Relational Disjoint-Set Forests

Walter Guttmann

March 17, 2025

Abstract

We give a simple relation-algebraic semantics of read and write operations on associative arrays. The array operations seamlessly integrate with assignments in the Hoare-logic library. Using relation algebras and Kleene algebras 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.

Contents

1 Overview

Relation algebras and Kleene algebras have previously been used to reason about graphs and graph algorithms [2, 3, 4, 5, 9, 13, 16]. The operations of these algebras manipulate entire graphs, which is useful for specification but not directly intended for implementation. Low-level array access is a key ingredient for efficient algorithms [6]. We give a relation-algebraic semantics for such read/write access to associative arrays. This allows us to extend relation-algebraic verification methods to a lower level of more efficient implementations.

In this theory we focus on arrays with the same index and value sets, which can be modelled as homogeneous relations and therefore as elements of relation algebras and Kleene algebras [14, 18]. We implement and verify the correctness of disjoint-set forests with path compression strategies and union-by-rank [6, 8, 17].

In order to prepare this theory for future applications with weighted graphs, the verification uses Stone relation algebras, which have weaker axioms than relation algebras [10].

Section 2 contains the simple relation-algebraic semantics of associative array read and write and basic properties of these access operations. In Section 3 we give a Kleene-relation-algebraic semantics of disjoint-set forests. The make-set operation, find-set with path compression and the naive union-sets operation are implemented and verified in Section 4. Section 5 presents further results on disjoint-set forests and relational array access. The initialisation of disjoint-set forests, path halving and path splitting are implemented and verified in Section 6. In Section 7 we study relational Peano structures and implement and verify union-by-rank. Section 8 instantiates the Peano axioms by Boolean matrices.

This Isabelle/HOL theory formally verifies results in [11] and an extended version of that paper [12]. Theorem numbers from the extended version of the paper are mentioned in the theories for reference. See the paper for further details and related work.

Several Isabelle/HOL theories are related to disjoint sets. The theory `HOL/Library/Disjoint_Sets.thy` contains results about partitions and sets of disjoint sets and does not consider their implementation. An implementation of disjoint-set forests with path compression and a size-based heuristic in the Imperative/HOL framework is verified in Archive of Formal Proofs entry [15]. Improved automation of this proof is considered in Archive of Formal Proofs entry [19]. These approaches are based on logical specifications whereas the present theory uses relation algebras and Kleene algebras.

theory *Disjoint-Set-Forests*

imports
HOL−Hoare.Hoare-Logic
Stone-Kleene-Relation-Algebras.Kleene-Relation-Algebras
begin

no-notation *minus* (**infixl** ‹−› *65*)
unbundle *no trancl-syntax*

context *p-algebra*
begin

abbreviation *minus* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹−› *65*)
where $x - y \equiv x \sqcap -y$

end

An arc in a Stone relation algebra corresponds to an atom in a relation algebra and represents a single edge in a graph. A point represents a set of nodes. A rectangle represents the Cartesian product of two sets of nodes [4].

context *times-top*
begin

abbreviation *rectangle* :: $'a \Rightarrow bool$
where *rectangle* $x \equiv x * top * x = x$

end

context *stone-relation-algebra*
begin

lemma *arc-rectangle*:
arc $x \Longrightarrow$ *rectangle* x
⟨*proof*⟩

2 Relation-Algebraic Semantics of Associative Array Access

The following two operations model updating array x at index y to value z, and reading the content of array x at index y, respectively. The read operation uses double brackets to avoid ambiguity with list syntax. The remainder of this section shows basic properties of these operations.

abbreviation *rel-update* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹(-[-⟼-])› [*70*, *65*, *65*] *61*)

where $x[y{\longmapsto}z] \equiv (y \sqcap z^T) \sqcup (-y \sqcap x)$

abbreviation *rel-access* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹(*2*-[[-]])› [*70*, *65*] *65*)
where $x[[y]] \equiv x^T * y$

lemma *update-univalent*:
assumes *univalent* x
and *vector* y
and *injective* z
shows *univalent* $(x[y{\longmapsto}z])$
⟨*proof*⟩

lemma *update-total*:
assumes *total* x
and *vector* y
and *regular* y
and *surjective* z
shows *total* $(x[y{\longmapsto}z])$
⟨*proof*⟩

lemma *update-mapping*:
assumes *mapping* x
and *vector* y
and *regular* y
and *bijective* z
shows *mapping* $(x[y{\longmapsto}z])$
⟨*proof*⟩

lemma *read-injective*:
assumes *injective* y
and *univalent* x
shows *injective* $(x[[y]])$
⟨*proof*⟩

lemma *read-surjective*:
assumes *surjective* y
and *total* x
shows *surjective* $(x[[y]])$
⟨*proof*⟩

lemma *read-bijective*:
assumes *bijective* y
and *mapping* x
shows *bijective* $(x[[y]])$
⟨*proof*⟩

lemma *read-point*:
assumes *point* p
and *mapping* x

shows *point* $(x[[p]])$
⟨*proof*⟩

lemma *update-postcondition*:
assumes *point* x *point* y
shows $x \sqcap p = x * y^T \longleftrightarrow p[[x]] = y$
⟨*proof*⟩

Back and von Wright's array independence requirements [1], later also lens laws [7]

lemma *put-get-sub*:
assumes *vector* y *surjective* u *vector* z $u \leq y$
shows $(x[y\longmapsto z])[[u]] = z$
⟨*proof*⟩

lemma *put-get*:
assumes *vector* y *surjective* y *vector* z
shows $(x[y\longmapsto z])[[y]] = z$
⟨*proof*⟩

lemma *put-put*:
$(x[y\longmapsto z])[y\longmapsto w] = x[y\longmapsto w]$
⟨*proof*⟩

lemma *get-put*:
assumes *point* y
shows $x[y\longmapsto x[[y]]] = x$
⟨*proof*⟩

lemma *update-inf*:
$u \leq y \Longrightarrow (x[y\longmapsto z]) \sqcap u = z^T \sqcap u$
⟨*proof*⟩

lemma *update-inf-same*:
$(x[y\longmapsto z]) \sqcap y = z^T \sqcap y$
⟨*proof*⟩

lemma *update-inf-different*:
$u \leq -y \Longrightarrow (x[y\longmapsto z]) \sqcap u = x \sqcap u$
⟨*proof*⟩

end

3 Relation-Algebraic Semantics of Disjoint-Set Forests

A disjoint-set forest represents a partition of a set into equivalence classes. We take the represented equivalence relation as the semantics of a forest.

It is obtained by operation *fc* below. Additionally, operation *wcc* giving the weakly connected components of a graph will be used for the semantics of the union of two disjoint sets. Finally, operation *root* yields the root of a component tree, that is, the representative of a set containing a given element. This section defines these operations and derives their properties.

context *stone-kleene-relation-algebra*
begin

lemma *omit-redundant-points*:
 assumes *point* p
 shows $p \sqcap x^\star = (p \sqcap 1) \sqcup (p \sqcap x) * (-p \sqcap x)^\star$
⟨*proof*⟩

Weakly connected components

abbreviation $wcc\ x \equiv (x \sqcup x^T)^\star$

lemma *wcc-equivalence*:
 equivalence $(wcc\ x)$
 ⟨*proof*⟩

lemma *wcc-increasing*:
 $x \le wcc\ x$
 ⟨*proof*⟩

lemma *wcc-isotone*:
 $x \le y \Longrightarrow wcc\ x \le wcc\ y$
 ⟨*proof*⟩

lemma *wcc-idempotent*:
 $wcc\ (wcc\ x) = wcc\ x$
 ⟨*proof*⟩

lemma *wcc-below-wcc*:
 $x \le wcc\ y \Longrightarrow wcc\ x \le wcc\ y$
 ⟨*proof*⟩

lemma *wcc-galois*:
 $x \le wcc\ y \longleftrightarrow wcc\ x \le wcc\ y$
 ⟨*proof*⟩

lemma *wcc-bot*:
 $wcc\ bot = 1$
 ⟨*proof*⟩

lemma *wcc-one*:
 $wcc\ 1 = 1$
 ⟨*proof*⟩

lemma *wcc-top*:

$wcc\ top = top$
⟨*proof*⟩

lemma *wcc-with-loops*:
$wcc\ x = wcc\ (x \sqcup 1)$
⟨*proof*⟩

lemma *wcc-without-loops*:
$wcc\ x = wcc\ (x - 1)$
⟨*proof*⟩

lemma *forest-components-wcc*:
$injective\ x \Longrightarrow wcc\ x = forest\text{-}components\ x$
⟨*proof*⟩

lemma *wcc-sup-wcc*:
$wcc\ (x \sqcup y) = wcc\ (x \sqcup wcc\ y)$
⟨*proof*⟩

Components of a forest, which is represented using edges directed towards the roots

abbreviation $fc\ x \equiv x^\star * x^{T\star}$

lemma *fc-equivalence*:
$univalent\ x \Longrightarrow equivalence\ (fc\ x)$
⟨*proof*⟩

lemma *fc-increasing*:
$x \leq fc\ x$
⟨*proof*⟩

lemma *fc-isotone*:
$x \leq y \Longrightarrow fc\ x \leq fc\ y$
⟨*proof*⟩

lemma *fc-idempotent*:
$univalent\ x \Longrightarrow fc\ (fc\ x) = fc\ x$
⟨*proof*⟩

lemma *fc-star*:
$univalent\ x \Longrightarrow (fc\ x)^\star = fc\ x$
⟨*proof*⟩

lemma *fc-plus*:
$univalent\ x \Longrightarrow (fc\ x)^+ = fc\ x$
⟨*proof*⟩

lemma *fc-bot*:
$fc\ bot = 1$

$\langle proof \rangle$

lemma *fc-one*:
  $fc\ 1 = 1$
  $\langle proof \rangle$

lemma *fc-top*:
  $fc\ top = top$
  $\langle proof \rangle$

lemma *fc-wcc*:
  $univalent\ x \Longrightarrow wcc\ x = fc\ x$
  $\langle proof \rangle$

lemma *fc-via-root*:
  assumes $total\ (p^\star * (p \sqcap 1))$
  shows $fc\ p = p^\star * (p \sqcap 1) * p^{T\star}$
$\langle proof \rangle$

lemma *update-acyclic-1*:
  assumes $acyclic\ (p - 1)$
    and $point\ y$
    and $vector\ w$
    and $w \le p^\star * y$
  shows $acyclic\ ((p[w \longmapsto y]) - 1)$
$\langle proof \rangle$

lemma *update-acyclic-2*:
  assumes $acyclic\ (p - 1)$
    and $point\ y$
    and $point\ x$
    and $y \le p^{T\star} * x$
    and $univalent\ p$
    and $p^T * y \le y$
  shows $acyclic\ ((p[p^{T\star}*x \longmapsto y]) - 1)$
$\langle proof \rangle$

lemma *update-acyclic-3*:
  assumes $acyclic\ (p - 1)$
    and $point\ y$
    and $point\ w$
    and $y \le p^{T\star} * w$
  shows $acyclic\ ((p[w \longmapsto y]) - 1)$
  $\langle proof \rangle$

lemma *rectangle-star-rectangle*:
  $rectangle\ a \Longrightarrow a * x^\star * a \le a$
  $\langle proof \rangle$

lemma *arc-star-arc*:
$arc\ a \Longrightarrow a * x^\star * a \leq a$
⟨*proof*⟩

lemma *star-rectangle-decompose*:
assumes $rectangle\ a$
shows $(a \sqcup x)^\star = x^\star \sqcup x^\star * a * x^\star$
⟨*proof*⟩

lemma *star-arc-decompose*:
$arc\ a \Longrightarrow (a \sqcup x)^\star = x^\star \sqcup x^\star * a * x^\star$
⟨*proof*⟩

lemma *plus-rectangle-decompose*:
assumes $rectangle\ a$
shows $(a \sqcup x)^+ = x^+ \sqcup x^\star * a * x^\star$
⟨*proof*⟩

lemma *plus-arc-decompose*:
$arc\ a \Longrightarrow (a \sqcup x)^+ = x^+ \sqcup x^\star * a * x^\star$
⟨*proof*⟩

lemma *update-acyclic-4*:
assumes $acyclic\ (p - 1)$
and $point\ y$
and $point\ w$
and $y \sqcap p^\star * w = bot$
shows $acyclic\ ((p[w \longmapsto y]) - 1)$
⟨*proof*⟩

lemma *update-acyclic-5*:
assumes $acyclic\ (p - 1)$
and $point\ w$
shows $acyclic\ ((p[w \longmapsto w]) - 1)$
⟨*proof*⟩

Root of the tree containing point x in the disjoint-set forest p

abbreviation $roots\ p \equiv (p \sqcap 1) * top$
abbreviation $root\ p\ x \equiv p^{T\star} * x \sqcap roots\ p$

lemma *root-var*:
$root\ p\ x = (p \sqcap 1) * p^{T\star} * x$
⟨*proof*⟩

lemma *root-successor-loop*:
$univalent\ p \Longrightarrow root\ p\ x = p[[root\ p\ x]]$
⟨*proof*⟩

lemma *root-transitive-successor-loop*:

$univalent\ p \Longrightarrow root\ p\ x = p^{T\star} * (root\ p\ x)$
⟨proof⟩

lemma *roots-successor-loop*:
$univalent\ p \Longrightarrow p[[roots\ p]] = roots\ p$
⟨proof⟩

lemma *roots-transitive-successor-loop*:
$univalent\ p \Longrightarrow p^{T\star} * (roots\ p) = roots\ p$
⟨proof⟩

The root of a tree of a node belongs to the same component as the node.

lemma *root-same-component*:
$injective\ x \Longrightarrow root\ p\ x * x^T \le fc\ p$
⟨proof⟩

lemma *root-vector*:
$vector\ x \Longrightarrow vector\ (root\ p\ x)$
⟨proof⟩

lemma *root-vector-inf*:
$vector\ x \Longrightarrow root\ p\ x * x^T = root\ p\ x \sqcap x^T$
⟨proof⟩

lemma *root-same-component-vector*:
$injective\ x \Longrightarrow vector\ x \Longrightarrow root\ p\ x \sqcap x^T \le fc\ p$
⟨proof⟩

lemma *univalent-root-successors*:
assumes $univalent\ p$
shows $(p \sqcap 1) * p^\star = p \sqcap 1$
⟨proof⟩

lemma *same-component-same-root-sub*:
assumes $univalent\ p$
and $bijective\ y$
and $x * y^T \le fc\ p$
shows $root\ p\ x \le root\ p\ y$
⟨proof⟩

lemma *same-component-same-root*:
assumes $univalent\ p$
and $bijective\ x$
and $bijective\ y$
and $x * y^T \le fc\ p$
shows $root\ p\ x = root\ p\ y$
⟨proof⟩

lemma *same-roots-sub*:

assumes *univalent* q
and $p \sqcap 1 \leq q \sqcap 1$
and $fc\ p \leq fc\ q$
shows $p^\star * (p \sqcap 1) \leq q^\star * (q \sqcap 1)$
$\langle proof \rangle$

lemma *same-roots*:
assumes *univalent* p
and *univalent* q
and $p \sqcap 1 = q \sqcap 1$
and $fc\ p = fc\ q$
shows $p^\star * (p \sqcap 1) = q^\star * (q \sqcap 1)$
$\langle proof \rangle$

lemma *same-root*:
assumes *univalent* p
and *univalent* q
and $p \sqcap 1 = q \sqcap 1$
and $fc\ p = fc\ q$
shows $root\ p\ x = root\ q\ x$
$\langle proof \rangle$

lemma *loop-root*:
assumes *injective* x
and $x = p[[x]]$
shows $x = root\ p\ x$
$\langle proof \rangle$

lemma *one-loop*:
assumes *acyclic* $(p - 1)$
and *univalent* p
shows $(p \sqcap 1) * (p^T - 1)^+ * (p \sqcap 1) = bot$
$\langle proof \rangle$

lemma *root-root*:
$root\ p\ x = root\ p\ (root\ p\ x)$
$\langle proof \rangle$

lemma *loop-root-2*:
assumes *acyclic* $(p - 1)$
and *univalent* p
and *injective* x
and $x \leq p^{T+} * x$
shows $x = root\ p\ x$
$\langle proof \rangle$

lemma *path-compression-invariant-simplify*:
assumes *point* w
and $p^{T+} * w \leq -w$

and $w \neq y$
shows $p[[w]] \neq w$
⟨*proof*⟩

end

context *stone-relation-algebra-tarski*
begin

lemma *distinct-points* has been moved to theory *Relation-Algebras* in entry *Stone-Relation-Algebras*

Back and von Wright's array independence requirements [1]

lemma *put-get-different-vector*:
assumes *vector* y $w \leq -y$
shows $(x[y \longmapsto z])[[w]] = x[[w]]$
⟨*proof*⟩

lemma *put-get-different*:
assumes *point* y *point* w $w \neq y$
shows $(x[y \longmapsto z])[[w]] = x[[w]]$
⟨*proof*⟩

lemma *put-put-different-vector*:
assumes *vector* y *vector* v $v \sqcap y = bot$
shows $(x[y \longmapsto z])[v \longmapsto w] = (x[v \longmapsto w])[y \longmapsto z]$
⟨*proof*⟩

lemma *put-put-different*:
assumes *point* y *point* v $v \neq y$
shows $(x[y \longmapsto z])[v \longmapsto w] = (x[v \longmapsto w])[y \longmapsto z]$
⟨*proof*⟩

end

4 Verifying Operations on Disjoint-Set Forests

In this section we verify the make-set, find-set and union-sets operations of disjoint-set forests. We start by introducing syntax for updating arrays in programs. Updating the value at a given array index means updating the whole array.

syntax
-rel-update :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b com* (‹(*2*-[-] :=/ -)› [*70*, *65*, *65*] *61*)

translations
$x[y]$:= z => $(x := (y \sqcap z^T) \sqcup (CONST\ uminus\ y \sqcap x))$

The finiteness requirement in the following class is used for proving that the operations terminate.

class *finite-regular-p-algebra* = *p-algebra* +
  assumes *finite-regular*: *finite* { *x* . *regular x* }
begin

abbreviation *card-down-regular* :: $'a \Rightarrow nat$ (‹-↓› [*100*] *100*)
  where $x\downarrow \equiv$ *card* { *z* . *regular z* $\wedge$ $z \le x$ }

end

class *stone-kleene-relation-algebra-tarski-finite-regular* =
stone-kleene-relation-algebra-tarski + *finite-regular-p-algebra*
begin

4.1 Make-Set

We prove two correctness results about make-set. The first shows that the forest changes only to the extent of making one node the root of a tree. The second result adds that only singleton sets are created.

definition *make-set-postcondition p x p0* $\equiv x \sqcap p = x * x^T \wedge -x \sqcap p = -x \sqcap p0$

theorem *make-set*:
  VARS p
  [*point x* $\wedge$ *p0* = *p*]
  p[*x*] := *x*
  [*make-set-postcondition p x p0*]
  ⟨*proof*⟩

theorem *make-set-2*:
  VARS p
  [*point x* $\wedge$ *p0* = *p* $\wedge$ $p \le 1$]
  p[*x*] := *x*
  [*make-set-postcondition p x p0* $\wedge$ $p \le 1$]
⟨*proof*⟩

The above total-correctness proof allows us to extract a function, which can be used in other implementations below. This is a technique of [10].

lemma *make-set-exists*:
  point x $\Longrightarrow \exists p'$. *make-set-postcondition* p' *x p*
  ⟨*proof*⟩

definition *make-set p x* $\equiv$ (*SOME* p' . *make-set-postcondition* p' *x p*)

lemma *make-set-function*:
  assumes *point x*
    and p' = *make-set p x*
  shows *make-set-postcondition* p' *x p*
⟨*proof*⟩

end

4.2 Find-Set

Disjoint-set forests are represented by their parent mapping. It is a forest except each root of a component tree points to itself.

We prove that find-set returns the root of the component tree of the given node.

context *pd-kleene-allegory*
begin

abbreviation *disjoint-set-forest* $p \equiv$ *mapping* $p \wedge$ *acyclic* $(p - 1)$

end

context *stone-kleene-relation-algebra-tarski*
begin

If two nodes are mutually reachable from each other in a disjoint-set forest, they must be equal.

lemma *forest-mutually-reachable*:
  assumes *acyclic* $(p - 1)$ *point* x *point* y $x \leq p^\star * y$ $y \leq p^\star * x$
  shows $x = y$
⟨*proof*⟩

lemma *forest-mutually-reachable-2*:
  assumes *acyclic* $(p - 1)$ *point* x *point* y $x \leq p^{T\star} * y$ $y \leq p^{T\star} * x$
  shows $x = y$
⟨*proof*⟩

end

context *stone-kleene-relation-algebra-tarski-finite-regular*
begin

definition *find-set-precondition* $p\ x \equiv$ *disjoint-set-forest* $p \wedge$ *point* x
definition *find-set-invariant* $p\ x\ y \equiv$ *find-set-precondition* $p\ x \wedge$ *point* $y \wedge y \leq p^{T\star} * x$
definition *find-set-postcondition* $p\ x\ y \equiv$ *point* $y \wedge y =$ *root* $p\ x$

lemma *find-set-1*:
  find-set-precondition $p\ x \Longrightarrow$ *find-set-invariant* $p\ x\ x$
  ⟨*proof*⟩

lemma *find-set-2*:
  find-set-invariant $p\ x\ y \wedge y \neq p[[y]] \Longrightarrow$ *find-set-invariant* $p\ x\ (p[[y]]) \wedge (p^{T\star} * (p[[y]]))\!\downarrow\ < (p^{T\star} * y)\!\downarrow$
⟨*proof*⟩

lemma *find-set-3*:
  find-set-invariant $p\ x\ y \wedge y = p[[y]] \Longrightarrow$ *find-set-postcondition* $p\ x\ y$

⟨*proof*⟩

theorem *find-set*:
VARS y
[*find-set-precondition p x*]
$y := x$;
WHILE $y \neq p[[y]]$
INV { *find-set-invariant p x y* }
VAR { $(p^{T\star} * y)\downarrow$ }
DO $y := p[[y]]$
OD
[*find-set-postcondition p x y*]
⟨*proof*⟩

lemma *find-set-exists*:
find-set-precondition p x $\Longrightarrow \exists\, y$. *find-set-postcondition p x y*
⟨*proof*⟩

The root of a component tree is a point, that is, represents a singleton set of nodes. This could be proved from the definitions using Kleene-relation algebraic calculations. But they can be avoided because the property directly follows from the postcondition of the previous correctness proof. The corresponding algorithm shows how to obtain the root. We therefore have an essentially constructive proof of the following result.

lemma *root-point*:
disjoint-set-forest p $\Longrightarrow$ *point x* $\Longrightarrow$ *point* (*root p x*)
⟨*proof*⟩

definition *find-set p x* $\equiv$ (*SOME y* . *find-set-postcondition p x y*)

lemma *find-set-function*:
assumes *find-set-precondition p x*
and $y =$ *find-set p x*
shows *find-set-postcondition p x y*
⟨*proof*⟩

4.3 Path Compression

The path-compression technique is frequently implemented in recursive implementations of find-set modifying the tree on the way out from recursive calls. Here we implement it using a second while-loop, which iterates over the same path to the root and changes edges to point to the root of the component, which is known after the while-loop in find-set completes. We prove that path compression preserves the equivalence-relational semantics of the disjoint-set forest and also preserves the roots of the component trees. Additionally we prove the exact effect of path compression.

definition *path-compression-precondition p x y* $\equiv$ *disjoint-set-forest p* $\wedge$ *point x* $\wedge$ *point y* $\wedge$ $y =$ *root p x*

definition *path-compression-invariant* $p\ x\ y\ p0\ w \equiv$
  path-compression-precondition $p\ x\ y \wedge$ *point* $w \wedge$
  $p \sqcap 1 = p0 \sqcap 1 \wedge fc\ p = fc\ p0 \wedge$
  root $p\ w = y \wedge p0[p0^{T\star} * x - p0^{T\star} * w \longmapsto y] = p \wedge$
  disjoint-set-forest $p0 \wedge w \leq p0^{T\star} * x$
definition *path-compression-postcondition* $p\ x\ y\ p0 \equiv$
  disjoint-set-forest $p \wedge y =$ *root* $p\ x \wedge p \sqcap 1 = p0 \sqcap 1 \wedge fc\ p = fc\ p0 \wedge$
  $p0[p0^{T\star} * x \longmapsto y] = p$

We first consider a variant that achieves the effect as a single update. The parents of all nodes reachable from x are simultaneously updated to the root of the component of x.

lemma *path-compression-exact*:
  assumes *path-compression-precondition* $p0\ x\ y$
    and $p0[p0^{T\star} * x \longmapsto y] = p$
  shows $p \sqcap 1 = p0 \sqcap 1\ fc\ p = fc\ p0$
⟨*proof*⟩

lemma *update-acyclic-6*:
  assumes *disjoint-set-forest* p
    and *point* x
  shows *acyclic* $((p[p^{T\star}*x \longmapsto root\ p\ x]) - 1)$
  ⟨*proof*⟩

theorem *path-compression-assign*:
  VARS p
  [*path-compression-precondition* $p\ x\ y \wedge p0 = p$]
  $p[p^{T\star} * x] := y$
  [*path-compression-postcondition* $p\ x\ y\ p0$]
  ⟨*proof*⟩

We next look at implementing these updates using a loop.

lemma *path-compression-1a*:
  assumes *point* x
    and *disjoint-set-forest* p
    and $x \neq$ *root* $p\ x$
  shows $p^{T+} * x \leq -\ x$
  ⟨*proof*⟩

lemma *path-compression-1b*:
  $x \leq p^{T\star} * x$
  ⟨*proof*⟩

lemma *path-compression-1*:
  path-compression-precondition $p\ x\ y \Longrightarrow$ *path-compression-invariant* $p\ x\ y\ p\ x$
  ⟨*proof*⟩

lemma *path-compression-2*:
  path-compression-invariant $p\ x\ y\ p0\ w \wedge y \neq p[[w]] \Longrightarrow$

path-compression-invariant $(p[w \mapsto y])$ *x y p0* $(p[[w]]) \wedge ((p[w \mapsto y])^{T\star} *$ $(p[[w]]))\downarrow < (p^{T\star} * w)\downarrow$
⟨*proof*⟩

lemma *path-compression-3a*:
  assumes *path-compression-invariant p x* $(p[[w]])$ *p0 w*
  shows $p0[p0^{T\star} * x \mapsto p[[w]]] = p$
⟨*proof*⟩

lemma *path-compression-3*:
  path-compression-invariant p x $(p[[w]])$ *p0 w* $\Longrightarrow$ *path-compression-postcondition p x* $(p[[w]])$ *p0*
  ⟨*proof*⟩

theorem *path-compression*:
  VARS p t w
  [*path-compression-precondition p x y* $\wedge$ $p0 = p$]
  $w := x$;
  WHILE $y \neq p[[w]]$
    INV { *path-compression-invariant p x y p0 w* }
    VAR { $(p^{T\star} * w)\downarrow$ }
     DO $t := w$;
        $w := p[[w]]$;
        $p[t] := y$
     OD
  [*path-compression-postcondition p x y p0*]
  ⟨*proof*⟩

lemma *path-compression-exists*:
  path-compression-precondition p x y $\Longrightarrow \exists\, p'$. *path-compression-postcondition* p' *x y p*
  ⟨*proof*⟩

definition *path-compression p x y* $\equiv$ (*SOME* p' . *path-compression-postcondition* p' *x y p*)

lemma *path-compression-function*:
  assumes *path-compression-precondition p x y*
    and $p' =$ *path-compression p x y*
  shows *path-compression-postcondition* p' *x y p*
  ⟨*proof*⟩

4.4 Find-Set with Path Compression

We sequentially combine find-set and path compression. We consider implementations which use the previously derived functions and implementations which unfold their definitions.

theorem *find-set-path-compression*:
  VARS p y

[*find-set-precondition* p x $\wedge$ $p0 = p$]
y := *find-set* p x;
p := *path-compression* p x y
[*path-compression-postcondition* p x y $p0$]
⟨*proof*⟩

theorem *find-set-path-compression-1*:
VARS p t w y
[*find-set-precondition* p x $\wedge$ $p0 = p$]
y := *find-set* p x;
$w := x$;
WHILE $y \neq p[[w]]$
INV { *path-compression-invariant* p x y $p0$ w }
VAR { $(p^{T\star} * w)\downarrow$ }
DO $t := w$;
$w := p[[w]]$;
$p[t] := y$
OD
[*path-compression-postcondition* p x y $p0$]
⟨*proof*⟩

theorem *find-set-path-compression-2*:
VARS p y
[*find-set-precondition* p x $\wedge$ $p0 = p$]
$y := x$;
WHILE $y \neq p[[y]]$
INV { *find-set-invariant* p x y $\wedge$ $p0 = p$ }
VAR { $(p^{T\star} * y)\downarrow$ }
DO $y := p[[y]]$
OD;
p := *path-compression* p x y
[*path-compression-postcondition* p x y $p0$]
⟨*proof*⟩

theorem *find-set-path-compression-3*:
VARS p t w y
[*find-set-precondition* p x $\wedge$ $p0 = p$]
$y := x$;
WHILE $y \neq p[[y]]$
INV { *find-set-invariant* p x y $\wedge$ $p0 = p$ }
VAR { $(p^{T\star} * y)\downarrow$ }
DO $y := p[[y]]$
OD;
$w := x$;
WHILE $y \neq p[[w]]$
INV { *path-compression-invariant* p x y $p0$ w }
VAR { $(p^{T\star} * w)\downarrow$ }
DO $t := w$;
$w := p[[w]]$;

$p[t] := y$
OD
[*path-compression-postcondition p x y p0*]
⟨*proof*⟩

Find-set with path compression returns two results: the representative of the tree and the modified disjoint-set forest.

lemma *find-set-path-compression-exists*:
find-set-precondition p x $\Longrightarrow \exists\, p'\ y$. *path-compression-postcondition* p' *x y p*
⟨*proof*⟩

definition *find-set-path-compression p x* $\equiv$ (*SOME* (p',y) . *path-compression-postcondition* p' *x y p*)

lemma *find-set-path-compression-function*:
assumes *find-set-precondition p x*
and (p',y) = *find-set-path-compression p x*
shows *path-compression-postcondition* p' *x y p*
⟨*proof*⟩

We prove that *find-set-path-compression* returns the same representative as *find-set*.

lemma *find-set-path-compression-find-set*:
assumes *find-set-precondition p x*
shows *find-set p x* = *snd* (*find-set-path-compression p x*)
⟨*proof*⟩

A weaker postcondition suffices to prove that the two forests have the same semantics; that is, they describe the same disjoint sets and have the same roots.

lemma *find-set-path-compression-path-compression-semantics*:
assumes *find-set-precondition p x*
shows *fc* (*path-compression p x* (*find-set p x*)) = *fc* (*fst* (*find-set-path-compression p x*))
and *path-compression p x* (*find-set p x*) $\sqcap$ *1* = *fst* (*find-set-path-compression p x*) $\sqcap$ *1*
⟨*proof*⟩

With the current, stronger postcondition of path compression describing the precise effect of how links change, we can prove that the two forests are actually equal.

lemma *find-set-path-compression-find-set-pathcompression*:
assumes *find-set-precondition p x*
shows *path-compression p x* (*find-set p x*) = *fst* (*find-set-path-compression p x*)
⟨*proof*⟩

4.5 Union-Sets

We only consider a naive union-sets operation (without ranks). The semantics is the equivalence closure obtained after adding the link between the two given nodes, which requires those two elements to be in the same set. The implementation uses temporary variable t to store the two results returned by find-set with path compression. The disjoint-set forest, which keeps being updated, is threaded through the sequence of operations.

definition *union-sets-precondition* $p\ x\ y \equiv$ *disjoint-set-forest* $p \wedge$ *point* $x \wedge$ *point* y

definition *union-sets-postcondition* $p\ x\ y\ p0 \equiv$ *disjoint-set-forest* $p \wedge$ *fc* $p =$ *wcc* $(p0 \sqcup x * y^T)$

lemma *union-sets-1*:
  assumes *union-sets-precondition* $p0\ x\ y$
    and *path-compression-postcondition* $p1\ x\ r\ p0$
    and *path-compression-postcondition* $p2\ y\ s\ p1$
  shows *union-sets-postcondition* $(p2[r \longmapsto s])\ x\ y\ p0$
⟨*proof*⟩

theorem *union-sets*:
  VARS $p\ r\ s\ t$
  [*union-sets-precondition* $p\ x\ y \wedge p0 = p$]
  $t :=$ *find-set-path-compression* $p\ x$;
  $p :=$ *fst* t;
  $r :=$ *snd* t;
  $t :=$ *find-set-path-compression* $p\ y$;
  $p :=$ *fst* t;
  $s :=$ *snd* t;
  $p[r] := s$
  [*union-sets-postcondition* $p\ x\ y\ p0$]
⟨*proof*⟩

lemma *union-sets-exists*:
  union-sets-precondition $p\ x\ y \Longrightarrow \exists p'$. *union-sets-postcondition* $p'\ x\ y\ p$
  ⟨*proof*⟩

definition *union-sets* $p\ x\ y \equiv$ (*SOME* p' . *union-sets-postcondition* $p'\ x\ y\ p$)

lemma *union-sets-function*:
  assumes *union-sets-precondition* $p\ x\ y$
    and $p' =$ *union-sets* $p\ x\ y$
  shows *union-sets-postcondition* $p'\ x\ y\ p$
  ⟨*proof*⟩

theorem *union-sets-2*:
  VARS $p\ r\ s$
  [*union-sets-precondition* $p\ x\ y \wedge p0 = p$]
  $r :=$ *find-set* $p\ x$;
  $p :=$ *path-compression* $p\ x\ r$;

$s := $ *find-set* $p\ y$;
$p := $ *path-compression* $p\ y\ s$;
$p[r] := s$
[*union-sets-postcondition* $p\ x\ y\ p0$]
⟨*proof*⟩

end

end

theory *More-Disjoint-Set-Forests*

imports *Disjoint-Set-Forests*

begin

5 More on Array Access and Disjoint-Set Forests

This section contains further results about directed acyclic graphs and relational array operations.

unbundle *no uminus-syntax*

context *stone-relation-algebra*
begin

lemma *update-square*:
assumes *point* y
shows $x[y\longmapsto x[[x[[y]]]]] \leq x * x \sqcup x$
⟨*proof*⟩

lemma *update-ub*:
$x[y\longmapsto z] \leq x \sqcup z^T$
⟨*proof*⟩

lemma *update-square-ub*:
$x[y\longmapsto (x * x)^T] \leq x \sqcup x * x$
⟨*proof*⟩

lemma *update-same-sub*:
assumes $u \sqcap x = u \sqcap z$
and $y \leq u$
and *regular* y
shows $x[y\longmapsto z^T] = x$
⟨*proof*⟩

lemma *update-point-get*:
point $y \Longrightarrow x[y\longmapsto z[[y]]] = x[y\longmapsto z^T]$

$\langle proof \rangle$

lemma *update-bot*:
$x[bot \longmapsto z] = x$
$\langle proof \rangle$

lemma *update-top*:
$x[top \longmapsto z] = z^T$
$\langle proof \rangle$

lemma *update-same*:
assumes *regular u*
shows $(x[y \longmapsto z])[u \longmapsto z] = x[y \sqcup u \longmapsto z]$
$\langle proof \rangle$

lemma *update-same-3*:
assumes *regular u*
and *regular v*
shows $((x[y \longmapsto z])[u \longmapsto z])[v \longmapsto z] = x[y \sqcup u \sqcup v \longmapsto z]$
$\langle proof \rangle$

lemma *update-split*:
assumes *regular w*
shows $x[y \longmapsto z] = (x[y - w \longmapsto z])[y \sqcap w \longmapsto z]$
$\langle proof \rangle$

lemma *update-injective-swap*:
assumes *injective x*
and *point y*
and *injective z*
and *vector z*
shows *injective* $((x[y \longmapsto x[[z]]])[z \longmapsto x[[y]]])$
$\langle proof \rangle$

lemma *update-injective-swap-2*:
assumes *injective x*
shows *injective* $((x[y \longmapsto x[[bot]]])[bot \longmapsto x[[y]]])$
$\langle proof \rangle$

lemma *update-univalent-swap*:
assumes *univalent x*
and *injective y*
and *vector y*
and *injective z*
and *vector z*
shows *univalent* $((x[y \longmapsto x[[z]]])[z \longmapsto x[[y]]])$
$\langle proof \rangle$

lemma *update-mapping-swap*:

assumes *mapping* x
and *point* y
and *point* z
shows *mapping* $((x[y \longmapsto x[[z]]])[z \longmapsto x[[y]]])$
⟨*proof*⟩

lemma *mapping-inf-point-arc* has been moved to theory *Relation-Algebras* in entry *Stone-Relation-Algebras*

end

context *stone-kleene-relation-algebra*
begin

lemma *omit-redundant-points-2*:
assumes *point* p
shows $p \sqcap x^\star = (p \sqcap 1) \sqcup (p \sqcap x \sqcap -p^T) * (x \sqcap -p^T)^\star$
⟨*proof*⟩

lemma *omit-redundant-points-3*:
assumes *point* p
shows $p \sqcap x^\star = (p \sqcap 1) \sqcup (p \sqcap (x \sqcap -p^T)^+)$
⟨*proof*⟩

lemma *even-odd-root*:
assumes *acyclic* $(x - 1)$
and *regular* x
and *univalent* x
shows $(x * x)^{T\star} \sqcap x^T * (x * x)^{T\star} = (1 \sqcap x) * ((x * x)^{T\star} \sqcap x^T * (x * x)^{T\star})$
⟨*proof*⟩

lemma *update-square-plus*:
point $y \Longrightarrow x[y \longmapsto x[[x[[y]]]]] \leq x^+$
⟨*proof*⟩

lemma *update-square-ub-plus*:
$x[y \longmapsto (x * x)^T] \leq x^+$
⟨*proof*⟩

lemma *acyclic-square*:
assumes *acyclic* $(x - 1)$
shows $x * x \sqcap 1 = x \sqcap 1$
⟨*proof*⟩

lemma *diagonal-update-square-aux*:
assumes *acyclic* $(x - 1)$
and *point* y
shows $1 \sqcap y \sqcap y^T * x * x = 1 \sqcap y \sqcap x$
⟨*proof*⟩

lemma *diagonal-update-square*:
assumes *acyclic* $(x - 1)$
and *point* y
shows $(x[y\longmapsto x[[x[[y]]]]]) \sqcap 1 = x \sqcap 1$
$\langle proof \rangle$

lemma *fc-update-square*:
assumes *mapping* x
and *point* y
shows $fc\ (x[y\longmapsto x[[x[[y]]]]]) = fc\ x$
$\langle proof \rangle$

lemma *acyclic-plus-loop*:
assumes *acyclic* $(x - 1)$
shows $x^+ \sqcap 1 = x \sqcap 1$
$\langle proof \rangle$

lemma *star-irreflexive-part-eq*:
$x^\star - 1 = (x - 1)^+ - 1$
$\langle proof \rangle$

lemma *star-irreflexive-part*:
$x^\star - 1 \leq (x - 1)^+$
$\langle proof \rangle$

lemma *square-irreflexive-part*:
$x * x - 1 \leq (x - 1)^+$
$\langle proof \rangle$

lemma *square-irreflexive-part-2*:
$x * x - 1 \leq x^\star - 1$
$\langle proof \rangle$

lemma *acyclic-update-square*:
assumes *acyclic* $(x - 1)$
shows *acyclic* $((x[y\longmapsto (x * x)^T]) - 1)$
$\langle proof \rangle$

lemma *disjoint-set-forest-update-square*:
assumes *disjoint-set-forest* x
and *vector* y
and *regular* y
shows *disjoint-set-forest* $(x[y\longmapsto (x * x)^T])$
$\langle proof \rangle$

lemma *disjoint-set-forest-update-square-point*:
assumes *disjoint-set-forest* x
and *point* y
shows *disjoint-set-forest* $(x[y\longmapsto (x * x)^T])$

⟨*proof*⟩

end

6 Verifying Further Operations on Disjoint-Set Forests

In this section we verify the init-sets, path-halving and path-splitting operations of disjoint-set forests.

class *choose-point* =
  fixes *choose-point* :: $'a \Rightarrow 'a$

Using the *choose-point* operation we define a simple for-each-loop abstraction as syntactic sugar translated to a while-loop. Regular vector h describes the set of all elements that are yet to be processed. It is made explicit so that the invariant can refer to it.

syntax
  -Foreach :: *idt* ⇒ *idt* ⇒ *'assn* ⇒ *'com* ⇒ *'com* (‹(1FOREACH -/ USING -/ INV {-} //DO - /OD)› [0,0,0,0] 61)
translations *FOREACH x USING h INV* { *i* } *DO c OD* =>
  h := *CONST top*;
   WHILE h ≠ *CONST bot*
    INV { *CONST regular h* ∧ *CONST vector h* ∧ *i* }
    VAR { *h*↓ }
     DO x := *CONST choose-point h*;
       c;
       h[*x*] := *CONST bot*
     OD

class *stone-kleene-relation-algebra-choose-point-finite-regular* =
stone-kleene-relation-algebra + *finite-regular-p-algebra* + *choose-point* +
  assumes *choose-point-point*: *vector x* ⟹ *x* ≠ *bot* ⟹ *point* (*choose-point x*)
  assumes *choose-point-decreasing*: *choose-point x* ≤ −−*x*
begin

subclass *stone-kleene-relation-algebra-tarski-finite-regular*
⟨*proof*⟩

6.1 Init-Sets

A disjoint-set forest is initialised by applying *make-set* to each node. We prove that the resulting disjoint-set forest is the identity relation.

theorem *init-sets*:
  VARS h p x
  [*True*]
  FOREACH x
    USING h

$INV \{ p - h = 1 - h \}$
$DO\ p := make\text{-}set\ p\ x$
OD
$[p = 1 \wedge disjoint\text{-}set\text{-}forest\ p \wedge h = bot]$
⟨*proof*⟩

end

6.2 Path Halving

Path halving is a variant of the path compression technique. Similarly to path compression, we implement path halving independently of find-set, using a second while-loop which iterates over the same path to the root. We prove that path halving preserves the equivalence-relational semantics of the disjoint-set forest and also preserves the roots of the component trees. Additionally we prove the exact effect of path halving, which is to replace every other parent pointer with a pointer to the respective grandparent.

context *stone-kleene-relation-algebra-tarski-finite-regular*
begin

definition *path-halving-invariant p x y p0* $\equiv$
$find\text{-}set\text{-}precondition\ p\ x \wedge point\ y \wedge y \leq p^{T\star} * x \wedge y \leq (p0 * p0)^{T\star} * x \wedge$
$p0[(p0 * p0)^{T\star} * x - p0^{T\star} * y \longmapsto (p0 * p0)^T] = p \wedge$
$disjoint\text{-}set\text{-}forest\ p0$
definition *path-halving-postcondition p x y p0* $\equiv$
$disjoint\text{-}set\text{-}forest\ p \wedge y = root\ p\ x \wedge p \sqcap 1 = p0 \sqcap 1 \wedge fc\ p = fc\ p0 \wedge$
$p0[(p0 * p0)^{T\star} * x \longmapsto (p0 * p0)^T] = p$

lemma *path-halving-invariant-aux-1*:
assumes *point x*
and *point y*
and *disjoint-set-forest p0*
shows $p0 \leq wcc\ (p0[(p0 * p0)^{T\star} * x - p0^{T\star} * y \longmapsto (p0 * p0)^T])$
⟨*proof*⟩

lemma *path-halving-invariant-aux*:
assumes *path-halving-invariant p x y p0*
shows $p[[y]] = p0[[y]]$
and $p[[p[[y]]]] = p0[[p0[[y]]]]$
and $p[[p[[p[[y]]]]]] = p0[[p0[[p0[[y]]]]]]$
and $p \sqcap 1 = p0 \sqcap 1$
and $fc\ p = fc\ p0$
⟨*proof*⟩

lemma *path-halving-1*:
$find\text{-}set\text{-}precondition\ p0\ x \Longrightarrow path\text{-}halving\text{-}invariant\ p0\ x\ x\ p0$
⟨*proof*⟩

lemma *path-halving-2*:
$path\text{-}halving\text{-}invariant\ p\ x\ y\ p0 \wedge y \neq p[[y]] \Longrightarrow path\text{-}halving\text{-}invariant$
$(p[y \longmapsto p[[p[[y]]]]])\ x\ ((p[y \longmapsto p[[p[[y]]]]])[[y]])\ p0 \wedge ((p[y \longmapsto p[[p[[y]]]]])^{T\star} *$
$((p[y \longmapsto p[[p[[y]]]]])[[y]]))\downarrow < (p^{T\star} * y)\downarrow$
$\langle proof \rangle$

lemma *path-halving-3*:
$path\text{-}halving\text{-}invariant\ p\ x\ y\ p0 \wedge y = p[[y]] \Longrightarrow path\text{-}halving\text{-}postcondition\ p\ x\ y$
$p0$
$\langle proof \rangle$

theorem *find-path-halving*:
$VARS\ p\ y$
$[\ find\text{-}set\text{-}precondition\ p\ x \wedge p0 = p\ ]$
$y := x;$
$WHILE\ y \neq p[[y]]$
$INV\ \{\ path\text{-}halving\text{-}invariant\ p\ x\ y\ p0\ \}$
$VAR\ \{\ (p^{T\star} * y)\downarrow\ \}$
$DO\ p[y] := p[[p[[y]]]];$
$y := p[[y]]$
OD
$[\ path\text{-}halving\text{-}postcondition\ p\ x\ y\ p0\ ]$
$\langle proof \rangle$

6.3 Path Splitting

Path splitting is another variant of the path compression technique. We implement it again independently of find-set, using a second while-loop which iterates over the same path to the root. We prove that path splitting preserves the equivalence-relational semantics of the disjoint-set forest and also preserves the roots of the component trees. Additionally we prove the exact effect of path splitting, which is to replace every parent pointer with a pointer to the respective grandparent.

definition *path-splitting-invariant* $p\ x\ y\ p0 \equiv$
$find\text{-}set\text{-}precondition\ p\ x \wedge point\ y \wedge y \leq p0^{T\star} * x \wedge$
$p0[p0^{T\star} * x - p0^{T\star} * y \longmapsto (p0 * p0)^T] = p \wedge$
$disjoint\text{-}set\text{-}forest\ p0$
definition *path-splitting-postcondition* $p\ x\ y\ p0 \equiv$
$disjoint\text{-}set\text{-}forest\ p \wedge y = root\ p\ x \wedge p \sqcap 1 = p0 \sqcap 1 \wedge fc\ p = fc\ p0 \wedge$
$p0[p0^{T\star} * x \longmapsto (p0 * p0)^T] = p$

lemma *path-splitting-invariant-aux-1*:
assumes *point* x
and *point* y
and *disjoint-set-forest* $p0$
shows $(p0[p0^{T\star} * x - p0^{T\star} * y \longmapsto (p0 * p0)^T]) \sqcap 1 = p0 \sqcap 1$
and $fc\ (p0[p0^{T\star} * x - p0^{T\star} * y \longmapsto (p0 * p0)^T]) = fc\ p0$
and $p0^{T\star} * x \leq p0^\star * root\ p0\ x$

$\langle proof \rangle$

lemma *path-splitting-invariant-aux*:
  assumes *path-splitting-invariant p x y p0*
  shows $p[[y]] = p0[[y]]$
    and $p[[p[[y]]]] = p0[[p0[[y]]]]$
    and $p[[p[[p[[y]]]]]] = p0[[p0[[p0[[y]]]]]]$
    and $p \sqcap 1 = p0 \sqcap 1$
    and $fc\ p = fc\ p0$
$\langle proof \rangle$

lemma *path-splitting-1*:
  find-set-precondition p0 x $\Longrightarrow$ *path-splitting-invariant p0 x x p0*
$\langle proof \rangle$

lemma *path-splitting-2*:
  path-splitting-invariant p x y p0 $\wedge\ y \neq p[[y]] \Longrightarrow$ *path-splitting-invariant*
$(p[y \longmapsto p[[p[[y]]]]])\ x\ (p[[y]])\ p0 \wedge ((p[y \longmapsto p[[p[[y]]]]])^{T\star} * (p[[y]]))\downarrow < (p^{T\star} * y)\downarrow$
$\langle proof \rangle$

lemma *path-splitting-3*:
  path-splitting-invariant p x y p0 $\wedge\ y = p[[y]] \Longrightarrow$ *path-splitting-postcondition p x y p0*
$\langle proof \rangle$

theorem *find-path-splitting*:
  VARS p t y
  [*find-set-precondition p x* $\wedge$ *p0 = p*]
  $y := x$;
  WHILE $y \neq p[[y]]$
    INV { *path-splitting-invariant p x y p0* }
    VAR { $(p^{T\star} * y)\downarrow$ }
     DO $t := p[[y]]$;
        $p[y] := p[[p[[y]]]]$;
        $y := t$
     OD
  [*path-splitting-postcondition p x y p0*]
  $\langle proof \rangle$

end

7 Verifying Union by Rank

In this section we verify the union-by-rank operation of disjoint-set forests. The rank of a node is an upper bound of the height of the subtree rooted at that node. The rank array of a disjoint-set forest maps each node to its rank. This can be represented as a homogeneous relation since the possible rank values are $0, \ldots, n-1$ where n is the number of nodes of the disjoint-set forest.

7.1 Peano structures

Since ranks are natural numbers we start by introducing basic Peano arithmetic. Numbers are represented as (relational) points. Constant Z represents the number 0. Constant S represents the successor function. The successor of a number x is obtained by the relational composition $S^T * x$. The composition $S * x$ results in the predecessor of x.

class *peano-signature* =
  fixes Z :: $'a$
  fixes S :: $'a$

The numbers will be used in arrays, which are represented by homogeneous finite relations. Such relations can only represent finitely many numbers. This means that we weaken the Peano axioms, which are usually used to obtain (infinitely many) natural numbers. Axiom *Z-point* specifies that 0 is a number. Axiom *S-univalent* specifies that every number has at most one 'successor'. Together with axiom *S-total*, which is added later, this means that every number has exactly one 'successor'. Axiom *S-injective* specifies that numbers with the same successor are equal. Axiom *S-star-Z-top* specifies that every number can be obtained from 0 by finitely many applications of the successor. We omit the Peano axiom $S * Z = bot$ which would specify that 0 is not the successor of any number. Since only finitely many numbers will be represented, the remaining axioms will model successor modulo m for some m depending on the carrier of the algebra. That is, the algebra will be able to represent numbers $0, \ldots, m-1$ where the successor of $m-1$ is 0.

class *skra-peano-1* = *stone-kleene-relation-algebra-tarski-consistent* +
peano-signature +
  assumes *Z-point*: *point* Z
  assumes *S-univalent*: *univalent* S
  assumes *S-injective*: *injective* S
  assumes *S-star-Z-top*: $S^{T\star} * Z = top$
begin

lemma *conv-Z-Z*:
  $Z^T * Z = top$
  ⟨*proof*⟩

lemma *Z-below-S-star*:
  $Z \leq S^\star$
⟨*proof*⟩

lemma *S-connected*:
  $S^{T\star} * S^\star = top$

⟨*proof*⟩

lemma *S-star-connex*:
$S^\star \sqcup S^{T\star} = top$
⟨*proof*⟩

lemma *Z-sup-conv-S-top*:
$Z \sqcup S^T * top = top$
⟨*proof*⟩

lemma *top-S-sup-conv-Z*:
$top * S \sqcup Z^T = top$
⟨*proof*⟩

lemma *S-inf-1-below-Z*:
$S \sqcap 1 \leq Z$
⟨*proof*⟩

lemma *S-inf-1-below-conv-Z*:
$S \sqcap 1 \leq Z^T$
⟨*proof*⟩

The successor operation provides a convenient way to compare two natural numbers. Namely, $k < m$ if m can be reached from k by finitely many applications of the successor, formally $m \leq S^{T\star} * k$ or $k \leq S^\star * m$. This does not work for numbers modulo m since comparison depends on the chosen representative. We therefore work with a modified successor relation S', which is a partial function that computes the successor for all numbers except $m-1$. If S is surjective, the point M representing the greatest number $m-1$ is the predecessor of 0 under S. If S is not surjective (like for the set of all natural numbers), $M = bot$.

abbreviation $S' \equiv S - Z^T$
abbreviation $M \equiv S * Z$

lemma *M-point-iff-S-surjective*:
point $M \longleftrightarrow$ *surjective* S
⟨*proof*⟩

lemma *S'-univalent*:
univalent S'
⟨*proof*⟩

lemma *S'-injective*:
injective S'
⟨*proof*⟩

lemma *S'-Z*:
$S' * Z = bot$

⟨*proof*⟩

lemma *S′-irreflexive*:
irreflexive S′
⟨*proof*⟩

end

class *skra-peano-2* = *skra-peano-1* +
assumes *S-total*: *total S*
begin

lemma *S-mapping*:
mapping S
⟨*proof*⟩

lemma *M-bot-iff-S-not-surjective*:
$M \neq bot \longleftrightarrow surjective\ S$
⟨*proof*⟩

lemma *M-point-or-bot*:
$point\ M \vee M = bot$
⟨*proof*⟩

Alternative way to express S'

lemma *S′-var*:
$S' = S - M$
⟨*proof*⟩

Special case of just 1 number

lemma *M-is-Z-iff-1-is-top*:
$M = Z \longleftrightarrow 1 = top$
⟨*proof*⟩

lemma *S-irreflexive*:
assumes $M \neq Z$
shows *irreflexive S*
⟨*proof*⟩

We show that S' satisfies most properties of S.

lemma *M-regular*:
regular M
⟨*proof*⟩

lemma *S′-regular*:
regular S′
⟨*proof*⟩

lemma *S′-star-Z-top*:

$S'^{T\star} * Z = top$
⟨proof⟩

lemma *Z-below-S'-star*:
$Z \leq S'^{\star}$
⟨proof⟩

lemma *S'-connected*:
$S'^{T\star} * S'^{\star} = top$
⟨proof⟩

lemma *S'-star-connex*:
$S'^{\star} \sqcup S'^{T\star} = top$
⟨proof⟩

lemma *Z-sup-conv-S'-top*:
$Z \sqcup S'^{T} * top = top$
⟨proof⟩

lemma *top-S'-sup-conv-Z*:
$top * S' \sqcup Z^{T} = top$
⟨proof⟩

lemma *S-power-point-or-bot*:
assumes *regular* S'
shows *point* $(S'^{T} \mathbin{\hat{}} n * Z) \vee S'^{T} \mathbin{\hat{}} n * Z = bot$
⟨proof⟩

end

7.2 Initialising Ranks

We show that the rank array satisfies three properties which are established/preserved by the union-find operations. First, every node has a rank, that is, the rank array is a mapping. 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. Third, the number of roots in the disjoint-set forest (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. The third property is useful to show that ranks never overflow (exceed $m - 1$). To compare the number of roots and $m - k$ we use the existence of an injective univalent relation between the set of roots and the set of $m - k$ largest numbers, both represented as vectors. The three properties are captured in *rank-property*.

class *skra-peano-3* = *stone-kleene-relation-algebra-tarski-finite-regular* + *skra-peano-2*
begin

definition *card-less-eq v w* $\equiv \exists\, i$. *injective i* $\wedge$ *univalent i* $\wedge$ *regular i* $\wedge$ $v \le i * w$
definition *rank-property p rank* $\equiv$ *mapping rank* $\wedge$ $(p - 1) * rank \le rank * S'^{+}$ $\wedge$ *card-less-eq* (*roots p*) $(-(S'^{+} * rank^T * top))$

end

class *skra-peano-4* = *stone-kleene-relation-algebra-choose-point-finite-regular* + *skra-peano-2*
begin

subclass *skra-peano-3* ⟨*proof*⟩

The initialisation loop is augmented by setting the rank of each node to 0. The resulting rank array satisfies the desired properties explained above.

theorem *init-ranks*:
  VARS h p x rank
  [*True*]
  FOREACH x
    USING h
      INV { $p - h = 1 - h \wedge rank - h = Z^T - h$ }
       DO p := *make-set p x*;
          $rank[x] := Z$
       OD
  [$p = 1 \wedge$ *disjoint-set-forest p* $\wedge\ rank = Z^T \wedge$ *rank-property p rank* $\wedge\ h = bot$]
⟨*proof*⟩

end

7.3 Union by Rank

We show that path compression and union-by-rank preserve the rank property.

context *stone-kleene-relation-algebra-tarski-finite-regular*
begin

lemma *union-sets-1-swap*:
  assumes *union-sets-precondition p0 x y*
    and *path-compression-postcondition p1 x r p0*
    and *path-compression-postcondition p2 y s p1*
  shows *union-sets-postcondition* $(p2[s \longmapsto r])$ *x y p0*
⟨*proof*⟩

lemma *union-sets-1-skip*:
  assumes *union-sets-precondition p0 x y*
    and *path-compression-postcondition p1 x r p0*
    and *path-compression-postcondition p2 y r p1*
  shows *union-sets-postcondition p2 x y p0*
⟨*proof*⟩

end

syntax
-Cond1 :: *'bexp* ⇒ *'com* ⇒ *'com* (‹(1IF -/ THEN -/ FI)› [0,0] 61)
translations *IF b THEN c FI* == *IF b THEN c ELSE SKIP FI*

context *skra-peano-3*
begin

lemma *path-compression-preserves-rank-property*:
assumes *path-compression-postcondition p x y p0*
and *point x*
and *disjoint-set-forest p0*
and *rank-property p0 rank*
shows *rank-property p rank*
⟨*proof*⟩

theorem *union-sets-by-rank*:
VARS p r s rank
[*union-sets-precondition p x y* ∧ *rank-property p rank* ∧ *p0* = *p*]
r := *find-set p x*;
p := *path-compression p x r*;
s := *find-set p y*;
p := *path-compression p y s*;
IF r ≠ *s THEN*
IF rank[[*r*]] ≤ S'^{+} ∗ (*rank*[[*s*]]) *THEN*
p[*r*] := *s*
ELSE
p[*s*] := *r*;
IF rank[[*r*]] = *rank*[[*s*]] *THEN*
rank[*r*] := S'^{T} ∗ (*rank*[[*r*]])
FI
FI
FI
[*union-sets-postcondition p x y p0* ∧ *rank-property p rank*]
⟨*proof*⟩

end

end

8 Matrix Peano Algebras

We define a Boolean matrix representation of natural numbers up to *n*, where *n* the size of an enumeration type *'a::enum*. Numbers (obtained by *Z-matrix* for *0* and *N-matrix n* for *n*) are represented as relational vectors. The total successor function (*S-matrix*, modulo *n*) and the partial successor function (*S'-matrix*, for numbers up to *n−1*) are relations that are (partial) functions.

We give an order-embedding of *nat* into this representation. We show that this representation satisfies a relational version of the Peano axioms. We also implement a function *CP-matrix* that chooses a number in a non-empty set.

theory *Matrix-Peano-Algebras*

imports *Aggregation-Algebras.M-Choose-Component*
Relational-Disjoint-Set-Forests.More-Disjoint-Set-Forests

begin

no-notation *minus-class.minus* (**infixl** ‹−› *65*)

definition *Z-matrix* :: ('*a*::*enum*,'*b*::{*bot*,*top*}) *square* (‹*mZero*›) **where** *mZero* = (λ(*i*,*j*) . *if i* = *hd enum-class.enum then top else bot*)
definition *S-matrix* :: ('*a*::*enum*,'*b*::{*bot*,*top*}) *square* (‹*msuccmod*›) **where** *msuccmod* = (λ(*i*,*j*) . *let e* = (*enum-class.enum* :: '*a list*) *in if* (∃ *k* . *Suc k*<*length e* ∧ *i* = *e* ! *k* ∧ *j* = *e* ! *Suc k*) ∨ (*i* = *e* ! *minus-class.minus* (*length e*) *1* ∧ *j* = *hd e*) *then top else bot*)
definition *S′-matrix* :: ('*a*::*enum*,'*b*::{*bot*,*top*}) *square* (‹*msucc*›) **where** *msucc* = (λ(*i*,*j*) . *let e* = (*enum-class.enum* :: '*a list*) *in if* ∃ *k* . *Suc k*<*length e* ∧ *i* = *e* ! *k* ∧ *j* = *e* ! *Suc k then top else bot*)
definition *N-matrix* :: *nat* ⇒ ('*a*::*enum*,'*b*::{*bot*,*top*}) *square* (‹*mnat*›) **where** *mnat n* = (λ(*i*,*j*) . *if i* = *enum-class.enum* ! *n then top else bot*)
definition *CP-matrix* :: ('*a*::*enum*,'*b*::{*bot*,*uminus*}) *square* ⇒ ('*a*,'*b*) *square* (‹*mcp*›) **where** *mcp f* = (λ(*i*,*j*) . *if Some i* = *find* (λ*x* . *f* (*x*,*x*) ≠ *bot*) *enum-class.enum then uminus-class.uminus* (*uminus-class.uminus* (*f* (*i*,*j*))) *else bot*)

lemma *S′-matrix-S-matrix*:
(*msucc* :: ('*a*::*enum*,'*b*::*stone-relation-algebra*) *square*) = *msuccmod* ⊖ $mZero^t$
⟨*proof*⟩

lemma *N-matrix-power-S*:
n < *length* (*enum-class.enum* :: '*a list*) ⟶ *mnat n* = *matrix-monoid.power* ($msuccmod^t$) *n* ⊙ (*mZero* :: ('*a*::*enum*,'*b*::*stone-relation-algebra*) *square*)
⟨*proof*⟩

lemma *N-matrix-power-S′*:
n < *length* (*enum-class.enum* :: '*a list*) ⟶ *mnat n* = *matrix-monoid.power* ($msucc^t$) *n* ⊙ (*mZero* :: ('*a*::*enum*,'*b*::*stone-relation-algebra*) *square*)
⟨*proof*⟩

lemma *N-matrix-power-S′-hom-zero*:
mnat 0 = (*mZero* :: ('*a*::*enum*,'*b*::*stone-relation-algebra*) *square*)
⟨*proof*⟩

lemma *N-matrix-power-S′-hom-succ*:
assumes *Suc n* < *length* (*enum-class.enum* :: ′*a list*)
shows *mnat* (*Suc n*) = *msucc*^{t} ⊙ (*mnat n* ::
(′*a*::*enum*,′*b*::*stone-relation-algebra*) *square*)
⟨*proof*⟩

lemma *N-matrix-power-S′-hom-inj*:
assumes *m* < *length* (*enum-class.enum* :: ′*a list*)
and *n* < *length* (*enum-class.enum* :: ′*a list*)
and *m* ≠ *n*
shows *mnat m* ≠ (*mnat n* :: (′*a*::*enum*,′*b*::*stone-relation-algebra-consistent*)
square)
⟨*proof*⟩

syntax
-sum-sup-monoid :: *idt* ⇒ *nat* ⇒ ′*a*::*bounded-semilattice-sup-bot* ⇒ ′*a* (‹(⨆ -<- .
-)› [*0*,*51*,*10*] *10*)
syntax-consts
-sum-sup-monoid == *sup-monoid.sum*
translations
⨆*x*<*y* . *t* => *XCONST sup-monoid.sum* (λ*x* . *t*) { *x* . *x* < *y* }

context *bounded-semilattice-sup-bot*
begin

lemma *ub-sum-nat*:
fixes *f* :: *nat* ⇒ ′*a*
assumes *i* < *l*
shows *f i* ≤ (⨆*k*<*l* . *f k*)
⟨*proof*⟩

lemma *lub-sum-nat*:
fixes *f* :: *nat* ⇒ ′*a*
assumes ∀*k*<*l* . *f k* ≤ *x*
shows (⨆*k*<*l* . *f k*) ≤ *x*
⟨*proof*⟩

end

lemma *ext-sum-nat*:
fixes *l* :: *nat*
shows (⨆*k*<*l* . *f k x*) = (⨆*k*<*l* . *f k*) *x*
⟨*proof*⟩

interpretation *matrix-skra-peano-1*: *skra-peano-1* **where** *sup* = *sup-matrix* **and** *inf* = *inf-matrix* **and** *less-eq* = *less-eq-matrix* **and** *less* = *less-matrix* **and** *bot* = *bot-matrix* ::
(′*a*::*enum*,′*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*)
square **and** *top* = *top-matrix* **and** *uminus* = *uminus-matrix* **and** *one* =

one-matrix **and** *times* = *times-matrix* **and** *conv* = *conv-matrix* **and** *star* = *star-matrix* **and** *Z* = *Z-matrix* **and** *S* = *S-matrix*
⟨*proof*⟩

interpretation *matrix-skra-peano-2*: *skra-peano-2* **where** *sup* = *sup-matrix* **and** *inf* = *inf-matrix* **and** *less-eq* = *less-eq-matrix* **and** *less* = *less-matrix* **and** *bot* = *bot-matrix* ::
('*a*::*enum*,'*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*) *square* **and** *top* = *top-matrix* **and** *uminus* = *uminus-matrix* **and** *one* = *one-matrix* **and** *times* = *times-matrix* **and** *conv* = *conv-matrix* **and** *star* = *star-matrix* **and** *Z* = *Z-matrix* **and** *S* = *S-matrix*
⟨*proof*⟩

interpretation *matrix-skra-peano-3*: *skra-peano-3* **where** *sup* = *sup-matrix* **and** *inf* = *inf-matrix* **and** *less-eq* = *less-eq-matrix* **and** *less* = *less-matrix* **and** *bot* = *bot-matrix* ::
('*a*::*enum*,'*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*) *square* **and** *top* = *top-matrix* **and** *uminus* = *uminus-matrix* **and** *one* = *one-matrix* **and** *times* = *times-matrix* **and** *conv* = *conv-matrix* **and** *star* = *star-matrix* **and** *Z* = *Z-matrix* **and** *S* = *S-matrix*
⟨*proof*⟩

interpretation *matrix-skra-peano-4*: *skra-peano-4* **where** *sup* = *sup-matrix* **and** *inf* = *inf-matrix* **and** *less-eq* = *less-eq-matrix* **and** *less* = *less-matrix* **and** *bot* = *bot-matrix* ::
('*a*::*enum*,'*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-plus-expansion*) *square* **and** *top* = *top-matrix* **and** *uminus* = *uminus-matrix* **and** *one* = *one-matrix* **and** *times* = *times-matrix* **and** *conv* = *conv-matrix*
and *star* = *star-matrix* **and** *Z* = *Z-matrix* **and** *S* = *S-matrix* **and** *choose-point* = *agg-square-m-kleene-algebra-2.m-choose-component-algebra-tarski.choose-component-point*
⟨*proof*⟩

interpretation *matrix'-skra-peano-1*: *skra-peano-1* **where** *sup* = *sup-matrix* **and** *inf* = *inf-matrix* **and** *less-eq* = *less-eq-matrix* **and** *less* = *less-matrix* **and** *bot* = *bot-matrix* ::
('*a*::*enum*,'*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*) *square* **and** *top* = *top-matrix* **and** *uminus* = *uminus-matrix* **and** *one* = *one-matrix* **and** *times* = *times-matrix* **and** *conv* = *conv-matrix* **and** *star* = *star-matrix* **and** *Z* = *Z-matrix* **and** *S* = *S'-matrix*
⟨*proof*⟩

lemma *nat-less-lesseq-pred*:
$(m :: nat) < n \Longrightarrow m \leq$ *minus-class.minus* n *1*
⟨*proof*⟩

lemma *S'-matrix-acyclic*:
matrix-stone-kleene-relation-algebra.acyclic (*msucc* ::
('*a*::*enum*,'*b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*) *square*)

⟨*proof*⟩

lemma *N-matrix-point*:
assumes n < *length* (*enum-class.enum* :: *'a list*)
shows *matrix-stone-relation-algebra.point* (*mnat* n ::
(*'a*::*enum*,*'b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*)
square)
⟨*proof*⟩

lemma *N-matrix-power-S'-hom-lesseq*:
assumes m < *length* (*enum-class.enum* :: *'a list*)
and n < *length* (*enum-class.enum* :: *'a list*)
shows $m < n \longleftrightarrow$ *mnat* $m \preceq$ *msucc* $\odot$ *msucc*$^{\odot}$ $\odot$ (*mnat* n ::
(*'a*::*enum*,*'b*::*linorder-stone-kleene-relation-algebra-tarski-consistent-expansion*)
square)
⟨*proof*⟩

end

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer, New York, 1998.

[2] R. C. Backhouse and B. A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15(2):161–186, 1975.

[3] R. Berghammer. Combining relational calculus and the Dijkstra–Gries method for deriving relational programs. *Information Sciences*, 119(3–4):155–171, 1999.

[4] R. Berghammer and G. Struth. On automated program construction and verification. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2010.

[5] R. Berghammer, B. von Karger, and A. Wolf. Relation-algebraic derivation of spanning tree algorithms. In J. Jeuring, editor, *Mathematics of Program Construction (MPC 1998)*, volume 1422 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 1998.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A lin-

guistic approach to the view-update problem. *ACM Trans. Prog. Lang. Syst.*, 29(3:17):1–65, 2007.

[8] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[9] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings*. Springer, 2008.

[10] W. Guttmann. Verifying minimum spanning tree algorithms with Stone relation algebras. *Journal of Logical and Algebraic Methods in Programming*, 101:132–150, 2018.

[11] W. Guttmann. Verifying the correctness of disjoint-set forests with Kleene relation algebras. In U. Fahrenberg, P. Jipsen, and M. Winter, editors, *Relational and Algebraic Methods in Computer Science (RAMiCS 2020)*, volume 12062 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2020.

[12] W. Guttmann. Relation-algebraic verification of disjoint-set forests. *arXiv*, 2301.10311, 2023. https://arxiv.org/abs/2301.10311.

[13] P. Höfner and B. Möller. Dijkstra, Floyd and Warshall meet Kleene. *Formal Aspects of Computing*, 24(4):459–476, 2012.

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

[15] P. Lammich and R. Meis. A separation logic framework for Imperative HOL. *Archive of Formal Proofs*, 2012.

[16] B. Möller. Derivation of graph and pointer algorithms. In B. Möller, H. A. Partsch, and S. A. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 123–160. Springer, 1993.

[17] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

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

[19] B. Zhan. Verifying imperative programs using Auto2. *Archive of Formal Proofs*, 2018.