ x\ y \Leftrightarrow \text{point } y \wedge y = \text{root } p\ x$$

The root of a node x in the disjoint-set forest represented by p is the unique node that has a loop and is reachable from x along a chain of parents:

$$\text{root } p\ x = (p^{\mathsf{T}*} \cdot x) \sqcap ((p \sqcap 1) \cdot \top)$$

The vector $(p \sqcap 1) \cdot \top$ contains all roots of the forest p, constructed from the relation $p \sqcap 1$ containing all loops of p. Part 1 of the following result gives an equivalent characterisation. Part 2 shows that following the parents of roots one or several times gives the roots again. We discuss part 3 below.

Theorem 3.
1. $\text{root } p\ x = (p \sqcap 1) \cdot p^{\mathsf{T}*} \cdot x$.
2. $\text{root } p\ x = p[\text{root } p\ x] = p^{\mathsf{T}*} \cdot (\text{root } p\ x)$ if p is univalent.
3. $\text{root } p\ x$ is a point if p is a forest and x is a point.

Because the above program contains one while-loop, three verification conditions are generated: one to establish the loop invariant before execution of the while-loop, one to maintain the loop invariant across execution of the body of the while-loop, and one to show the postcondition at the end of the while-loop. For partial correctness, the generated conditions are:

1. $\text{find\_set\_precondition } p\ x \Rightarrow \text{find\_set\_invariant } p\ x\ x$
2. $\text{find\_set\_invariant } p\ x\ y \wedge y \neq p[[y]] \Rightarrow \text{find\_set\_invariant } p\ x\ p[[y]]$
3. $\text{find\_set\_invariant } p\ x\ y \wedge y = p[[y]] \Rightarrow \text{find\_set\_postcondition } p\ x\ y$

To maintain the invariant we can assume that the condition of the while-loop holds. To show the postcondition we can assume that the condition of the while-loop does not hold. For total correctness, the first and third verification conditions are the same but maintenance of the invariant is modified taking into account the variant of the while-loop:

2. $\text{find\_set\_invariant } p\ x\ y \wedge y \neq p[[y]] \wedge \text{card } \{z \,.\, z \sqsubseteq p^{\mathsf{T}*} \cdot y\} = n \Rightarrow$
   $\text{find\_set\_invariant } p\ x\ p[[y]] \wedge \text{card } \{z \,.\, z \sqsubseteq p^{\mathsf{T}*} \cdot p^{\mathsf{T}} \cdot y\} < n$

Every while-loop in the program needs to be annotated with a variant. The variant is an expression that yields a natural number depending on the program variables. The value of this expression decreases after execution of the body of the loop. Because it is a natural number, it will decrease only a finite number of times which ensures termination of the while-loop. The variable n transports the initial value of the variant from the assumption to the conclusion, where it is compared with the final value of the variant.

For the above program, the variant is the number of elements in the algebra below $p^{\mathsf{T}*} \cdot y$. The expression $p^{\mathsf{T}*} \cdot y$ is a vector representing the set of nodes reachable from y by successively following parents. The variant is an order-preserving expression that turns this vector into a natural number. This works because the algebra is finite.

We now discuss Theorem 3.3, which states that the root of the tree containing point x in the forest p is a point, that is, a vector representing a single node. This result could be proved by working with the definitions of roots, points and forests. We give a different proof based on find-set. Observe that this operation computes the desired root and the postcondition states it is a point. Moreover the precondition of find-set contains just the assumptions of Theorem 3.3. Hence this result immediately follows from total correctness of find-set. In Isabelle/HOL, similary to make-set discussed in Section 6.1 we obtain:

```
lemma find_set_exists:
  "find_set_precondition $p$ $x$ $\Rightarrow \exists y$ . find_set_postcondition $p$ $x$ $y$"
  using tc_extract_function find_set by blast
```

Theorem 3.3 then is a simple consequence:

```
lemma root_point: "forest $p \wedge$ point $x \Rightarrow$ point (root $p$ $x$)"
  using find_set_exists find_set_precondition_def find_set_postcondition_def
  by simp
```

Essentially this is a constructive proof using the imperative programs supported by the Hoare-logic library. This method does not necessarily reduce the amount of work needed for proving a result but shifts the work to the correctness proof of a program. However, once the correctness proof is established it saves additional work. Moreover, this approach facilitates computational reasoning.

6.3 Path Compression

Path compression is a technique to decrease the depth of the disjoint-set forest, which makes subsequent find-set operations faster. The idea is to change the parent of every node encountered during the execution of find-set to the root of the tree. Because the root is known only after the chain of parents has been traversed, modifying the parents takes place in a separate traversal. In a recursive implementation of find-set, these modifications would take place on the way out from the recursion. We use two while-loops for the same purpose. The first loop is the find-set operation described in Section 6.2 to find the root y of the tree. As shown here, the second loop traverses the same sequence of nodes and adjusts the parents on the way:

```
1  theorem path_compression:
2    "VARS $p$ $t$ $w$
3      [ path_compression_precondition $p$ $x$ $y \wedge p_0 = p$ ]
4      $w := x$;
5      WHILE $y \neq p[[w]]$
6        INV { path_compression_invariant $p$ $x$ $y$ $p_0$ $w$ }
7        VAR { card $\{z\ .\ z \sqsubseteq p^{\mathsf{T}*} \cdot w\}$ }
```

```
8        DO
9          $t := w$;
10         $w := p[[w]]$;
11         $p[t] := y$
12       OD
13     [ path_compression_postcondition $p\ x\ y\ p_0$ ]”
14   apply vcg_tc_simp
15   – proof of three verification conditions omitted
```

This program is executed immediately after the while-loop of find-set, where p is the parent array, x is the original node and y is its representative computed by find-set, which is the root of the tree that contains x. The assignments in lines 4 and 10 traverse the same sequence of nodes as find-set. According to line 5 this finishes when the root is reached. Lines 9 and 11 set the parent of the current node to the root. Temporary variable t is used to save the current node w, which is changed by line 10.

The variant in line 7 is the same as the one used for find-set, except the current node is now stored in w. Also the generated verification conditions have the same structure as in the proof for find-set. It remains to discuss the actual precondition, invariant and postcondition. The precondition is:

$$\begin{aligned}&\text{path\_compression\_precondition } p\ x\ y \Leftrightarrow\\&\quad \text{forest } p \wedge \text{point } x \wedge \text{point } y \wedge y = \text{root } p\ x\end{aligned}$$

It extends the precondition of find-set by two conditions, which are just the post-condition of find-set. This ensures the two loops can be composed sequentially. The invariant significantly extends the precondition:

$$\begin{aligned}&\text{path\_compression\_invariant } p\ x\ y\ p_0\ w \Leftrightarrow\\&\quad \text{path\_compression\_precondition } p\ x\ y \wedge \text{fc}(p) = \text{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1\\&\quad \wedge \text{ point } w \wedge y \sqsubseteq p^{\mathsf{T}*} \cdot w \wedge (w \neq x \Rightarrow (y \neq x \wedge p[[x]] = y \wedge p^{\mathsf{T}+} \cdot w \sqsubseteq \overline{x}))\end{aligned}$$

First, $\text{fc}(p) = \text{fc}(p_0)$ states that the components of p do not change, that is, p represents the same disjoint sets. Second, $p \sqcap 1 = p_0 \sqcap 1$ states that the roots of the component trees of p do not change. Third, the invariant requires that w is a point. Fourth, $y \sqsubseteq p^{\mathsf{T}*} \cdot w$ states that the root y is reachable from w by following the chain of parents. The last part of the invariant only applies if $w \neq x$, that is, in the second or later iterations of the while-loop. In these iterations, the start node x and the root y are different, the parent of x is y, and any node reachable from w by one or more steps along the chain of parents is different from x. The postcondition is part of the invariant:

$$\begin{aligned}&\text{path\_compression\_postcondition } p\ x\ y\ p_0 \Leftrightarrow\\&\quad \text{path\_compression\_precondition } p\ x\ y \wedge \text{fc}(p) = \text{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1\end{aligned}$$

For correctness we only require that path compression does not change the disjoint sets represented by the forest. We also get that the roots do not change.

We discuss a selection of results used for maintaining the invariant. Part of the maintenance is to show that the parent relation (without loops) remains acyclic. Path compression updates the parent relation by letting the parents of visited nodes point to the root of the tree. Part 1 of the following theorem shows that updating the parent of a node w to any node y reachable from w along the chain of parents does not introduce cycles (ignoring loops).

Theorem 4.
1. $p[w \mapsto y] \sqcap \overline{1}$ is acyclic if $p \sqcap \overline{1}$ is acyclic, w and y are points and $y \sqsubseteq p^{\mathsf{T}*} \cdot w$.
2. $x \sqcap p^* = (x \sqcap 1) \sqcup ((x \sqcap p) \cdot (\overline{x} \sqcap p)^*)$ if x is a point.
3. $x \sqcap y = \bot$ if x and y are points such that $x \neq y$.

Part 2 optimises iterations similar to [25, Lemma 4]; for related techniques see also [3]. The element $x \sqcap p^*$ on the left-hand side relates the node x to all nodes reachable from it by zero or more steps in the graph p. The right-hand side contains $x \sqcap 1$, which relates x to itself, and $(x \sqcap p) \cdot (\overline{x} \sqcap p)^*$, which relates x to nodes reachable from it by one step in p followed by zero or more steps in $\overline{x} \sqcap p$. This means that edges starting in x have to be considered at most in the first step and can be omitted in the remaining steps. In maintaining the invariant, this is applied with $x = w$, so that the remaining steps only use edges not starting in w, which is important since these edges are not affected by the update to the forest.

Part 3 of the previous theorem ultimately derives from the Tarski rule and states that different points are disjoint as relations. This is a general result used in several arguments; we explain one of them. In maintaining the invariant, we need to show that updating p does not change the set of its roots. The update changes p at index w to the new value y, so this part of p changes from $w \sqcap p$ to $w \sqcap y^{\mathsf{T}}$. The roots in this part are $w \sqcap p \sqcap 1$ and $w \sqcap y^{\mathsf{T}} \sqcap 1$ and we show that both expressions are $\bot$. To this end, observe that $y \neq w$ since y is a root according to the precondition, but the parent of w is different from y according to the condition of the while-loop. First, $w \sqcap p \sqcap 1 \sqsubseteq w \sqcap 1 = \bot$ because the node w does not have a loop; otherwise $y = w$ would hold since y is reachable from w according to the loop invariant. Second, $w \sqcap y^{\mathsf{T}} \sqcap 1 = w \sqcap y \sqcap 1 \sqsubseteq w \sqcap y = \bot$ by a general property of relation algebras and part 3 of the previous theorem.

6.4 The find-set Operation with Path Compression

Using the technique of Section 6.1 we extract function definitions for the find-set operation of Section 6.2 and the path-compression operation of Section 6.3. This allows us to combine the two programs into the following one with a simple correctness proof:

```
theorem find_set_path_compression:
  "VARS p y
    [ find_set_precondition p x ∧ p_0 = p ]
    y := find_set p x;
    p := path_compression p x y
    [ path_compression_postcondition p x y p_0 ]"
  apply vcg_tc_simp
  using find_set_function find_set_postcondition_def
    find_set_precondition_def path_compression_function
    path_compression_precondition_def by fastforce
```

We can also extract a function for this program, but this function returns a pair of values as the find-set operation with path compression both modifies the disjoint-set forest and returns the root of the tree containing node x:

definition "find_set_path_compression $p\ x$ =
  (SOME (p', y) . path_compression_postcondition $p'\ x\ y\ p$)"

6.5 The union-sets Operation

We finally consider the union-sets operation, which takes two elements and joins the corresponding disjoint sets into a single set. To this end it finds the representatives of the equivalence classes of the elements and links one to the other:

```
 1 theorem union_sets:
 2   "VARS $p\ r\ s\ t$
 3     [ union_sets_precondition $p\ x\ y \wedge p_0 = p$ ]
 4     $t$ := find_set_path_compression $p\ x$;
 5     $p$ := fst $t$;
 6     $r$ := snd $t$;
 7     $t$ := find_set_path_compression $p\ y$;
 8     $p$ := fst $t$;
 9     $s$ := snd $t$;
10     $p[r] := s$
11     [ union_sets_postcondition $p\ x\ y\ p_0$ ]"
12   apply vcg_tc_simp
13   – proof of one verification condition omitted
```

Because the Hoare-logic library does not support parallel assignments, we assign the resulting pair of find-set to a temporary variable in lines 4 and 7 and separate the components in lines 5–6 and 8–9, respectively. Note how the forest p is threaded through both occurrences of find-set, where it may be modified by path compression, before line 10 adds the link from the root r of the tree containing x to the root s of the tree containing y.

The precondition of union-sets requires that p is a forest and x and y are single nodes:

$$\text{union\_sets\_precondition}\ p\ x\ y \Leftrightarrow \text{forest}\ p \wedge \text{point}\ x \wedge \text{point}\ y$$

The postcondition also requires that the final value of p represents the equivalence relation where the sets containing x and y have been merged into one:

$$\begin{aligned}&\text{union\_sets\_postcondition}\ p\ x\ y\ p_0 \Leftrightarrow\\ &\quad \text{union\_sets\_precondition}\ p\ x\ y \wedge \text{fc}(p) = \text{wcc}(p_0 \sqcup (x \cdot y^{\mathsf{T}}))\end{aligned}$$

To get the latter equivalence relation, we add the pair (x, y) to the initial equivalence relation p_0 and compute its equivalence closure, that is, the smallest equivalence relation containing p_0 and the pair (x, y). The pair (x, y) is described by $x \cdot y^\mathsf{T}$ and according to [32] the equivalence closure is given by:

$$\mathrm{wcc}(x) = (x \sqcup x^\mathsf{T})^*$$

Interpreting the relation x as a directed graph, the equivalence closure represents the weakly-connected components of x, which are obtained by reachability while ignoring the direction of edges. Properties of wcc are given in the following result.

Theorem 5.
1. $\mathrm{wcc}(x)$ is an equivalence.
2. wcc is a closure operation, that is, idempotent, $\sqsubseteq$-isotone and $\sqsubseteq$-increasing.
3. $\mathrm{wcc}(x) \sqsubseteq \mathrm{wcc}(y)$ if $x \sqsubseteq \mathrm{wcc}(y)$.
4. $\mathrm{wcc}(\bot) = \mathrm{wcc}(1) = 1$.
5. $\mathrm{wcc}(\top) = \top$.
6. $\mathrm{wcc}(x \sqcup 1) = \mathrm{wcc}(x \sqcap \overline{1}) = \mathrm{wcc}(x)$.
7. $\mathrm{wcc}(x) = \mathrm{fc}(x)$ if x is univalent.

We further discuss a selection of results used for proving the correctness of union-sets. Part 1 of the following result is similar to [6, Proposition 3]. It considers reachability under the union of two relations x and y, where x is an arc containing just one edge. It then suffices to use the edge x at most once: y^+ describes the case where x is not needed and $y^* \cdot x \cdot y^*$ describes the case where x is used once, preceded and followed by any number of edges in y.

Theorem 6.
1. $(x \sqcup y)^+ = y^+ \sqcup (y^* \cdot x \cdot y^*)$ if x is an arc.
2. $p[w \mapsto y] \sqcap \overline{1}$ is acyclic if $p \sqcap \overline{1}$ is acyclic, w and y are points, and $y \sqcap p^* \cdot w = \bot$.
3. $p[w \mapsto w] \sqcap \overline{1}$ is acyclic if $p \sqcap \overline{1}$ is acyclic and w is a point.

Parts 2 and 3 are similar to Theorem 4.1. In part 2 the parent of w is updated to a node y from which w is not reachable in p. This does not introduce a cycle (ignoring loops). Part 3 shows that creating a loop on w does not introduce a cycle (ignoring loops).

These results are used to show that the assignment in line 10 of union-sets maintains the forest property. If the arguments x and y of union-sets are in the same tree, the roots r and s will be equal, so line 10 creates a loop and the correctness proof uses part 3 of the preceding result. Alternatively, it could be proved that the assignment in line 10 does not change the forest in this case. Part 2 of the preceding result is used if nodes x and y are in different trees.

7 Conclusion

This paper has given a simple relation-algebraic semantics for read and write operations on associative arrays. Based on this semantics, we added such operations to a sequential programming language used for specifying and verifying programs in Isabelle/HOL. We implemented disjoint-set forests with path compression this way and proved their correctness.

Correctness of the union-sets operation would not be affected if the assignment in line 10 of the program in Section 6.5 was replaced with $p[s] := r$. More efficient implementations of union-sets therefore decide which of these two assignments to use based on heuristics such as union by rank. The rank of a node is a natural number giving an upper bound on the depth of the subtree at the node. It is more efficient to add a link from the root with smaller rank to the other. Using ranks in a disjoint-set forest implementation requires comparisons and simple arithmetic operations. In future work we will consider how to implement this extension using relation-algebraic methods.

Another task is to integrate the implementation given in this paper with relation-algebraic implementations of Kruskal's minimum spanning tree algorithm. For this reason our Isabelle/HOL theory uses Stone-Kleene relation algebras, which are weaker than Kleene relation algebras and can represent weighted graphs [15]. A further direction of research is to consider how relation-algebraic methods can support complexity analysis of algorithms.

Acknowledgement. I thank the anonymous referees for their helpful comments.

References

1. Back, R.J., von Wright, J.: Refinement Calculus. Springer, New York (1998)
2. Back, R.J.R., von Wright, J.: Reasoning algebraically about loops. Acta Inf. **36**(4), 295–334 (1999)
3. Backhouse, R.C., Carré, B.A.: Regular algebra applied to path-finding problems. Journal of the Institute of Mathematics and its Applications **15**(2), 161–186 (1975)
4. Berghammer, R.: Combining relational calculus and the Dijkstra–Gries method for deriving relational programs. Information Sciences **119**(3–4), 155–171 (1999)
5. Berghammer, R., von Karger, B., Wolf, A.: Relation-algebraic derivation of spanning tree algorithms. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 23–43. Springer (1998)
6. Berghammer, R., Struth, G.: On automated program construction and verification. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 22–41. Springer (2010)
7. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1997)
8. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning **62**(3), 331–365 (2019)
9. Conchon, S., Filliâtre, J.C.: A persistent union-find data structure. In: Dreyer, D., Russo, C. (eds.) ML 2007. pp. 37–45. ACM (2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
11. Ehm, T.: Pointer Kleene algebra. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS/KA. LNCS, vol. 3051, pp. 99–111. Springer (2004)

12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Progr. Lang. Syst. **29**(3:17), 1–65 (2007)
13. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM **7**(5), 301–303 (1964)
14. Gondran, M., Minoux, M.: Graphs, Dioids and Semirings. Springer (2008)
15. Guttmann, W.: Verifying minimum spanning tree algorithms with Stone relation algebras. Journal of Logical and Algebraic Methods in Programming **101**, 132–150 (2018)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580/583 (1969)
17. Hoare, C.A.R.: Notes on data structuring. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.) Structured Programming, chap. 2, pp. 83–174. Academic Press (1972)
18. Hoare, C.A.R., He, J.: Unifying theories of programming. Prentice Hall Europe (1998)
19. Höfner, P., Möller, B.: Dijkstra, Floyd and Warshall meet Kleene. Formal Aspects of Computing **24**(4), 459–476 (2012)
20. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation **110**(2), 366–390 (1994)
21. Kozen, D.: Kleene algebra with tests. ACM Trans. Progr. Lang. Syst. **19**(3), 427–443 (1997)
22. Lammich, P., Meis, R.: A separation logic framework for Imperative HOL. Archive of Formal Proofs (2012)
23. Maddux, R.D.: Relation-algebraic semantics. Theor. Comput. Sci. **160**(1–2), 1–85 (1996)
24. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) IFIP 1962. IFIP congress series, vol. 2, pp. 21–28. North-Holland Publishing Company (1963)
25. Möller, B.: Derivation of graph and pointer algorithms. In: Möller, B., Partsch, H.A., Schuman, S.A. (eds.) Formal Program Development. LNCS, vol. 755, pp. 123–160. Springer (1993)
26. Möller, B.: Towards pointer algebra. Sci. Comput. Program. **21**(1), 57–90 (1993)
27. Möller, B.: Calculating with pointer structures. In: Bird, R., Meertens, L.G.L.T. (eds.) Algorithmic Languages and Calculi 1997. IFIP Conference Proceedings, vol. 95, pp. 24–48. Chapman and Hall (1997)
28. Möller, B.: Calculating with acyclic and cyclic lists. Information Sciences **119**(3–4), 135–154 (1999)
29. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing **10**(2), 171–186 (1998)
30. Nipkow, T.: Hoare logics in Isabelle/HOL. In: Schwichtenberg, H., Steinbrüggen, R. (eds.) Proof and System-Reliability. pp. 341–367. Kluwer Academic Publishers (2002)
31. Reynolds, J.C.: Reasoning about arrays. Commun. ACM **22**(5), 290–299 (1979)
32. Schmidt, G., Ströhlein, T.: Relationen und Graphen. Springer (1989)
33. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall (1989)
34. Tarski, A.: On the calculus of relations. The Journal of Symbolic Logic **6**(3), 73–89 (1941)