arXiv:2301.10311v4 [cs.LO] 16 Jul 2024

Relation-Algebraic Verification of Disjoint-Set Forests

Walter Guttmann[*]

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

Abstract. This paper studies how to use relation algebras, which are useful for high-level specification and verification, for proving the correctness of lower-level array-based implementations of algorithms. 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 using the union-by-rank strategy and find operations with path compression, path splitting and path halving. All results are formally proved in Isabelle/HOL. This paper is an extended version of [1].

Keywords: arrays, Kleene algebras, path compression, program verification, relation algebras, union-find

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 [2–5]. 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 [6, 7]. Program transformations and refinements can be carried out algebraically; for example, see [6, 8]. 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 or reachability in graphs [9–15]. Relations as a generalisation of functions are also useful for the specification and derivation of functional programs [16].

This paper extends the relation-algebraic method by verifying algorithms which use associative arrays as data. Hoare logic [17] is commonly used for verifying programs. A verification condition generator automatically derives from the structure of the program a collection of logical 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 [12].

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. In contrast, efficient algorithms are often expressed at a lower level, in particular, using arrays. For example, the pseudo-code for disjoint-set forests in [18] uses two arrays: one for the rank of a node and one for its parent. A difference between these arrays is their type: the rank of a node is a natural number while the parent of a node is also a node.

This paper addresses the gap between the benefits of relation algebras for high-level specification and verification on the one hand, and the need for lower-level formalisms to express efficient algorithms on the other hand. Specifically, we show how relation algebras, Kleene algebras and related structures can be used for proving the correctness of array-based implementations of algorithms.

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, but this is not a fundamental limitation. Without assuming any specific structure on the index/value set, 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 [18] and verify their correctness in Isabelle/HOL [19]. This facilitates the use of relation-algebraic reasoning about algorithms expressed at a low level. In this paper we do not go below the pseudo-code level of [18], for example, to a concrete programming language. That said, associative arrays are commonly used to implement disjoint-set forests, not just in pseudo-code. For example, the JGraphT library for Java (version 1.5.1, see https://jgrapht.org/) uses finite maps, and the Boost library for C++ (version 1.79.0, see https://www.boost.org/) uses finite maps from elements to integer indices and vectors (integer-indexed arrays whose size can change) for the actual data structure. We formalise specifications and implementations and carry out the verification using relation algebras, Kleene alge-

bras and related structures rather than Isabelle/HOL's finite mappings and relations in order to show that algebraic methods can be applied for lower-level reasoning.

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. This includes path compression, path splitting, path halving and union-by-rank. Additional structure is assumed to deal with ranks, which are natural numbers.

- Constructive proof of a theorem of Kleene relation algebras (Theorem 5.3) using an imperative program.

- Formalisation of the above and all other results in Isabelle/HOL.

Proofs of all results can be found in the Isabelle/HOL theory files available in the Archive of Formal Proofs [20]. The PDF version of this paper includes links to the respective theorems; click on occurrences of 🐝 to follow these links. Appendix B contains proofs for a sample of theorems.

In Section 2 we motivate the relation-algebraic approach. 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, Sections 6–8 describe our Isabelle/HOL verification of the total correctness of key operations on disjoint-set forests. Section 6 deals with initialising disjoint-set forests by applying the make-set operation to all elements. Section 7 covers the find-set operation and various path-compression techniques such as path splitting and path halving. Section 8 is concerned with the union-sets operation, in particular, using the union-by-rank strategy which involves working with natural numbers in a relational framework. In Section 9 we discuss related work and limitations.

In addition to the main results, which prove the correctness of various algorithms for disjoint-set forests, throughout the paper we include a number of theorems capturing properties of associative array access, directed forests, reachability in graphs, graph components and Peano structures. Some of these facts are used in the correctness proofs while others arise from studying the involved concepts. They all build on and become part of a formally verified library of (currently several thousand) results related to Kleene algebras, relation algebras and similar algebraic structures, which is part of the Archive of Formal Proofs. These libraries have been developed over many years in previous work and are structured into a hierarchy of algebras; each result is usually proved in the most general structure to avoid unnecessary assumptions and make it more widely applicable. Such theory-building not only helps to understand concepts but also enables effective automated theorem proving support during a development and in future developments. It is often not clear in advance which results might turn out to be useful later; automated tools such as Sledgehammer [21] can filter the large collection and identify relevant results if and when they are needed. Hence in this paper we present not just results that are actually used in the overall correctness proof; in fact it would require tool support to gather this information and the result might not give much insight as automatically constructed proofs often differ from manually constructed ones.

The present paper is a revised and extended version of [1]. Section 6 of the conference version has been split into current Sections 6–8 to accomodate new material. The extensions comprise new Sections 2, 6.2, 7.4, 7.5, 8.2–8.4 and 9.2 and Appendix B, as well as additions to Sections 4, 7.2, 7.3 and 8.1 and Appendix A. They cover additional properties of read and write operations on arrays, the initialisation of disjoint-set forests, a number of abstractions, the precise effect of path compression, path splitting, path halving, results on relational Peano structures and union-by-rank. All new results have been formally proved in Isabelle/HOL requiring a substantial extension of the theories developed for [1].

2.   Motivation

In this section we motivate the relation-algebraic approach to program correctness. We argue that relation algebras facilitate program development and verification by supporting concise high-level specifications, equational reasoning and generalisation.

In the introduction we have summarised how relations and algebras have been used to model programs and their development [2–16]. The underlying idea is to use algebraic equations and inequalities instead of logical formulas. For example, recall that relation R is transitive if

$$\forall x, y, z : (x, y) \in R \land (y, z) \in R \Rightarrow (x, z) \in R$$

In relation algebras this is expressed as $R \cdot R \subseteq R$ using relational composition. For another example, directed graph R is acyclic if

$$\neg \exists n : \exists x_0, \ldots, x_n : (x_n, x_0) \in R \land \forall i : 0 \leq i < n \Rightarrow (x_i, x_{i+1}) \in R$$

In relation algebras this is expressed as $R^+ \subseteq \overline{1}$ stating that the transitive closure of R is included in the complement of the identity relation. These examples demonstrate how eliminating quantifiers can lead to concise specifications. The resulting algebraic expressions allow equational proofs instead of pointwise arguments. This is similar to how higher-order functions and pointfree properties raise the level of abstraction in functional programming: for example, the map fusion law can be expressed as $\mathrm{map}\, f \circ \mathrm{map}\, g = \mathrm{map}\, (f \circ g)$ rather than the pointwise $\mathrm{map}\, f\, (\mathrm{map}\, g\, xs) = \mathrm{map}\, (f \circ g)\, xs$.

To further motivate the relational approach, we discuss two examples from algorithm development. First, the maximum segment sum problem computes the maximum sum of a contiguous sublist of a given list of numbers. A relational specification is given by $\Lambda(suffix \cdot prefix \cdot sum) \cdot max$ [22]. Here, *prefix* and *suffix* are the prefix and suffix relations on lists, respectively; *sum* computes the sum of a list of numbers; the power transpose Λ converts a relation into a set-valued mapping; *max* picks the maximum element from a set of numbers. Thus the specification generates all contiguous sublists (prefixes of suffixes of a list), computes the sum of each, and takes the maximum of the resulting numbers. This concise specification is easily seen to be valid, but a straight-forward implementation is not optimal. In [22], an optimal linear-time, constant-space algorithm is derived from this specification using equational reasoning. Correctness of the efficient algorithm follows from the method of construction by semantics-preserving program derivation. This is based on algebraic properties of the relations involved in the specification. Thus the specification avoids quantification over list indices and

facilitates the application of general optimisation strategies, and the correctness proof avoids low-level reasoning about list indices.

Second, the minimum spanning tree problem computes a tree with minimum total weight that spans a given weighted graph, where to span requires the tree to be an inclusion-maximal acyclic subgraph. This problem and a number of efficient algorithms for it are well known [18]. A relation-algebraic specification and implementation have been given in [23], and correctness of the implementation has been formally verified in Isabelle/HOL. Because correctness is entirely proved in extensions of relation algebras, different instances of these algebras can be constructed to solve different problems. Hence the same correctness proof covers the minimum-weight spanning tree problem, the minimum bottleneck spanning tree problem and further optimisation problems; see [23] for details. This is similar to how Warshall's transitive closure algorithm and Floyd's all-pairs shortest paths algorithm have been generalised to semirings, which cover many other instances. In both cases, the minimum spanning tree and the shortest path, the generalisation relies on using algebras to model the problem, implement the solution and perform its correctness proof.

The aim of this paper is to show that the relation-algebraic approach to program correctness can be extended by associative arrays, thereby providing access to efficient implementations when needed while maintaining the above benefits of high-level specification, equational reasoning and generalisation due to algebras.

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 [5, 8, 24].

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

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

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

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 a closure operation: it is idempotent, $\sqsubseteq$-isotone and $\sqsubseteq$-increasing, that is, $x \sqsubseteq x^*$. It describes finite iterations with zero or more steps; the related operation $x^+ = x \cdot x^*$ describes finite iterations with one or more steps. In particular, $y^* \cdot z$ and $z \cdot y^*$ are the $\sqsubseteq$-least fixpoints of the functions $\lambda x.(y \cdot x) \sqcup z$ and $\lambda x.(x \cdot y) \sqcup z$, respectively. Moreover $(x \sqcup y)^* = x^* \cdot (y \cdot x^*)^*$ and $x \cdot (y \cdot x)^* = (x \cdot y)^* \cdot x$.

A *Kleene relation algebra* $(S, \sqcup, \sqcap, \cdot, ^-, ^\mathsf{T}, ^*, \bot, \top, 1)$ comprises a Kleene algebra $(S, \sqcup, \cdot, ^*, \bot, 1)$ and a relation algebra $(S, \sqcup, \sqcap, \cdot, ^-, ^\mathsf{T}, \bot, \top, 1)$. It follows that $x^{*\mathsf{T}} = x^{\mathsf{T}*}$.

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. In some cases it would suffice to require that S is complete at the cost of strengthening acyclic to progressively finite [5]; we do not pursue this because the algorithms are usually applied to finite disjoint-set forests.

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, $^-$ 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. Vectors correspond to row-constant matrices and are also called 'conditions' in the literature. 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. In particular, a point is injective, surjective and a vector, but not univalent (if A has more than two elements). 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. We use such relations to model arrays which have not yet been fully initialised. 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 a 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:

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

In the remainder of this paper, we use the above equational definitions of array write and read without implicitly assuming that y and z are points. This allows updating the values of several indices at once and other general ways of writing and reading which can be useful for specification and reasoning.

The following result shows basic preservation properties of write and read operations on arrays. Part 1 is similar to [25, 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.

Proof: See Appendix B for a proof of Theorem 1.8 and [20] for all proofs.

The following result shows further properties of write and read operations on arrays, which are helpful for verifying the correctness of operations on disjoint-set forests. Parts 1–5 are the put-get, put-put and get-put lens properties [26] and the array independence requirements of [2]. Parts 6–7 combine and split updates with the same value. Parts 8–10 swap the values $x[y]$ and $x[z]$.

Theorem 2.

1. $x[y \mapsto z][y] = z$ if y is a surjective vector and z is a vector. 🔷

2. $x[y \mapsto z][u] = x[u]$ if $u \sqsubseteq \overline{y}$ and y is a vector. 🔷

3. $x[y \mapsto z][y \mapsto u] = x[y \mapsto u]$. 🔷

4. $x[y \mapsto z][u \mapsto v] = x[u \mapsto v][y \mapsto z]$ if u and y are vectors with $u \sqcap y = \bot$. 🔷

5. $x[y \mapsto x[y]] = x$ if y is a point. 🔷

6. $x[y \mapsto z][u \mapsto z] = x[y \sqcup u \mapsto z]$. 🔷

7. $x[y \mapsto z] = x[y - w \mapsto z][y \sqcap w \mapsto z]$. 🔷

8. $x[y \mapsto x[z]][z \mapsto x[y]]$ is injective if x is injective, y is a point and z is an injective vector. 🔷

9. $x[y \mapsto x[z]][z \mapsto x[y]]$ is univalent if x is univalent and y and z are injective vectors. 🔷

10. $x[y \mapsto x[z]][z \mapsto x[y]]$ is a mapping if x is a mapping and y and z are points. 🔷

11. $x[\bot \mapsto z] = x$. 🔷

12. $x[\top \mapsto z] = z^{\mathsf{T}}$. 🔷

13. $x[y \mapsto z] \sqsubseteq x \sqcup z^{\mathsf{T}}$. 🔷

14. $x[y \mapsto z^{\mathsf{T}}] = x$ if $y \sqsubseteq u$ and $u \sqcap x = u \sqcap z$ for some u. 🔷

15. $x[y \mapsto z[y]] = x[y \mapsto z^{\mathsf{T}}]$ if y is a point. 🔷

16. $x \sqcap y$ is an arc if x is a mapping and y is a point. 🔷

Proof: See Appendix B for a proof of Theorem 2.8 and [20] for all proofs.

We conclude this section by discussing the representation of array indices using points. Restrictions with vectors, such as $y \sqcap z^{\mathsf{T}}$ and $\overline{y} \sqcap x$ in an array update, can also be obtained by composing with partial identities, as in $(y \sqcap 1) \cdot z^{\mathsf{T}}$ and $(\overline{y} \sqcap 1) \cdot x$. In relation algebras, partial identities and vectors provide equivalent ways of representing sets which are readily converted to each other. Yet another way to represent sets is to use the transposition of a vector. Each of these representations has advantages in some situations and disadvantages in others. For example, using the transposition of a point for array indices would simplify array read $x[y]$ to a simple composition, but would complicate array update; representing sets by partial identities would make restrictions from the left and from the right more symmetric and would allow some results such as Theorem 10.2 in Appendix A to be generalised to Kleene algebras, but would require an extra domain operation to identify certain sets of nodes in a graph. It is not the purpose of this paper to study advantages and disadvantages of the various representations; any of them would be fine as regards the contributions of this paper.

5. Disjoint sets

A disjoint-set data structure keeps track of a set of elements that is partitioned into disjoint sets [27]. 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 $^\top$ and, in Kleene relation algebras, under * and $^+$.

The *equivalence closure* of a relation x is the smallest equivalence relation containing x. According to [5] it is given by:

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

Interpreting the relation x as a directed graph, the equivalence closure represents the weakly-connected components of x (hence the short name wcc), which are obtained by reachability while ignoring the direction of edges. Properties of wcc are given in the following result. Part 3 shows a Galois connection between wcc and the forgetful functor mapping equivalences to relations, which is a consequence of being a closure operation.

Theorem 3.

1. $\mathrm{wcc}(x)$ is an equivalence. 🐝

2. wcc is a closure operation. 🐝

3. $\mathrm{wcc}(x) \sqsubseteq \mathrm{wcc}(y)$ if and only if $x \sqsubseteq \mathrm{wcc}(y)$. 🐝

4. $\mathrm{wcc}(\bot) = \mathrm{wcc}(1) = 1$. 🐝

5. $\mathrm{wcc}(\top) = \top$. 🐝

6. $\mathrm{wcc}(x \sqcup 1) = \mathrm{wcc}(x - 1) = \mathrm{wcc}(x)$. 🐝

7. $\mathrm{wcc}(x \sqcup y) = \mathrm{wcc}(x \sqcup \mathrm{wcc}(y))$. 🐝

Proof: See Appendix B for a proof of Theorem 3.2 and [20] for all proofs.

Following [18] we implement the data structure as a disjoint-set forest. These are directed forests in the graph-theoretical sense except the root of each tree has a self-loop, that is, an edge to itself. Each equivalence class corresponds to a tree in the forest. Singleton sets correspond to singleton trees, which contain one node with a self-loop. 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 $x - 1$ 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 self-loop in the graph. However, $x - 1$ removes all self-loops, so we require that the result is acyclic. Related helpful lemmas are $x^* = (x - 1)^*$ and $x^* - 1 = x^+ - 1$.

In a forest x, it is possible to reach from a node every 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 (fc is short for forest-components):

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

Properties of this construction are given in the following result. Part 8 shows that fc and wcc coincide for forests.

Theorem 4.

1. $\mathrm{fc}(x)$ is an equivalence if x is univalent. 🔧

2. fc is $\sqsubseteq$-increasing. 🔧

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. $\mathrm{fc}(\top) = \top$. 🔧

8. $\mathrm{fc}(x) = \mathrm{wcc}(x)$ if x is univalent. 🔧

Proof: See Appendix B for a proof of Theorem 4.1 and [20] for all proofs.

Functions wcc and fc can be used to provide abstract specifications for operations on disjoint-set forests. The abstraction function fc maps a forest to the equivalence relation it represents. Operations such as path compression change the forest but not the represented equivalence relation, which is expressed by keeping the value under fc constant. Operations such as union-sets change the equivalence relation, which is expressed by computing the new equivalence using wcc. The specifications will be given implicitly by the pre- and postconditions of the verified implementations in the following sections. Alternative methods to construct formally correct programs start with a specification and proceed by program transformations and data refinement; for example, see [28–30]. This paper follows the program verification method, which proves correctness of a given implementation with respect to a given specification.

6.   Verifying disjoint-set forests – initialisation

For implementing the operations on disjoint-set forests and verifying their correctness we use a Hoare-logic library of Isabelle/HOL [31, 32], which we have extended from partial correctness to total correctness [33]. 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. Soundness of the Hoare-logic rules is verified with respect to a standard operational semantics of the while-programs.

Program variables can range over arbitrary HOL types. We write programs in the context of an Isabelle/HOL 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. The values of the program variables can be interpreted as binary relations and represented by Boolean matrices, like the examples in Section 4, but other interpretations are possible as long as they satisfy the given axioms. The Boolean matrix model has been formalised in Isabelle/HOL and proved to satisfy the axioms to ensure their soundness.

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 in Isabelle/HOL. This paper uses $x[y]$ to improve readability. In the Boolean matrix model of relations, arrays representing forests are in one-to-one correspondence with finite maps.

This section covers verification of the initialisation phase of disjoint-set forests. The find and union operations are covered in Sections 7 and 8, respectively.

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 a singleton tree:

```
1  theorem make_set: 🧩
2     "VARS p
3        [ point x ∧ p₀ = p ]
4        p[x] := x
5        [ make_set_postcondition p x p₀ ]"
6     apply vcg_tc_simp
7     by (simp add: ...) – names of four lemmas omitted
```

Lines 2–5 combine an implementation (line 4) with a specification comprising a frame (line 2), a precondition (line 3) and a postcondition (line 5) as in [34]. The specification can also be understood

as a Hoare triple extended to include the frame whose scope is the entire triple. The theorem states that the implementation is totally correct with respect to the specification.

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 be 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. Both conditions together are equivalent to $p = p_0[x \mapsto x]$.

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 [33]. 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

Ideally, the above technique or another construction that provides an Isabelle/HOL function and a corresponding pre-/postcondition result should be part of the Hoare-logic library.

6.2.   Initialising disjoint-set forests

A disjoint-set forest is initialised by applying the make-set operation to each element. In [18] this is achieved by a simple for-each-loop. Since the Hoare-logic library only supports while-loops we first introduce a suitable abstraction.

Iteration through the elements is implemented using a working set of elements that have not been considered yet from which an element is selected and removed in each step. Sets of elements are represented as vectors. The selected element is represented as a point and obtained by the operation choose_point with the following axioms [10]:

1. vector $x \wedge x \neq \bot \Rightarrow$ point (choose_point x)

2. choose_point $x \sqsubseteq x$

The first axiom specifies that the result of choose_point x is one element if x is a non-empty set of elements. The second axiom specifies that the resulting element is contained in x. Using this operation we define the control structure

     FOREACH x
        USING h
        INV $\{\, i \,\}$
        DO c OD

as syntactic sugar for

```
1   h := ⊤;
2   WHILE h ≠ ⊥
3       INV { vector h ∧ i }
4       VAR { h↓ }
5       DO
6           x := choose_point h;
7           c;
8           h[x] := ⊥
9       OD
```

Vector h describes the set of all elements that are yet to be processed, which contains all elements at the start according to line 1. The for-each-loop abstraction makes this vector explicit so the invariant can refer to it. Point x is the element selected in each iteration in line 6. In line 8, vector h is interpreted as an array whose value at index x is set to $\bot$ signifying the removal of x from h. As per line 2, the while-loop terminates when all elements have been processed.

Every while-loop in a program needs to be annotated with an invariant and a variant. Line 3 augments the invariant of the for-each-loop by the fact that h is a vector. We also show that the while-loop terminates using the variant in line 4. The variant is an expression that yields a natural number depending on the program variables. For the above program, the variant is $h\downarrow$, the number of elements in the algebra below h. 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. This works because the algebra is finite.

Using a for-each-loop the overall initialisation of a disjoint set forest works as follows:

```
1   theorem init_sets: 🎲
2      "VARS h p x
3        [ True ]
4        FOREACH x
5          USING h
6          INV { p − h = 1 − h }
7          DO
8              p := make_set p x
9          OD
10        [ p = 1 ∧ forest p ∧ h = ⊥ ]"
11     apply vcg_tc_simp
12      – proof of one verification condition omitted
```

The loop iterates over each element x according to line 4. Line 8 initialises the disjoint-set forest at that element using the make_set operation extracted in Section 6.1.

Line 5 declares h to refer to the set of elements yet to be initialised. The loop invariant in line 6 maintains that all elements in $\overline{h}$ have already been initialised, that is, the parent of such an element is itself. According to the postcondition in line 10, after the initialisation the parent array is the identity relation, that is, all elements are roots in the disjoint-set forest. Line 3 shows that this requires no precondition.

Because the above program contains one loop, three verification conditions are generated: one to establish the loop invariant before execution of the loop, one to maintain the loop invariant across execution of the body of the loop, and one to show the postcondition at the end of the loop. The built-in simplifier automatically discharges the first and the last of these conditions for init_sets, and it only remains to show that the invariant is maintained.

An Isabelle/HOL function could be extracted from the correctness of init_sets as we have done for make_set in Section 6.1. We omit this as the effect of init_sets can be achieved simply by setting $p := 1$. The difference is that this assignment initialises the entire array p at once whereas the loop in init_sets is closer to how initialisation would be implemented in lower-level algorithms. For example, the algorithm in [18] calls make-set for each node in the graph without specifying an order, just like the for-each-loop abstraction does not specify the order of iteration.

7. Verifying disjoint-set forests – find and path-compression

In this section we implement the find-set operation of disjoint-set forests and various path-compression techniques and verify correctness of these algorithms. Section 7.1 covers the basic find-set operation without modifying parent links. A separate path-compression phase is the topic of Section 7.2, which also describes the precise effect of this phase. Both phases are combined and their precise effect is

studied in Section 7.3. Two alternative path-compression techniques, namely path splitting and path halving, are discussed in Sections 7.4 and 7.5, respectively.

7.1. The find-set operation

We start by implementing the find-set operation of disjoint-set forests and verifying 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 [18] 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 ≠ p[y]
 6          INV { find_set_invariant p x y }
 7          VAR { p*[y]↓ }
 8          DO
 9             y := p[y]
10          OD
11        [ find_set_postcondition p x y ]"
12      apply vcg_tc_simp
13        – 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 9.

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

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

The loop invariant requires the precondition and that y is a point reachable from x along a chain of parents:

$$\text{find\_set\_invariant } p\ x\ y \Leftrightarrow \text{find\_set\_precondition } p\ x \wedge \text{point } y \wedge y \sqsubseteq p^*[x]$$

Vector $p^*[x] = p^{\mathsf{T}*} \cdot x$ describes the ancestors of x under p using array-read notation, that is, 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:

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

A vector of all roots of the disjoint-set forest represented by p is constructed from the relation $p \sqcap 1$ containing all self-loops of p:

$$\text{roots } p = (p \sqcap 1) \cdot \top$$

The root of a node x in forest p is the unique root that is an ancestor of x:

$$\text{root } p \; x = p^*[x] \sqcap \text{roots } p$$

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 a constructive proof of part 3 at the end of Section 7.1.

Theorem 5.

1. $\text{root } p \; x = (p \sqcap 1) \cdot p^*[x]$.

2. $\text{root } p \; x = p[\text{root } p \; x] = p^*[\text{root } p \; x]$ and $\text{roots } p = p[\text{roots } p] = p^*[\text{roots } p]$ if p is univalent.

3. $\text{root } p \; x$ is a point if p is a forest and x is a point.

   For partial correctness, the three verification conditions generated for find-set 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. $\text{find\_set\_invariant } p \; x \; y \wedge y \neq p[y] \wedge p^*[y]{\downarrow} = n \Rightarrow \text{find\_set\_invariant } p \; x \; p[y] \wedge p^*[p[y]]{\downarrow} < n$

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 $p^*[y]{\downarrow}$, the number of elements in the algebra below $p^*[y]$. The expression $p^*[y]$ is a vector representing the ancestors of y. The variant is an order-preserving expression that turns this vector into a natural number.

See Appendix B for a proof of the total-correctness verification conditions generated for find-set.

We now discuss Theorem 5.3, 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 5.3. Hence this result immediately follows from total correctness of find-set. In Isabelle/HOL, similar 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 5.3 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.

7.2. Path compression and its effect

Path compression is a technique to decrease the depth of disjoint-set forests, 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 7.1 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 ∧ p₀ = p ]
4       w := x;
5       WHILE y ≠ p[w]
6         INV { path_compression_invariant p x y p₀ w }
7         VAR { p*[w]↓ }
8         DO
9           t := w;
10          w := p[w];
11          p[t] := y
12        OD
13      [ path_compression_postcondition p x y p₀ ]"
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, except according to line 5 this loop finishes immediately before reaching the root. 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:

$$\text{path\_compression\_precondition } p\ x\ y \Leftrightarrow \text{forest } p \wedge \text{point } x \wedge \text{point } y \wedge y = \text{root } p\ x$$

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

$$\text{path\_compression\_invariant } p\ x\ y\ p_0\ w \Leftrightarrow$$
$$\text{path\_compression\_precondition } p\ x\ y \wedge \text{forest } p_0 \wedge \text{fc}(p) = \text{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1$$
$$\wedge \text{ point } w \wedge w \sqsubseteq p_0{}^*[x] \wedge y = \text{root } p\ w \wedge p_0[(p_0{}^*[x] - p_0{}^*[w]) \mapsto y] = p$$

First, the invariant requires that p_0 is a forest. Second, $\text{fc}(p) = \text{fc}(p_0)$ states that the components of p do not change, that is, p represents the same disjoint sets. Third, $p \sqcap 1 = p_0 \sqcap 1$ states that the roots of the component trees of p do not change. Next, node w is an ancestor of x in p_0, and y is the representative in p of the set containing w.

The final condition $p_0[(p_0{}^*[x] - p_0{}^*[w]) \mapsto y] = p$ shows the effect of path compression while the loop is in progress. This specification contains an update of array p_0 at all nodes that are in vector $p_0{}^*[x] - p_0{}^*[w]$. Hence, the parent array is obtained from the original parent array p_0 by updating to value y the entries for nodes that are ancestors of x but not ancestors of w. These are just the nodes between x inclusively and w exclusively.

The postcondition is part of the invariant:

$$\text{path\_compression\_postcondition } p\ x\ y\ p_0 \Leftrightarrow$$
$$\text{forest } p \wedge y = \text{root } p\ x \wedge \text{fc}(p) = \text{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1 \wedge p_0[p_0{}^*[x] \mapsto y] = p$$

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. The final condition $p_0[p_0{}^*[x] \mapsto y] = p$ expresses the precise effect of path compression: the resulting parent array p is obtained from p_0 by updating the entries for ancestors of x to the value y.

Appendix A discusses a selection of results used for maintaining the invariant; see Theorem 10.

In Section 6.2 we showed how disjoint-set forests can be initialised either by a loop applying make-set to elements one at a time or by a single assignment to the parent array. We now discuss an implementation of path compression similarly by one assignment.

Path compression updates the parent array at all nodes on the path from the start x to the root y of the same tree. The set of updated nodes therefore comprises all ancestors of x, obtained by the vector $p^*[x]$. Moreover, each update assigns the same value y to the parent. This is also evident from the postcondition of path compression. Hence the effect of path compression is obtained by the single assignment:

$$p[p^*[x]] := y$$

The following result confirms this:

```
1   theorem path_compression_assign: 🦑
2      "VARS p
3         [ path_compression_precondition p x y ∧ p₀ = p ]
4         p[p*[x]] := y
5         [ path_compression_postcondition p x y p₀ ]"
6      apply vcg_tc_simp
7      – proof of one verification condition omitted
```

Such relational assignments can be used to obtain higher-level, more concise implementations. Moreover the underlying array updates can be used in specifications as the postcondition of path compression demonstrates.

7.3.   The find-set operation with path compression and its effect

Using the technique of Section 6.1 we extract function definitions for the find-set operation of Section 7.1 and the path-compression operation of Section 7.2. This allows us to combine the two programs into the following one with a simple correctness proof:

```
1   theorem find_set_path_compression: 🦑
2      "VARS p y
3         [ find_set_precondition p x ∧ p₀ = p ]
4         y := find_set p x;
5         p := path_compression p x y
6         [ path_compression_postcondition p x y p₀ ]"
7      apply vcg_tc_simp
8      using find_set_function find_set_postcondition_def find_set_precondition_def
9         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)"

The function extracted for find_set_path_compression returns values which satisfy the postcondition of path_compression. It no longer contains the information that the return value was constructed using find_set and path_compression. In other words, part of the connection between the function extracted for find_set_path_compression and the composition of the functions extracted for find_set and path_compression is lost.

   We can prove that the root of the tree returned by find_set_path_compression is the result of find_set. We can also prove that the forest returned by find_set_path_compression has the same roots as the forest returned by the composition of find_set and path_compression. Moreover we can prove that the two

forests have the same semantics, that is, represent the same equivalence relation. To show that the two forests are equal, however, it is necessary to know precisely how path compression modifies the parent array, which is captured by the postcondition of path_compression given in Section 7.2.

7.4. Path splitting

As shown in Section 7.3, finding the representative with path compression takes two iterations. Alternative path-compression techniques have been developed which require only one iteration and may perform better in practice [35, 36]. We discuss two such techniques in this and the next section.

Path splitting makes one iteration from the start node to the root of its tree and updates the parent array to point to grandparents along the way. This shortens paths towards the root for subsequent find-set operations:

```
1   theorem find_path_splitting: 🧊
2     "VARS p t y
3       [ find_set_precondition p x ∧ p₀ = p ]
4       y := x;
5       WHILE y ≠ p[y]
6         INV { path_splitting_invariant p x y p₀ }
7         VAR { p*[y]↓ }
8         DO
9           t := p[y];
10          p[y] := p[p[y]];
11          y := t
12        OD
13      [ path_splitting_postcondition p x y p₀ ]"
14    apply vcg_tc_simp
15    – proof of three verification conditions omitted
```

The program is similar to find_set except the additional compression update in line 10, which makes it necessary to use temporary variable t to store the parent of y in line 9. The verification condition generator with simplification eliminates the temporary variable so the condition to maintain the invariant is the same that would be obtained for the simultaneous assignment:

$$y, p[y] := p[y], p[p[y]]$$

The pre- and postconditions of find_path_splitting are the same as those of find_set_path_compression except the effect of path splitting is different:

path_splitting_postcondition $p \ x \ y \ p_0 \Leftrightarrow$
    forest $p \wedge y = \text{root } p \ x \wedge \text{fc}(p) = \text{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1 \wedge p_0[p_0{}^*[x] \mapsto (p_0 \cdot p_0)^\mathsf{T}] = p$

The update $p_0[p_0{}^*[x] \mapsto (p_0 \cdot p_0)^\mathsf{T}] = p$ specifies that all ancestors of x are updated according to the grandparent relation $p_0 \cdot p_0$. Hence also the effect of path splitting can be expressed as a single assignment to the parent array.

The postcondition can be proved using the loop invariant:

> path_splitting_invariant p x y p_0 $\Leftrightarrow$
>   find_set_precondition p x $\wedge$ point y $\wedge$ $y \sqsubseteq p_0{}^*[x]$ $\wedge$ forest p_0
> $\wedge\; p_0[(p_0{}^*[x] - p_0{}^*[y]) \mapsto (p_0 \cdot p_0)^{\mathsf{T}}] = p$

Here again, while the loop is in progress only the nodes between x inclusively and y exclusively have been updated, this time to point to their grandparents. A number of useful properties, including some which were part of the invariant of path_compression, follow from path_splitting_invariant:

> path_splitting_invariant p x y p_0 $\Rightarrow$
> $p[y] = p_0[y] \wedge p[p[y]] = p_0[p_0[y]] \wedge \mathrm{fc}(p) = \mathrm{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1$

This is because the invariant reflects the precise effect of path splitting.

7.5. Path halving

Path halving is similar to path splitting except the parent of only every second node is updated to the grandparent:

```
1   theorem find_path_halving: 🔧
2      "VARS p y
3        [ find_set_precondition p x ∧ p0 = p ]
4        y := x;
5        WHILE y ≠ p[y]
6          INV { path_halving_invariant p x y p0 }
7          VAR { p*[y]↓ }
8          DO
9            p[y] := p[p[y]];
10           y := p[y]
11         OD
12        [ path_halving_postcondition p x y p0 ]"
13     apply vcg_tc_simp
14     – proof of three verification conditions omitted
```

The updates in lines 9–10 amount to the simultaneous assignment:

> $y, p[y] := p[p[y]], p[p[y]]$

It shows that the loop proceeds from each node to its grandparent. This is reflected in the postcondition:

> path_halving_postcondition p x y p_0 $\Leftrightarrow$
>   forest p $\wedge$ $y = $ root p x $\wedge$ $\mathrm{fc}(p) = \mathrm{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1$
> $\wedge\; p_0[(p_0 \cdot p_0)^*[x] \mapsto (p_0 \cdot p_0)^{\mathsf{T}}] = p$

Vector $(p_0 \cdot p_0)^*[x]$ contains the nodes reachable in an even number of steps from x. Again this means that the effect of path halving can be expressed as a single assignment to the parent array. The following invariant is used to prove the postcondition:

path_halving_invariant $p\ x\ y\ p_0 \Leftrightarrow$
   find_set_precondition $p\ x \wedge$ point $y \wedge y \sqsubseteq p_0{}^*[x] \wedge$ forest p_0
   $\wedge\ p_0[((p_0 \cdot p_0)^*[x] - p_0{}^*[y]) \mapsto (p_0 \cdot p_0)^\mathsf{T}] = p$

Vector $(p_0 \cdot p_0)^*[x] - p_0{}^*[y]$ contains all nodes which are reachable from x in an even number of steps but not an ancestor of y. This is every second node between x inclusively and y exclusively. Again the precise knowledge about the effect of path halving allows us to derive useful properties so they do not have to be included in the invariant:

path_halving_invariant $p\ x\ y\ p_0 \Rightarrow$
   $p[y] = p_0[y] \wedge p[p[y]] = p_0[p_0[y]] \wedge \mathrm{fc}(p) = \mathrm{fc}(p_0) \wedge p \sqcap 1 = p_0 \sqcap 1$

Appendix A discusses properties used for verifying the correctness of path splitting and path halving; see Theorem 11.

8.   Verifying disjoint-set forests – union and ranks

In this section we implement the union-sets operation of disjoint-set forests and the union-by-rank strategy and verify correctness of these algorithms. Section 8.1 covers the basic union-sets operation without ranks, and recalls the concept of ranks. Peano structures used to represent numbers are studied in Section 8.2. Section 8.3 covers the initialisation of ranks. Union-by-rank is implemented and verified in Section 8.4.

8.1.   The union-sets operation

We start by considering 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:
2      "VARS p r s
3         [ union_sets_precondition p x y ∧ p₀ = p ]
4         r := find_set p x;
5         p := path_compression p x r;
6         s := find_set p y;
7         p := path_compression p y s;
8         p[r] := s
9         [ union_sets_postcondition p x y p₀ ]"
10     apply vcg_tc_simp
11     – proof of one verification condition omitted
```

We have also verified an alternative implementation using the combined find_set_path_compression of Section 7.3. Note how the forest p is threaded through both occurrences of path_compression, where it may be modified, before line 8 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 represents a forest and x and y represent single nodes:

> union_sets_precondition $p\ x\ y \Leftrightarrow$ forest $p \wedge$ point $x \wedge$ 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:

> union_sets_postcondition $p\ x\ y\ p_0 \Leftrightarrow$ forest $p \wedge \mathrm{fc}(p) = \mathrm{wcc}(p_0 \sqcup (x \cdot y^\mathsf{T}))$

To get the latter equivalence relation, we add the pair (x, y) to the initial equivalence relation p_0 and compute its equivalence closure. The pair (x, y) is described by $x \cdot y^\mathsf{T}$ since x and y are points.

Appendix A discusses a selection of results used for proving the correctness union-sets; see Theorem 12.

Correctness of the union-sets operation would not be affected if the assignment in line 8 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 [36]. 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. We implement this extension in the relation-algebraic framework using basic Peano arithmetic.

8.2. Peano structures

Disjoint-set forests and their algorithms can be implemented using many different data structures in Isabelle/HOL; for example, [37] uses lists and arrays over the built-in type of natural numbers. The aim of this paper is to verify the correctness of disjoint-set forests using relation-algebraic structures, hence we need to encode natural numbers as elements of relation algebras. Peano arithmetic offers a simple yet suitable foundation for such an encoding.

Peano arithmetic describes the natural numbers based on the constant 0 and a successor operation $succ$. In a relational setting, numbers are represented as points. A point Z represents the number 0. An injective mapping S represents the successor operation $succ$. The successor of a number represented by point x is obtained by the relational operation $\mathsf{S}^\mathsf{T} \cdot x$. The composition $\mathsf{S} \cdot x$ results in the predecessor of x. Our axiomatisation of Peano structures using relation algebras is based on [38, 39].

The following lists the arithmetic operations we use for union-by-rank:

$0 = \mathsf{Z}$
$succ(x) = \mathsf{S'}^\mathsf{T} \cdot x$
$x < y \Leftrightarrow x \sqsubseteq \mathsf{S'}^+ \cdot y$

For technical reasons discussed below we use a modified successor operation $\mathsf{S}' = \mathsf{S} - \mathsf{Z}^\mathsf{T}$ which also works for finite sets of numbers $\{0, \ldots, m - 1\}$. The rest of this section studies properties of this relational encoding of numbers and discusses some challenges of formalising it in Isabelle/HOL.

Ideally, the rank information of nodes would be represented by a heterogeneous relation between the set of nodes and the set of natural numbers. However, the class hierarchy for semirings, lattices, relation algebras, Kleene algebras and similar structures in Isabelle/HOL models only homogeneous relations. A formalisation using set-theoretic functions and heterogeneous relations would be possible, but this would not support the use of algebras for verification because no formalisation of heterogeneous relation algebras is available in Isabelle/HOL.

Moreover the set of nodes in disjoint-set forests is finite (which ensures, for example, that initialisation of the forest terminates). Hence we have to use a finite subset of the natural numbers which has as many elements as there are nodes in the disjoint-set forest. Since the ranks in a forest with m nodes will be at most $m - 1$, it is enough to use the numbers $0, \ldots, m - 1$.

We therefore axiomatise the zero constant Z and the partial successor relation S' that maps each number except $m - 1$ to its successor. This is achieved in two steps. We first consider the total successor relation S that maps each number to its successor modulo m. The desired partial successor relation S' is derived from it in the second step.

For the first step we simply use a subset of the Peano axioms:

1. point Z

2. mapping S

3. injective S

4. $\mathsf{S}^{\mathsf{T}*} \cdot \mathsf{Z} = \top$

Axiom 1 specifies that 0 is a number. Axiom 2 specifies that every number has exactly one successor. Axiom 3 specifies that numbers with the same successor are equal. Axiom 4 specifies that every number can be obtained from 0 by finitely many applications of the successor. We omit the Peano axiom $\mathsf{S} \cdot \mathsf{Z} = \bot$ which would specify that 0 is not the successor of any number.

Since this is a weakening of the Peano axioms, the natural numbers are a model, but so are the numbers modulo m. The following result gives consequences of the above four axioms.

Theorem 6.

1. $\mathsf{S} \sqcap 1 \sqsubseteq \mathsf{Z}$. 🧩

2. $\mathsf{Z} \sqsubseteq \mathsf{S}^*$. 🧩

3. $\mathsf{S}^{\mathsf{T}*} \cdot \mathsf{S}^* = \top$. 🧩

4. $\mathsf{S}^* \sqcup \mathsf{S}^{\mathsf{T}*} = \top$. 🧩

5. $\mathsf{Z} \sqcup (\mathsf{S}^\mathsf{T} \cdot \top) = \top$. 🧩

Proof: See Appendix B for proofs of Theorems 6.1 and 6.5, and [20] for all proofs.

Part 4 shows that the preorder S^* is connex, that is, any two numbers are comparable. Part 5 shows that every number is 0 or the successor of a number.

The successor operation provides a convenient way to compare two natural numbers. Namely, $x \leq y$ if y can be reached from x by finitely many applications of the successor, formally $y \sqsubseteq S^{\mathsf{T}*} \cdot x$ or equivalently $x \sqsubseteq S^* \cdot y$ for points x and y. This does not work for numbers modulo m since comparison depends on the chosen representative.

In the second step, we therefore derive a partial successor relation $S' = S - Z^{\mathsf{T}}$, which computes the successor for all numbers except $m - 1$. The modified S' is obtained from S by removing any entries that map to 0; in the intended model this is just the mapping from $m - 1$ to 0.

We have verified the soundness of the relational Peano axioms in Isabelle/HOL. Specifically, we have defined a model of natural numbers modulo m as relations and proved that this model satisfies the above axioms for Z and S. We have also shown that the natural numbers below m, taken from Isabelle/HOL's nat type, order-embed into our relational model. The order-embedding preserves the arithmetic operations 0, partial successor and the $<$ relation on this subset of nat. Moreover, the relational model satisfies the axioms of all algebraic structures used for the results of this paper. See the Isabelle/HOL theories for further details about this model and proofs of these results.

The following result shows that S' satisfies most of the properties of S and some additional properties.

Theorem 7.

1. S' is univalent. 🧩

2. S' is injective. 🧩

3. $S'^{\mathsf{T}*} \cdot Z = \top$. 🧩

4. S' is irreflexive. 🧩

5. $Z \sqsubseteq S'^*$. 🧩

6. $S'^{\mathsf{T}*} \cdot S'^* = \top$. 🧩

7. $S'^* \sqcup S'^{\mathsf{T}*} = \top$. 🧩

8. $Z \sqcup (S'^{\mathsf{T}} \cdot \top) = \top$. 🧩

9. $S' \cdot Z = \bot$. 🧩

Proof: See Appendix B for a proof of Theorem 7.3 and [20] for all proofs.

Part 9 shows that 0 has no predecessor under S'. We next consider the predecessor of 0 under S given by the relation $M = S \cdot Z$. The following result characterises the two models: the natural numbers and the natural numbers modulo m.

Theorem 8.

1. M is a point if and only if S is surjective. 🐞

2. $M \neq \bot$ if and only if S is surjective. 🐞

3. $M = \bot$ or M is a point. 🐞

Proof: See Appendix B or [20] for a proof of Theorem 8.

In the natural-number model, 0 has no predecessor under S so S is not surjective; then $M = \bot$. In the modulo-m model, 0 has exactly one predecessor under S and S is bijective; then M is a point.

The relation M facilitates an alternative characterisation of S' as the following result shows. We also characterise the model $m = 1$, that is, $M = Z$.

Theorem 9.

1. $S' = S - M$. 🐞

2. $M = Z$ if and only if $1 = \top$. 🐞

3. S is irreflexive if $M \neq Z$. 🐞

Proof: See Appendix B for a proof of Theorem 9.1 and [20] for all proofs.

According to part 1, the partial S' can also be obtained from S by removing the entry in row $m - 1$. Part 3 shows that S is irreflexive unless $m = 1$.

8.3. Initialising ranks

For the union-by-rank strategy to work, the rank of each element needs to be initialised to 0. To this end we augment init-sets from Section 6.2 by adding the assignment in line 9:

```
1    theorem init_ranks: 🐞
2       "VARS h p x rank
3          [ True ]
4          FOREACH x
5             USING h
6             INV { p − h = 1 − h ∧ rank − h = Zᵀ − h }
7             DO
8                p := make_set p x;
9                rank[x] := 0
10            OD
11         [ p = 1 ∧ forest p ∧ h = ⊥ ∧ rank = Zᵀ ∧ rank_property p rank ]"
12      apply vcg_tc_simp
13      – proof of one verification condition omitted
```

The overall effect is to set $rank := \mathsf{Z}^\mathsf{T}$ in addition to setting $p := 1$. The loop invariant reflects the partial state of these updates. The postcondition establishes additional properties of the rank array which are discussed below.

8.4. Union-by-rank

The implementation of union-by-rank extends the algorithm presented in Section 8.1:

```
 1   theorem union_sets_by_rank: 🐉
 2     "VARS p r s rank
 3       [ union_sets_precondition p x y ∧ p₀ = p ∧ rank_property p rank ]
 4       r := find_set p x;
 5       p := path_compression p x r;
 6       s := find_set p y;
 7       p := path_compression p y s;
 8       IF r ≠ s THEN
 9         IF rank[r] < rank[s] THEN
10           p[r] := s
11         ELSE
12           p[s] := r;
13           IF rank[r] = rank[s] THEN
14             rank[r] := succ(rank[r])
15           FI
16         FI
17       FI
18       [ union_sets_postcondition p x y p₀ ∧ rank_property p rank ]"
19   apply vcg_tc_simp
20   – proof of one verification condition omitted
```

Lines 4–7 remain unchanged and obtain the root r of x and the root s of y. If the two roots are identical, x and y are already in the same set and nothing needs to be done. Otherwise either the parent of r is set to s in line 10 or the other way around in line 12, depending on how their ranks compare in line 9. The node with smaller rank needs to point to the node with larger rank to obtain trees with smaller heights. If the ranks of r and s are equal, as checked in line 13, the rank of the remaining root r needs to be incremented, which happens in line 14.

We show that the rank array satisfies three properties which are established or preserved by the union-find operations:

$$\text{rank\_property } p \; rank \Leftrightarrow$$
$$\text{mapping } rank \wedge (p-1) \cdot rank \sqsubseteq rank \cdot \mathsf{S'}^+ \wedge |\text{roots } p| \leq |\overline{\mathsf{S'}^+ \cdot rank^\mathsf{T} \cdot \top}|$$

First, every node has a rank, that is, the rank array is a mapping. This is non-trivial to maintain because only finitely many numbers can be represented. Specifically, since the successor $\mathsf{S'}$ is partial, the assignment in line 14 of union_sets_by_rank would not increment the largest number that can be represented. A proof that this rank property is maintained ensures that an overflow never occurs.

Second, the rank of a node is strictly smaller than the rank of its parent, except if the node is a root. This implies that the rank of a node is an upper bound on the height of its subtree. The left-hand side $(p-1) \cdot rank$ relates a node to the rank of its parent if the parent is a different node. The right-hand side $rank \cdot \mathsf{S}'^+$ relates a node to all numbers greater than its rank. The inclusion $(p-1) \cdot rank \sqsubseteq rank \cdot \mathsf{S}'^+$ specifies that the parent's rank must be among those numbers. We can verify this interpretation by deriving the logical meaning of this relation-algebraic formula:

$$(p - 1) \cdot rank \sqsubseteq rank \cdot \mathsf{S}'^+$$
$$\Leftrightarrow \forall a, b : (a, b) \in (p - 1) \cdot rank \Rightarrow (a, b) \in rank \cdot \mathsf{S}'^+$$
$$\Leftrightarrow \forall a, b : (\exists c : (a, c) \in (p - 1) \wedge (c, b) \in rank) \Rightarrow (\exists d : (a, d) \in rank \wedge (d, b) \in \mathsf{S}'^+)$$
$$\Leftrightarrow \forall a, b : (\exists c : (a, c) \in p \wedge (a, c) \notin 1 \wedge rank(c) = b) \Rightarrow (\exists d : rank(a) = d \wedge d < b)$$
$$\Leftrightarrow \forall a, b : (\exists c : p(a) = c \wedge a \neq c \wedge rank(c) = b) \Rightarrow rank(a) < b$$
$$\Leftrightarrow \forall a, b : a \neq p(a) \wedge rank(p(a)) = b \Rightarrow rank(a) < b$$
$$\Leftrightarrow \forall a : a \neq p(a) \Rightarrow rank(a) < rank(p(a))$$

Third, the number of roots in the disjoint-set forest (which is the number of disjoint sets) is not larger than $m - k$ where m is the total number of nodes and k is the maximum rank of any node. This property is established since initially all ranks are 0 and every node is a root. To show that this property is maintained, note that whenever the rank of a node increases by 1 in line 14, the former root s has been linked to r in line 12 and therefore the number of roots has decreased by 1. In other cases the number of roots decreases without changing the maximum rank. Because there is always at least one root, $1 \leq m - k$ ensures that the maximum rank will never exceed $m - 1$, that is, ranks never overflow.

To compare the number of roots and $m - k$ we compare the cardinality of the set of roots and the set of numbers between k inclusively and m exclusively. The following definition compares the cardinality of two sets using the existence of an injective univalent relation between the vectors v and w representing the sets:

$$|v| \leq |w| \Leftrightarrow \exists i \, . \, \text{injective } i \wedge \text{univalent } i \wedge v \sqsubseteq i \cdot w$$

See [40] for a more comprehensive axiomatisation of the cardinalities of heterogeneous relations. The vector $\mathsf{S}'^+ \cdot rank^\mathsf{T} \cdot \top$ contains a number if there is a greater number that is the rank of a node. These are just the numbers smaller than the maximum rank, so the complement of the vector gives the desired numbers between k and m. We can verify this interpretation by deriving the logical meaning of this relation-algebraic expression:

$$(a, b) \in \overline{\mathsf{S}'^+ \cdot rank^\mathsf{T} \cdot \top}$$
$$\Leftrightarrow \neg (a, b) \in \mathsf{S}'^+ \cdot rank^\mathsf{T} \cdot \top$$
$$\Leftrightarrow \neg \exists c, d : (a, c) \in \mathsf{S}'^+ \wedge (c, d) \in rank^\mathsf{T} \wedge (d, b) \in \top$$
$$\Leftrightarrow \neg \exists c, d : a < c \wedge (d, c) \in rank$$
$$\Leftrightarrow \neg \exists c, d : a < c \wedge rank(d) = c$$
$$\Leftrightarrow \neg \exists d : a < rank(d)$$
$$\Leftrightarrow \forall d : rank(d) \leq a$$
$$\Leftrightarrow k \leq a$$

Proof that union-by-rank maintains the rank property involves showing that it is maintained by path compression, which changes the parent relation. To this end we use the precise effect of path compression described in Section 7.2 and captured in its postcondition. This avoids adding the rank property to the loop invariant of path compression.

9. Related work and limitations

In this section we review work related to the contributions of this paper and discuss its limitations.

9.1. 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 [41] for updating and reading state vectors, which map variables to values. Arrays are modelled as mappings in [42] and selective array updates are defined as updates of mappings. Such updates are more formally defined in [43]. Functional and relational override in Z [44] dates back to [45]. Overwriting one relation with another also appears in [25] where it is used for pointer structures. Axioms for state attributes and array access are given in [2]; some of these are used for lenses [26]. A definition of general updates in Kleene algebras with domain is given in [46]. The relation-algebraic semantics given in the present paper specialises definitions given in [2, 25, 26, 44–46] to selective array updates studied in [41–43].

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 [5, 10, 15]. Also relevant are relational models of stores modelling pointer structures and using relational overwrite operations [15, 25, 47, 48].

There are several formally verified implementations of disjoint-set forests. A persistent version of the data structure is verified in Coq by [49]. The specification is in terms of predicate logic and the implementation is based on a mathematical model of ML including references. See [37] for a verification using separation logic in Isabelle/HOL also based on a logical specification; the programs can be translated to executable code in various target languages. Program complexity and correctness of an OCaml implementation is proved in [50] 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. We also mention the manual derivation of implementations of union and find from an abstract specification by data refinement in [29, 51]; it does not integrate path compression or union-by-rank. The present paper gives a relation-algebraic specification and proof, which does not cover complexity of the union and find operations.

9.2. Limitations

The approach used in this paper has two limitations. First, we only cover arrays with the same index and value sets. This is because Isabelle/HOL's class hierarchy currently only supports homogeneous relation algebras. In this regard, we argue as follows. Firstly, we value formally verified results

which improve confidence compared to manual proofs. Secondly, the choice of Isabelle/HOL is due to having extensive libraries for relation algebras and a high degree of automation. Thirdly, the choice of using classes of algebraic structures instead of concrete relations follows our aim to further develop and strengthen algebraic techniques, which have been useful in other verification efforts. Finally, this limitation is not fundamental but can be overcome by providing a library for heterogeneous relation algebras with automation support, however this is outside the scope of the current paper.

The second limitation is that we do not prove the time complexity of the disjoint-forest operations. We are aware of two libraries for time-complexity proofs in Isabelle/HOL: Imperative HOL with Time [52] and LLVM with Time [53]. Currently, neither is part of the Archive of Formal Proofs or sufficiently documented to be easily applicable to new projects. Again, this is not a fundamental limitation.

10. 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, path splitting, path halving and union-by-rank this way and proved their correctness.

We have used single relational updates to express the precise effect of various path-compression techniques. This was helpful for specification and for reasoning about the lower-level implementations. It could also feature in refinement-based program development [54].

To deal with union-by-rank it was necessary to extend relation algebras by basic arithmetic. This is usually done with heterogeneous relations which are, however, not supported by the current relation-algebraic framework in Isabelle/HOL. On the one hand, this required working in an alternative model that represents only a restricted subset of numbers. On the other, such a setting shares properties with fixed-size number types available in many programming languages for which it can be important to show that overflows do not occur.

Examples in Section 2 show that Kleene relation algebras can be used to express complex logical specifications in compact, readable and understandable ways. Another such example is the final condition $p_0[(p_0{}^*[x] - p_0{}^*[w]) \mapsto y] = p$ of the path compression invariant in Section 7.2, which expresses the effect of path compression using a single relational update. Further examples are the conditions describing the effect of path splitting and path halving using relational updates in Sections 7.4 and 7.5. The rank properties discussed in Section 8.4 require more familiarity with expressing specifications using relational operations. In these cases it is always possible to link relation-algebraic expressions to their logical or set-theoretic interpretation, which helps constructing and understanding such formulas.

In this paper we have focused on data represented by associative arrays. Relation algebras can be extended by other types of data using the domain constructions of products, sums, quotients, subsets and power sets discussed, for example, in [55]. Various data types and their application to sorting algorithms, optimisation problems, dynamic programming and greedy algorithms in a relation-algebraic framework are studied in [16].

A future 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 [33]. While Kruskal's algorithm has already been formally verified in Isabelle/HOL [56], the relation-algebraic specification and implementation mean that the same algorithm is correct for various optimisation problems as discussed in Section 2. A further direction of research is to consider how relation-algebraic methods can support complexity analysis of algorithms.

Acknowledgement

A.   Further properties of forests

In this appendix we discuss results used for verifying the correctness of path compression, path splitting, path halving and union-sets.

We first discuss a selection of results used for maintaining the invariant for path compression in Section 7.2. Part of the maintenance is to show that the parent relation (without self-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 ancestor y of w does not introduce cycles (ignoring self-loops).

Theorem 10.

1. $p[w \mapsto y] - 1$ is acyclic if $p - 1$ is acyclic, w and y are points and $y \sqsubseteq p^*[w]$. 🐝

2. $x \sqcap p^* = (x \sqcap 1) \sqcup ((x \sqcap p) \cdot (\overline{x} \sqcap p)^*)$ if x is a point. 🐝

3. $x \sqcap p^* = (x \sqcap 1) \sqcup (x \sqcap (p \sqcap \overline{x}^\mathsf{T})^+)$ if x is a point. 🐝

4. $x \sqcap y = \bot$ if x and y are points such that $x \neq y$. 🐝

Proof: See Appendix B for proofs of Theorems 10.1 and 10.4, and [20] for all proofs.   □

Parts 2 and 3 optimise iterations similar to [15, Lemma 4]; for related techniques see also [9]. 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 of part 2 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 4 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 self-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 4 of the previous theorem.

Next, we present properties involving the grandparent relation, which are used for verifying the correctness of path splitting and path halving in Sections 7.4 and 7.5.

Theorem 11.

1. $(x \cdot x)^{\mathsf{T}*} \sqcap (x^\mathsf{T} \cdot (x \cdot x)^{\mathsf{T}*}) = (1 \sqcap x) \cdot ((x \cdot x)^{\mathsf{T}*} \sqcap (x^\mathsf{T} \cdot (x \cdot x)^{\mathsf{T}*}))$ if x is univalent and $x - 1$ is acyclic.

2. $x^+ \sqcap 1 = (x \cdot x) \sqcap 1 = x \sqcap 1$ if $x - 1$ is acyclic.

3. $(x \cdot x) - 1 \sqsubseteq x^* - 1 \sqsubseteq (x - 1)^+$.

4. $x[y \mapsto x[x[y]]] \sqsubseteq x \sqcup (x \cdot x)$ if y is a point.

5. $x[y \mapsto x[x[y]]] \sqcap 1 = x \sqcap 1$ if $x - 1$ is acyclic and y is a point.

6. $\mathrm{fc}(x[y \mapsto x[x[y]]]) = \mathrm{fc}(x)$ if x is a mapping and y is a point.

7. $x[y \mapsto (x \cdot x)^\mathsf{T}] \sqsubseteq x \sqcup (x \cdot x)$.

8. $x[y \mapsto (x \cdot x)^\mathsf{T}] - 1$ is acyclic if $x - 1$ is acyclic.

9. $x[y \mapsto (x \cdot x)^\mathsf{T}]$ is a forest if x is a forest and y is a vector.

Proof: See Appendix B for proofs of Theorems 11.3, 11.7, 11.8 and 11.9, and [20] for all proofs.

Part 1 shows that if between two nodes of a forest there is a path of even length and one of odd length, the start node must be a root. Part 2 shows that the self-loops of the grandparent relation are the self-loops of the parent relation. Parts 5–6 show that updating nodes to their grandparents does not change the semantics of the disjoint-set forest or its roots. Parts 8–9 show that such updates maintain the forest property.

Finally, we discuss a selection of results used for proving the correctness of union-sets in Section 8.1. Part 1 of the following result is similar to [12, Proposition 3]; also see [9, Section 4.3]. 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 12.

1. $(x \sqcup y)^+ = y^+ \sqcup (y^* \cdot x \cdot y^*)$ if x is an arc. 🔸

2. $p[w \mapsto y] - 1$ is acyclic if $p - 1$ is acyclic, w and y are points, and $y \sqcap (p^* \cdot w) = \bot$. 🔸

3. $p[w \mapsto w] - 1$ is acyclic if $p - 1$ is acyclic and w is a point. 🔸

Proof: See Appendix B for proofs of Theorems 12.1 and 12.2, and [20] for all proofs.

Parts 2 and 3 are similar to Theorem 10.1. 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 self-loops). Part 3 shows that creating a self-loop on w does not introduce a cycle (ignoring self-loops).

These results are used to show that the assignment in line 8 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 8 creates a self-loop and the correctness proof uses part 3 of the preceding result. Alternatively, it could be proved that the assignment in line 8 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.

B.   Example proofs

All results of this paper are proved in the Isabelle/HOL theory files available in the Archive of Formal Proofs [20]. In this appendix we prove some results as examples.

We first list a number of laws of Kleene relation algebras used in the proofs below. They are grouped by their assumptions and labelled for reference. In this list, $u, v, x, y, z \in S$ for a Kleene relation algebra S.

(1) Let u and v be vectors. Then (1.1) $u \sqcap (x \cdot y) = (u \sqcap x) \cdot y$ and (1.2) $(x \sqcap u^\mathsf{T}) \cdot y = x \cdot (u \sqcap y)$ and (1.3) $x \cdot (y \sqcap u^\mathsf{T}) = (x \cdot y) \sqcap u^\mathsf{T}$. Moreover (1.4) $u \sqcap v$ and $x \cdot u$ and $\overline{u}$ are vectors and (1.5) $u \sqcap v^\mathsf{T} = u \cdot v^\mathsf{T}$.

(2) Let $u \sqsubseteq 1$. Then (2.1) $(u \cdot \top) \sqcap x = u \cdot x$ and $u^\mathsf{T} = u$, hence (2.2) $x \sqcap 1 = x^\mathsf{T} \sqcap 1$.

(3) Let $u \cdot v \sqsubseteq 1$. Then $(u \sqcup v)^* = v^* \cdot u^*$.

(4) Let $u \sqsubseteq v$ and let v be injective. Then $u = v \sqcap (\top \cdot u)$.

(5) x is surjective if and only if $\top \cdot x = x$.

(6) Let u and v be bijective. Then (6.1) $u \sqsubseteq x \cdot v \Leftrightarrow v \sqsubseteq x^\mathsf{T} \cdot u$ and (6.2) $x \sqsubseteq y \cdot u \Leftrightarrow x \cdot u^\mathsf{T} \sqsubseteq y$ and (6.3) $x \sqsubseteq u^\mathsf{T} \cdot y \Leftrightarrow u \cdot x \sqsubseteq y$.

(7) Let u be univalent. Then $u \cdot (x \sqcap y) = (u \cdot x) \sqcap (u \cdot y)$.

(8) Let u and v be mappings. Then (8.1) $u \cdot v$ is a mapping and (8.2) $\overline{u \cdot x} = u \cdot \overline{x}$.

(9) Let u be a mapping and let v be a point. Then $u \sqcap v$ is an arc.

(10) Let u be an arc. Then $u \cdot x \cdot u \sqsubseteq u$.

(11) (11.1) $x \sqcap y \sqsubseteq z \Leftrightarrow x \sqsubseteq z \sqcup \overline{y}$ and (11.2) $x \cdot y \sqsubseteq z \Leftrightarrow x^\mathsf{T} \cdot \overline{z} \sqsubseteq \overline{y}$ and (11.3) $x \cdot y \sqsubseteq \overline{1} \Leftrightarrow y \cdot x \sqsubseteq \overline{1}$.

(12) (12.1) $x^* \cdot x^* = x^*$ and (12.2) $x^{**} = x^*$ and (12.3) $x^{++} = x^+$.

We furthermore use properties of semirings, lattices, Boolean algebras and partial orders without explicitly mentioning them. Details are available in the Isabelle/HOL proofs.

Results are grouped by content not by the order in which they are proved. Some proofs refer to later results; formal verification of course ensures that there are no cyclic dependencies.

Proof of Theorem 1.8:
For the backward implication assume $y \sqcap x = y \cdot z^\mathsf{T}$, then

$$x[y] = x^\mathsf{T} \cdot y = (x^\mathsf{T} \sqcap y^\mathsf{T}) \cdot \top = (x \sqcap y)^\mathsf{T} \cdot \top = (y \cdot z^\mathsf{T})^\mathsf{T} \cdot \top = z \cdot y^\mathsf{T} \cdot \top = z \cdot \top \cdot y = z \cdot \top = z$$

using the definition of array read, (1.2), distributivity of transposition over meet, the assumption, distributivity of transposition over composition and involution of transposition, (1.2), (5) and that z is a vector.

For the forward implication assume $x^\mathsf{T} \cdot y = z$, then we show $y \sqcap x \sqsubseteq y \cdot z^\mathsf{T}$ and $y \cdot z^\mathsf{T} \sqsubseteq y \sqcap x$. First,

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

using that composition with $\top$ is $\sqsubseteq$-increasing, (1.2), distributivity of transposition over composition and involution of transposition, the assumption and (1.5). Second,

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

using the assumption, distributivity of transposition over composition and involution of transposition, injectivity of y and that y is a vector.

In Isabelle/HOL, the user's contribution is to reduce the overall goal to the three subgoals discussed above, which is a natural way of breaking the proof into smaller chunks. Then each of the three subgoals is proved automatically by Sledgehammer using integrated automated theorem provers or SMT solvers [21]. $\square$

Proof of Theorem 2.8:
By definition of array write and read,

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

and therefore

$$x[y \mapsto x[z]][z \mapsto x[y]] = (z \sqcap (y^\mathsf{T} \cdot x)) \sqcup (\overline{z} \sqcap y \sqcap (z^\mathsf{T} \cdot x)) \sqcup (\overline{z} \sqcap \overline{y} \sqcap x) = u_1 \sqcup u_2 \sqcup u_3$$

for $u_1 = z \sqcap (y^\mathsf{T} \cdot x)$ and $u_2 = \overline{z} \sqcap y \sqcap (z^\mathsf{T} \cdot x)$ and $u_3 = \overline{z} \sqcap \overline{y} \sqcap x$. To show that $u_1 \sqcup u_2 \sqcup u_3$ is injective, it thus remains to show $u_i \cdot u_j{}^\mathsf{T} \sqsubseteq 1$ for each $i, j \in \{1, 2, 3\}$:

1. $u_1 \cdot u_1^{\mathsf{T}} \sqsubseteq z \cdot z^{\mathsf{T}} \sqsubseteq 1$ since z is injective.

2. $u_1 \cdot u_2^{\mathsf{T}} \sqsubseteq u_1 \cdot (y^{\mathsf{T}} \sqcap (x^{\mathsf{T}} \cdot z)) = (z \sqcap (z^{\mathsf{T}} \cdot x)) \cdot (y^{\mathsf{T}} \sqcap (x^{\mathsf{T}} \cdot y)) = (z \sqcap z^{\mathsf{T}}) \cdot x \cdot x^{\mathsf{T}} \cdot (y^{\mathsf{T}} \sqcap y) \sqsubseteq 1$ using (1.4) and (1.2), (1.1) and (1.3), (1.5) and injectivity of x, y and z.

3. $u_1 \cdot u_3^{\mathsf{T}} \sqsubseteq y^{\mathsf{T}} \cdot x \cdot (\overline{y}^{\mathsf{T}} \sqcap x^{\mathsf{T}}) = (y^{\mathsf{T}} \cdot x \cdot x^{\mathsf{T}}) \sqcap \overline{y}^{\mathsf{T}} \sqsubseteq y^{\mathsf{T}} \sqcap \overline{y}^{\mathsf{T}} = \bot \sqsubseteq 1$ using (1.4) and (1.3), and injectivity of x.

4. $u_2 \cdot u_1^{\mathsf{T}} \sqsubseteq 1$ follows by transposition from subgoal 2.

5. $u_2 \cdot u_2^{\mathsf{T}} \sqsubseteq y \cdot y^{\mathsf{T}} \sqsubseteq 1$ since y is injective.

6. $u_2 \cdot u_3^{\mathsf{T}} \sqsubseteq z^{\mathsf{T}} \cdot x \cdot (\overline{z}^{\mathsf{T}} \sqcap x^{\mathsf{T}}) = (z^{\mathsf{T}} \cdot x \cdot x^{\mathsf{T}}) \sqcap \overline{z}^{\mathsf{T}} \sqsubseteq z^{\mathsf{T}} \sqcap \overline{z}^{\mathsf{T}} = \bot \sqsubseteq 1$ similarly to the proof of subgoal 3.

7. $u_3 \cdot u_1^{\mathsf{T}} \sqsubseteq 1$ follows by transposition from subgoal 3.

8. $u_3 \cdot u_2^{\mathsf{T}} \sqsubseteq 1$ follows by transposition from subgoal 6.

9. $u_3 \cdot u_3^{\mathsf{T}} \sqsubseteq x \cdot x^{\mathsf{T}} \sqsubseteq 1$ since x is injective. $\qquad\qquad\qquad\qquad\qquad\square$

Proof of Theorem 3.2:
First, $\mathrm{wcc}(x)$ is symmetric by $(x \sqcup x^{\mathsf{T}})^{*\mathsf{T}} = (x \sqcup x^{\mathsf{T}})^{\mathsf{T}*} = (x^{\mathsf{T}} \sqcup x^{\mathsf{T}\mathsf{T}})^* = (x \sqcup x^{\mathsf{T}})^*$ using that Kleene star and transposition commute. Hence $(\mathrm{wcc}(x) \sqcup \mathrm{wcc}(x)^{\mathsf{T}})^* = \mathrm{wcc}(x)^* = (x \sqcup x^{\mathsf{T}})^{**} = (x \sqcup x^{\mathsf{T}})^*$ using (12.2), which shows that wcc is idempotent. Next, $x \sqsubseteq x \sqcup x^{\mathsf{T}} \sqsubseteq (x \sqcup x^{\mathsf{T}})^*$ shows that wcc is $\sqsubseteq$-increasing. Finally, wcc is $\sqsubseteq$-isotone since $\sqcup$, $^{\mathsf{T}}$ and * are $\sqsubseteq$-isotone. $\qquad\square$

Proof of Theorem 4.1:
Reflexivity of $\mathrm{fc}(x)$ follows by $1 = 1 \cdot 1 \sqsubseteq x^* \cdot x^{\mathsf{T}*}$ using reflexivity of the Kleene star. Transitivity of $\mathrm{fc}(x)$ follows by $x^* \cdot x^{\mathsf{T}*} \cdot x^* \cdot x^{\mathsf{T}*} \sqsubseteq (x \sqcup x^{\mathsf{T}})^* = x^* \cdot x^{\mathsf{T}*}$ using (12.1) and (3) since x is univalent. Symmetry of $\mathrm{fc}(x)$ follows by $(x^* \cdot x^{\mathsf{T}*})^{\mathsf{T}} = x^{\mathsf{T}*\mathsf{T}} \cdot x^{*\mathsf{T}} = x^{\mathsf{T}\mathsf{T}*} \cdot x^{\mathsf{T}*} = x^* \cdot x^{\mathsf{T}*}$ using that Kleene star and transposition commute. $\qquad\square$

Proof of total correctness of find-set:
The invariant comprises three conditions. The first two are established immediately from the precondition. The remaining condition $x \sqsubseteq p^*[x] = p^{\mathsf{T}*} \cdot x$ follows by reflexivity of the Kleene star.

For maintaining the invariant we need to show that it holds for $p[y]$ assuming that it holds for y and $y \neq p[y]$ indicating the while-loop has not yet terminated. The first condition for $p[y]$ follows immediately from the assumption for y. The second condition requires that $p[y]$ is a point, which follows by Theorem 1.7 since y is a point and p is a forest. The third condition follows by

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

using $y \sqsubseteq p^*[x]$ and properties of the Kleene star.

For total correctness we prove $\{z \mid z \sqsubseteq p^*[p[y]]\} \subset \{z \mid z \sqsubseteq p^*[y]\}$, which implies that the variant decreases since these sets are finite. The non-strict inclusion follows from

$$p^*[p[y]] = p^{\mathsf{T}*} \cdot p^{\mathsf{T}} \cdot y \sqsubseteq p^{\mathsf{T}*} \cdot y = p^*[y]$$

For strict inclusion we show y is in the difference of the two sets. First, y is in the bigger set since $y \sqsubseteq p^{\mathsf{T}*} \cdot y = p^*[y]$ by reflexivity of the Kleene star. Second, we derive a contradiction from assuming that y is in the smaller set. Since y and $p[y]$ are different points, we have $(p^{\mathsf{T}} \cdot y) \sqcap y = p[y] \sqcap y = \bot$ by Theorem 10.4, and therefore

$$p^{\mathsf{T}} \cdot y = ((p^{\mathsf{T}} \sqcap 1) \cdot y) \sqcup ((p^{\mathsf{T}} - 1) \cdot y) \sqsubseteq ((p^{\mathsf{T}} \cdot y) \sqcap y) \sqcup ((p^{\mathsf{T}} - 1) \cdot y) = (p^{\mathsf{T}} - 1) \cdot y$$

So y being in the smaller set yields

$$y \sqsubseteq p^*[p[y]] = p^{\mathsf{T}*} \cdot p^{\mathsf{T}} \cdot y = (p^{\mathsf{T}} - 1)^* \cdot p^{\mathsf{T}} \cdot y \sqsubseteq (p^{\mathsf{T}} - 1)^+ \cdot y$$

Hence $y \cdot y^{\mathsf{T}} \sqsubseteq (p^{\mathsf{T}} - 1)^+ \sqsubseteq \overline{1}$ using (6.2) since y is a point and that p is a forest. With injectivity of y this implies $y = \bot$, which contradicts that y is a point.

Finally we show that the invariant implies the postcondition when the while-loop terminates at $y = p[y]$. The first postcondition, y is a point, is immediate from the invariant. It remains to show $y = \text{root } p \, x$. Observe that $y = p[y] = p^{\mathsf{T}} \cdot y$ implies $y \cdot y^{\mathsf{T}} \sqsubseteq p^{\mathsf{T}}$ using (6.2), whence $y \cdot y^{\mathsf{T}} \sqsubseteq p \sqcap 1$ by transposition and since y is injective. Hence $y \sqsubseteq (p \sqcap 1) \cdot y \sqsubseteq \text{roots } p$ using (6.2), which together with $y \sqsubseteq p^*[x]$ from the invariant establishes $y \sqsubseteq \text{root } p \, x$. For the opposite inequality, first note $x \sqsubseteq p^* \cdot y$ is a consequence of the invariant using (6.1). Second,

$$p^{\mathsf{T}} \cdot p^* \cdot y = (p^{\mathsf{T}} \cdot p \cdot p^* \cdot y) \sqcup (p^{\mathsf{T}} \cdot y) \sqsubseteq (p^* \cdot y) \sqcup y = p^* \cdot y$$

since p is univalent. Together we obtain $p^{\mathsf{T}*} \cdot x \sqsubseteq p^* \cdot y$ by Kleene star induction. We also have $(p \cdot y) \sqcap \text{roots } p = (p \sqcap 1) \cdot p \cdot y \sqsubseteq p^{\mathsf{T}} \cdot p \cdot y \sqsubseteq y$ using (2.1) and (2.2) and that p is univalent. Therefore $p \cdot y \sqsubseteq y \sqcup \overline{\text{roots } p}$ using (11.1). Next, $p \cdot \overline{\text{roots } p} \sqsubseteq \overline{\text{roots } p}$ by Theorem 5.2 and (11.2). Hence $p \cdot (y \sqcup \overline{\text{roots } p}) = (p \cdot y) \sqcup (p \cdot \overline{\text{roots } p}) \sqsubseteq y \sqcup \overline{\text{roots } p}$. Kleene star induction and a previous fact imply $p^{\mathsf{T}*} \cdot x \sqsubseteq p^* \cdot y \sqsubseteq y \sqcup \overline{\text{roots } p}$. Using (11.1) we conclude root $p \, x \sqsubseteq y$. □

Proof of Theorems 6.1 and 6.5:
For Theorem 6.1 we have $(\mathsf{S} \sqcap 1) \cdot \mathsf{S}^{\mathsf{T}} \sqsubseteq (\mathsf{S} \cdot \mathsf{S}^{\mathsf{T}}) \sqcap \mathsf{S}^{\mathsf{T}} \sqsubseteq 1 \sqcap \mathsf{S}^{\mathsf{T}} = \mathsf{S} \sqcap 1$ using injectivity of S and (2.2). Hence $(\mathsf{S} \sqcap 1) \cdot \mathsf{S}^{\mathsf{T}*} \sqsubseteq \mathsf{S} \sqcap 1$ by Kleene star induction. Thus

$$\mathsf{S} \sqcap 1 \sqsubseteq (\mathsf{S} \sqcap 1) \cdot \top = (\mathsf{S} \sqcap 1) \cdot \mathsf{S}^{\mathsf{T}*} \cdot \mathsf{Z} \sqsubseteq (\mathsf{S} \sqcap 1) \cdot \mathsf{Z} \sqsubseteq 1 \cdot \mathsf{Z} = \mathsf{Z}$$

using Peano axiom $\mathsf{S}^{\mathsf{T}*} \cdot \mathsf{Z} = \top$. Theorem 6.5 immediately follows from this Peano axiom by unfolding the Kleene star. □

Proof of Theorem 7.3:
With $\mathsf{M} = \mathsf{S} \cdot \mathsf{Z}$ we have

$$\top = \mathsf{S}^{\mathsf{T}*} \cdot \mathsf{Z} = ((\mathsf{S} - \mathsf{M}) \sqcup (\mathsf{S} \sqcap \mathsf{M}))^{\mathsf{T}*} \cdot \mathsf{Z} = (\mathsf{S}' \sqcup (\mathsf{S} \sqcap \mathsf{M}))^{\mathsf{T}*} \cdot \mathsf{Z}$$

using Theorem 9.1. If $\mathsf{M} = \bot$, then $(\mathsf{S}' \sqcup (\mathsf{S} \sqcap \mathsf{M}))^{\mathsf{T}*} \cdot \mathsf{Z} = \mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z}$ and we are done. Otherwise M is a point by Theorem 8.3 and hence $\mathsf{S} \sqcap \mathsf{M}$ is an arc using (9). Moreover $\mathsf{S} - \mathsf{Z}^{\mathsf{T}} = \mathsf{S}' = \mathsf{S} - \mathsf{M} \sqsubseteq \overline{\mathsf{M}}$ by Theorem 9.1 and therefore $\mathsf{S} \sqcap \mathsf{M} \sqsubseteq \mathsf{Z}^{\mathsf{T}}$ using (11.1). Hence

$$(\mathsf{S}' \sqcup (\mathsf{S} \sqcap \mathsf{M}))^{\mathsf{T}*} \cdot \mathsf{Z} = (\mathsf{S}'^{\mathsf{T}*} \cdot (\mathsf{S} \sqcap \mathsf{M})^{\mathsf{T}} \cdot \mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z}) \sqcup (\mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z}) \sqsubseteq (\mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z} \cdot \top) \sqcup (\mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z})$$
$$= \mathsf{S}'^{\mathsf{T}*} \cdot \mathsf{Z}$$

using [12, Proposition 3] (see proof of Theorem 12.1 in this appendix) and that Z is a vector.   □

Proof of Theorem 8:

For the forward implication of Theorem 8.1 assume M is a point. Hence $1 \sqsubseteq \mathsf{Z}^{\mathsf{T}} \cdot \mathsf{S}^{\mathsf{T}} \cdot \mathsf{S} \cdot \mathsf{Z}$, which implies $\mathsf{Z} \sqsubseteq \mathsf{S}^{\mathsf{T}} \cdot \mathsf{S} \cdot \mathsf{Z} \sqsubseteq \mathsf{S}^{\mathsf{T}} \cdot \top$ using (6.3). It follows that $(\mathsf{S}^{\mathsf{T}} \cdot \mathsf{S}^{\mathsf{T}} \cdot \top) \sqcup \mathsf{Z} \sqsubseteq \mathsf{S}^{\mathsf{T}} \cdot \top$. Therefore $\top = \mathsf{S}^{\mathsf{T}*} \cdot \mathsf{Z} \sqsubseteq \mathsf{S}^{\mathsf{T}} \cdot \top$ by a Peano axiom and Kleene star induction. Thus S is surjective using (5).

For the backward implication assume S is surjective. Hence $\mathsf{M} = \mathsf{S} \cdot \mathsf{Z}$ is a point: it is bijective using (8.1) since both S and Z are bijective, and it is a vector using (1.4) since Z is a vector.

For the forward implication of Theorem 8.2 assume that $\mathsf{M} \neq \bot$. Hence $\top \cdot \mathsf{S} \cdot \mathsf{Z} = \top$ using the Tarski rule since Z is a vector. Therefore $\mathsf{Z}^{\mathsf{T}} \sqsubseteq \top \cdot \mathsf{S}$ using (6.2) since Z is a point. Thus $\top = (\top \cdot \mathsf{S}) \sqcup \mathsf{Z}^{\mathsf{T}} = \top \cdot \mathsf{S}$ using Theorem 6.5, whence S is surjective using (5).

For the backward implication assume S is surjective. Hence M is a point by part 1 and therefore $\mathsf{M} \neq \bot$.

Theorem 8.3 follows immediately from parts 1 and 2.   □

Proof of Theorem 9.1:

We obtain

$$\mathsf{S}' = \mathsf{S} - \mathsf{Z}^{\mathsf{T}} = \mathsf{S} \cdot (1 - \mathsf{Z}^{\mathsf{T}}) = \mathsf{S} \cdot (1 - \mathsf{Z}) = (\mathsf{S} \cdot 1) \sqcap (\mathsf{S} \cdot \overline{\mathsf{Z}}) = \mathsf{S} \sqcap \overline{\mathsf{S} \cdot \mathsf{Z}} = \mathsf{S} - \mathsf{M}$$

using (1.4) and (1.3), (2.2), (7) and (8.2).   □

Proof of Theorems 10.1 and 10.4:

For Theorem 10.1, since w and y are points, $y \sqsubseteq p^*[w]$ is equivalent to $w \sqcap y^{\mathsf{T}} = w \cdot y^{\mathsf{T}} \sqsubseteq p^*$ using (1.5), (6.1) and (6.2). Hence

$$w \sqcap y^{\mathsf{T}} - 1 \sqsubseteq p^* - 1 = (p-1)^* - 1 = (p-1)^+ - 1 \sqsubseteq (p-1)^+$$

Thus

$$p[w \mapsto y] - 1 = (w \sqcap y^{\mathsf{T}} - 1) \sqcup (\overline{w} \sqcap p - 1) \sqsubseteq (p-1)^+ \sqcup (p-1) = (p-1)^+$$

Therefore $(p[w \mapsto y] - 1)^+ \sqsubseteq (p-1)^{++} = (p-1)^+ \sqsubseteq \overline{1}$ using (12.3).

For Theorem 10.4, we first show that $x \sqsubseteq y$ or $x \sqsubseteq \overline{y}$ for any point x and any vector y. Assume $x \sqsubseteq \overline{y}$ does not hold, then $x \sqcap y \neq \bot$ using (11.1), hence $\top \cdot (x \sqcap y) = \top \cdot (x \sqcap y) \cdot \top = \top$ using (1.4) and the Tarski rule. Hence $x \sqcap y = x \sqcap (\top \cdot (x \sqcap y)) = x \sqcap \top = x$ using (4) since $x \sqcap y \sqsubseteq x$ and x is injective. Thus $x \sqsubseteq y$.

We apply this both ways to points x and y. If $x \neq y$, we cannot get both $x \sqsubseteq y$ and $y \sqsubseteq x$, so $x \sqsubseteq \overline{y}$ or $y \sqsubseteq \overline{x}$ must hold. Either is equivalent to $x \sqcap y = \bot$ using (11.1).   □

Proof of Theorems 11.3, 11.7, 11.8 and 11.9:
For Theorem 11.3 observe that

$$x \cdot x = ((x \sqcap 1) \cdot x) \sqcup ((x - 1) \cdot x) \sqsubseteq x \sqcup ((x - 1) \cdot x) \sqsubseteq 1 \sqcup (x - 1) \sqcup ((x - 1) \cdot x)$$
$$= ((x - 1) \cdot (x \sqcup 1)) \sqcup 1$$

which implies

$$(x \cdot x) - 1 \sqsubseteq (x - 1) \cdot (x \sqcup 1) = ((x - 1) \cdot (x - 1)) \sqcup (x - 1) \sqsubseteq (x - 1)^+$$

using (11.1). Theorem 11.7 follows from Theorem 2.13 since transposition is involutive. As a consequence we obtain Theorem 11.8 since

$$(x[y \mapsto (x \cdot x)^\mathsf{T}] - 1)^+ \sqsubseteq ((x \sqcup (x \cdot x)) - 1)^+ = ((x - 1) \sqcup ((x \cdot x) - 1))^+ \sqsubseteq (x - 1)^{++}$$
$$= (x - 1)^+ \sqsubseteq \overline{1}$$

using Theorem 11.3 and (12.3). For Theorem 11.9 it remains to show that $x[y \mapsto (x \cdot x)^\mathsf{T}]$ is a mapping, which follows by Theorem 1.3 since $(x \cdot x)^\mathsf{T}$ is bijective using (8.1). □

Proof of Theorems 12.1 and 12.2:
For Theorem 12.1 assume y is an arc. We first show $(x \sqcup y)^* = x^* \sqcup (x^* \cdot y \cdot x^*)$ [12, Proposition 3]. Note that

$$(x \sqcup y) \cdot (x^* \sqcup (x^* \cdot y \cdot x^*)) = x^+ \sqcup (x^+ \cdot y \cdot x^*) \sqcup (y \cdot x^*) \sqcup (y \cdot x^* \cdot y \cdot x^*)$$
$$= x^+ \sqcup (x^+ \cdot y \cdot x^*) \sqcup (y \cdot x^*)$$
$$= x^+ \sqcup (x^* \cdot y \cdot x^*)$$

using (10). This implies $(x \sqcup y) \cdot (x^* \sqcup (x^* \cdot y \cdot x^*)) \sqsubseteq x^* \sqcup (x^* \cdot y \cdot x^*)$, whence $(x \sqcup y)^* \sqsubseteq x^* \sqcup (x^* \cdot y \cdot x^*)$ by Kleene star induction. The opposite inequality follows using (12.1). Hence

$$(x \sqcup y)^+ = (x \sqcup y) \cdot (x \sqcup y)^* = x^+ \sqcup (x^* \cdot y \cdot x^*)$$

For Theorem 12.2 note first that assumption $y \sqcap (p^* \cdot w) = \bot$ is equivalent to $y^\mathsf{T} \cdot p^* \cdot w \sqsubseteq \overline{1}$ using (11.1) and (11.2). The latter is equivalent to $p^* \cdot w \cdot y^\mathsf{T} \cdot p^* \sqsubseteq \overline{1}$ using (12.1) and (11.3). Second,

$$p[w \mapsto y] - 1 \sqsubseteq (w \sqcap y^\mathsf{T}) \sqcup (p - 1) = (w \cdot y^\mathsf{T}) \sqcup (p - 1)$$

using (1.5). Hence

$$(p[w \mapsto y] - 1)^+ \sqsubseteq ((w \cdot y^\mathsf{T}) \sqcup (p - 1))^+$$
$$= (p - 1)^+ \sqcup ((p - 1)^* \cdot w \cdot y^\mathsf{T} \cdot (p - 1)^*)$$
$$\sqsubseteq (p - 1)^+ \sqcup (p^* \cdot w \cdot y^\mathsf{T} \cdot p^*)$$
$$\sqsubseteq \overline{1}$$

by Theorem 12.1 since $w \cdot y^\mathsf{T}$ is an arc using (1.5) and (9). □

References

[1] Guttmann W. Verifying the Correctness of Disjoint-Set Forests with Kleene Relation Algebras. In: Fahrenberg U, Jipsen P, Winter M (eds.), Relational and Algebraic Methods in Computer Science (RAMiCS 2020), volume 12062 of *Lecture Notes in Computer Science*. Springer, 2020 pp. 134–151. doi:10.1007/978-3-030-43520-2_9.

[2] Back RJ, von Wright J. Refinement Calculus. Springer, New York, 1998. doi:10.1007/978-1-4612-1674-2.

[3] Hoare CAR, He J. Unifying theories of programming. Prentice Hall Europe, 1998. ISBN-10:0134587618, 13:978-0134587615.

[4] Maddux RD. Relation-algebraic semantics. *Theoretical Computer Science*, 1996. **160**(1-2):1–85. doi:10.1016/0304-3975(95)00082-8.

[5] Schmidt G, Ströhlein T. Relationen und Graphen. Springer, 1989. doi:10.1007/978-3-642-83608-4.

[6] Back RJR, von Wright J. Reasoning algebraically about loops. *Acta Informatica*, 1999. **36**(4):295–334. doi:10.1007/s002360050163.

[7] Kozen D. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 1997. **19**(3):427–443. doi:10.1145/256167.256195.

[8] Kozen D. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 1994. **110**(2):366–390. doi:10.1006/inco.1994.1037.

[9] Backhouse RC, Carré BA. Regular Algebra Applied to Path-finding Problems. *Journal of the Institute of Mathematics and its Applications*, 1975. **15**(2):161–186. doi:10.1093/imamat/15.2.161.

[10] Berghammer R. Combining relational calculus and the Dijkstra–Gries method for deriving relational programs. *Information Sciences*, 1999. **119**(3-4):155–171. doi:10.1016/S0020-0255(99)00012-2.

[11] Berghammer R, von Karger B, Wolf A. Relation-Algebraic Derivation of Spanning Tree Algorithms. In: Jeuring J (ed.), Mathematics of Program Construction (MPC 1998), volume 1422 of *Lecture Notes in Computer Science*. Springer, 1998 pp. 23–43. doi:10.1007/BFb0054283.

[12] Berghammer R, Struth G. On Automated Program Construction and Verification. In: Bolduc C, Desharnais J, Ktari B (eds.), Mathematics of Program Construction (MPC 2010), volume 6120 of *Lecture Notes in Computer Science*. Springer, 2010 pp. 22–41. doi:10.1007/978-3-642-13321-3_4.

[13] Gondran M, Minoux M. Graphs, Dioids and Semirings. Springer, 2008. doi:10.1007/978-0-387-75450-5.

[14] Höfner P, Möller B. Dijkstra, Floyd and Warshall meet Kleene. *Formal Aspects of Computing*, 2012. **24**(4):459–476. doi:10.1007/s00165-012-0245-4.

[15] Möller B. Derivation of Graph and Pointer Algorithms. In: Möller B, Partsch HA, Schuman SA (eds.), Formal Program Development, volume 755 of *Lecture Notes in Computer Science*. Springer, 1993 pp. 123–160. doi:10.1007/3-540-57499-9_19.

[16] Bird R, de Moor O. Algebra of Programming. Prentice Hall, 1997. ISBN-10:013507245X, 13:978-0135072455.

[17] Hoare CAR. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 1969. **12**(10):576–580/583. doi:10.1145/363235.363259.

[18] Cormen TH, Leiserson CE, Rivest RL. Introduction to Algorithms. MIT Press, 1990. ISBN-10:0262031418, 13:978-0262031417.

[19] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL: A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

[20] Guttmann W. Relational Disjoint Set Forests. *Archive of Formal Proofs*, 2020. First version 2020–08–26, updated 2021–06–19 and 2022–09–19 and 2023–08–10 and 2024–01–10, URL https://www.isa-afp.org/entries/Relational_Disjoint_Set_Forests.html.

[21] Paulson LC, Blanchette JC. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In: Sutcliffe G, Ternovska E, Schulz S (eds.), Proceedings of the 8th International Workshop on the Implementation of Logics. 2010 pp. 3–13. doi:10.29007/36dt.

[22] Mu SC. Maximum Segment Sum is Back. In: Glück R, de Moor O (eds.), Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2008). ACM, 2008 pp. 31–39. doi:10.1145/1328408.1328414.

[23] Guttmann W. An Algebraic Framework for Minimum Spanning Tree Problems. *Theoretical Computer Science*, 2018. **744**:37–55. doi:10.1016/j.tcs.2018.04.012.

[24] Tarski A. On the calculus of relations. *The Journal of Symbolic Logic*, 1941. **6**(3):73–89. doi:10.2307/2268577.

[25] Möller B. Towards pointer algebra. *Science of Computer Programming*, 1993. **21**(1):57–90. doi:10.1016/0167-6423(93)90008-D.

[26] Foster JN, Greenwald MB, Moore JT, Pierce BC, Schmitt A. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems*, 2007. **29**(3:17):1–65. doi:10.1145/1232420.1232424.

[27] Galler BA, Fisher MJ. An Improved Equivalence Algorithm. *Communications of the ACM*, 1964. **7**(5):301–303. doi:10.1145/364099.364331.

[28] Derrick J, Boiten E. Refinement: Semantics, Languages and Applications. Springer, 2018. doi:10.1007/978-3-319-92711-4.

[29] Jones CB. Software Development: A Rigorous Approach. Prentice Hall, 1980. ISBN-10:0138218846, 13:978-0138218843.

[30] Partsch HA. Specification and Transformation of Programs: A Formal Approach to Software Development. Springer, 1990. doi:10.1007/978-3-642-61512-2.

[31] Nipkow T. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 1998. **10**(2):171–186. doi:10.1007/s001650050009.

[32] Nipkow T. Hoare Logics in Isabelle/HOL. In: Schwichtenberg H, Steinbrüggen R (eds.), Proof and System-Reliability. Kluwer Academic Publishers, 2002 pp. 341–367. doi:10.1007/978-94-010-0413-8_11.

[33] Guttmann W. Verifying Minimum Spanning Tree Algorithms with Stone Relation Algebras. *Journal of Logical and Algebraic Methods in Programming*, 2018. **101**:132–150. doi:10.1016/j.jlamp.2018.09.005.

[34] Morgan C. Programming from specifications. Prentice Hall, second edition, 1994. ISBN-10: 0131232746, 13:978-0131232747.

[35] van Leeuwen J, van der Weide T. Alternative Path Compression Techniques. Technical Report RUU-CS-77-3, University of Utrecht, 1977. ISSN:0924-3275.

[36] Tarjan RE, van Leeuwen J. Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*, 1984. **31**(2):245–281. doi:10.1145/62.2160.

[37] Lammich P, Meis R. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs*, 2012. URL https://www.isa-afp.org/entries/Separation_Logic_Imperative_HOL.html.

[38] Berghammer R, Zierer H. Relational Algebraic Semantics of Deterministic and Nondeterministic Programs. *Theoretical Computer Science*, 1986. **43**:123–147. doi:10.1016/0304-3975(86)90172-6.

[39] Berghammer R. A Relation-Algebraic Treatment of the Dedekind Recursion Theorem. In: Fahrenberg U, Jipsen P, Winter M (eds.), Relational and Algebraic Methods in Computer Science (RAMiCS 2020), volume 12062 of *Lecture Notes in Computer Science*. Springer, 2020 pp. 15–30. doi:10.1007/978-3-030-43520-2_2.

[40] Kawahara Y. On the Cardinality of Relations. In: Schmidt RA (ed.), Relations and Kleene Algebra in Computer Science (RelMiCS/AKA 2006), volume 4136 of *Lecture Notes in Computer Science*. Springer, 2006 pp. 251–265. doi:10.1007/11828563_17.

[41] McCarthy J. Towards a Mathematical Science of Computation. In: Popplewell CM (ed.), Information Processing (IFIP 1962), volume 2 of *IFIP congress series*. North-Holland Publishing Company, 1963 pp. 21–28.

[42] Hoare CAR. Notes on Data Structuring. In: Dahl OJ, Dijkstra EW, Hoare CAR (eds.), Structured Programming, chapter 2, pp. 83–174. Academic Press, 1972.

[43] Reynolds JC. Reasoning About Arrays. *Communications of the ACM*, 1979. **22**(5):290–299. doi:10.1145/359104.359110.

[44] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall, 1989. ISBN-10:013983768X, 13:978-0139837685.

[45] Hayes I. Specification Case Studies. Technical Report PRG-46, Programming Research Group, University of Oxford, 1985.

[46] Ehm T. Pointer Kleene Algebra. In: Berghammer R, Möller B, Struth G (eds.), Relational and Kleene-Algebraic Methods in Computer Science (RelMiCS 2003), volume 3051 of *Lecture Notes in Computer Science*. Springer, 2004 pp. 99–111. doi:10.1007/978-3-540-24771-5_9.

[47] Möller B. Calculating With Pointer Structures. In: Bird R, Meertens LGLT (eds.), Algorithmic Languages and Calculi, volume 95 of *IFIP Conference Proceedings*. Chapman and Hall, 1997 pp. 24–48. doi:10.1007/978-0-387-35264-0_2.

[48] Möller B. Calculating with acyclic and cyclic lists. *Information Sciences*, 1999. **119**(3-4):135–154. doi:10.1016/S0020-0255(99)00011-0.

[49] Conchon S, Filliâtre JC. A Persistent Union-Find Data Structure. In: Dreyer D, Russo C (eds.), Proceedings of the 2007 Workshop on ML (ML 2007). ACM, 2007 pp. 37–45. doi:10.1145/1292535.1292541.

[50] 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*, 2019. **62**(3):331–365. doi:10.1007/s10817-017-9431-7.

[51] Jones CB. Systematic Software Development Using VDM. Prentice Hall, second edition, 1990. ISBN-10:0138807337, 13:978-0138807337.

[52] Zhan B, Haslbeck MPL. Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle. In: Galmiche D, Schulz S, Sebastiani R (eds.), Automated Reasoning (IJCAR 2018), volume 10900 of *Lecture Notes in Computer Science*. Springer, 2018 pp. 532–548. doi:10.1007/978-3-319-94205-6_35.

[53] Haslbeck MPL, Lammich P. For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM. *ACM Transactions on Programming Languages and Systems*, 2022. **44**(3:14):1–36. doi:10.1145/3486169.

[54] Armstrong A, Gomes VBF, Struth G. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 2016. **28**(2):265–293. doi:10.1007/s00165-015-0343-1.

[55] Schmidt G. Relational Mathematics. Cambridge University Press, 2011. ISBN-10:0521762685, 13:978-0521762687. doi:10.1017/CBO9780511778810.

[56] Haslbeck MPL, Lammich P, Biendarra J. Kruskal's Algorithm for Minimum Spanning Forest. *Archive of Formal Proofs*, 2019. URL https://www.isa-afp.org/entries/Kruskal.html.