A highly-available move operation for replicated trees and distributed filesystems (Proof Document)

Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford

Contents


```
theory Move
  imports Main
begin
```

1 Definitions

```
datatype ('t, 'n, 'm) operation
  = Move (move_time: 't)
         (move_parent: 'n)
         (move_meta: 'm)
         (move_child: 'n)

datatype ('t, 'n, 'm) log_op
  = LogMove (log_time: 't)
            (old_parent: ‹('n × 'm) option›)
            (new_parent: 'n)
            (log_meta: 'm)
            (log_child: 'n)

type_synonym ('t, 'n, 'm) state = ‹('t, 'n, 'm) log_op list × ('n × 'm × 'n) set›

definition get_parent :: ‹('n × 'm × 'n) set ⇒ 'n ⇒ ('n × 'm) option› where
  ‹get_parent tree child ≡
     if ∃!parent. ∃!meta. (parent, meta, child) ∈ tree then
       Some (THE (parent, meta). (parent, meta, child) ∈ tree)
     else None›

inductive ancestor :: ‹('n × 'm × 'n) set ⇒ 'n ⇒ 'n ⇒ bool› where
  ‹⟦(parent, meta, child) ∈ tree⟧ ⟹ ancestor tree parent child› |
  ‹⟦(parent, meta, child) ∈ tree; ancestor tree anc parent⟧ ⟹ ancestor tree anc child›

inductive_cases ancestor_indcases: ‹ancestor 𝒯 m p›

fun do_op :: ‹('t, 'n, 'm) operation × ('n × 'm × 'n) set ⇒
              ('t, 'n, 'm) log_op × ('n × 'm × 'n) set› where
```

```
  ‹do_op (Move t newp m c, tree) =
     (LogMove t (get_parent tree c) newp m c,
      if ancestor tree c newp ∨ c = newp then tree
      else {(p', m', c') ∈ tree. c' ≠ c} ∪ {(newp, m, c)})›

fun undo_op :: ‹('t, 'n, 'm) log_op × ('n × 'm × 'n) set ⇒ ('n × 'm × 'n) set› where
  ‹undo_op (LogMove t None newp m c, tree) = {(p', m', c') ∈ tree. c' ≠ c}› |
  ‹undo_op (LogMove t (Some (oldp, oldm)) newp m c, tree) =
     {(p', m', c') ∈ tree. c' ≠ c} ∪ {(oldp, oldm, c)}›

fun redo_op :: ‹('t, 'n, 'm) log_op ⇒ ('t, 'n, 'm) state ⇒ ('t, 'n, 'm) state› where
  ‹redo_op (LogMove t _ p m c) (ops, tree) =
     (let (op2, tree2) = do_op (Move t p m c, tree)
      in (op2 # ops, tree2))›

fun apply_op :: ‹('t::{linorder}, 'n, 'm) operation ⇒
                  ('t, 'n, 'm) state ⇒ ('t, 'n, 'm) state› where
  ‹apply_op op1 ([], tree1) =
     (let (op2, tree2) = do_op (op1, tree1)
      in ([op2], tree2))› |
  ‹apply_op op1 (logop # ops, tree1) =
     (if move_time op1 < log_time logop
      then redo_op logop (apply_op op1 (ops, undo_op (logop, tree1)))
      else let (op2, tree2) = do_op (op1, tree1) in (op2 # logop # ops, tree2))›

abbreviation apply_ops' :: ‹('t::{linorder}, 'n, 'm) operation list ⇒ ('t, 'n, 'm) state ⇒ ('t,
'n, 'm) state› where
  ‹apply_ops' ops initial ≡ foldl (λstate oper. apply_op oper state) initial ops›

definition apply_ops :: ‹('t::{linorder}, 'n, 'm) operation list ⇒ ('t, 'n, 'm) state›
  where ‹apply_ops ops ≡ apply_ops' ops ([], {})›

definition unique_parent :: ‹('n × 'm × 'n) set ⇒ bool› where
  ‹unique_parent tree ≡ (∀p1 p2 m1 m2 c. (p1, m1, c) ∈ tree ∧ (p2, m2, c) ∈ tree ⟶ p1 = p2 ∧
m1 = m2)›

lemma unique_parent_empty[simp]:
  shows ‹unique_parent {}›
  by (auto simp: unique_parent_def)

lemma unique_parentD [dest]:
  assumes ‹unique_parent T›
      and ‹(p1, m1, c) ∈ T›
      and ‹(p2, m2, c) ∈ T›
    shows ‹p1 = p2 ∧ m1 = m2›
using assms by(force simp add: unique_parent_def)

lemma unique_parentI [intro]:
  assumes ‹⋀p1 p2 m1 m2 c. (p1, m1, c) ∈ T ⟹ (p2, m2, c) ∈ T ⟹ p1 = p2 ∧ m1 = m2›
  shows ‹unique_parent T›
using assms by(force simp add: unique_parent_def)

lemma apply_ops_base [simp]:
  shows ‹apply_ops [Move t1 p1 m1 c1, Move t2 p2 m2 c2] =
                    apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) ([], {}))›
  by (clarsimp simp add: apply_ops_def)

lemma apply_ops_step [simp]:
  shows ‹apply_ops (xs @ [x]) = apply_op x (apply_ops xs)›
  by (clarsimp simp add: apply_ops_def)

lemma apply_ops_Nil [simp]:
  shows ‹apply_ops [] = ([], {})›
```

```
  by (clarsimp simp add: apply_ops_def)
```

2 Tree invariant 1: at most one parent

```
lemma subset_unique_parent:
  assumes ‹unique_parent tree›
  shows ‹unique_parent {(p', m', c') ∈ tree. c' ≠ c}›
proof -
  {
    fix p1 p2 m1 m2 c'
    assume 1: ‹(p1, m1, c') ∈ {(p', m', c') ∈ tree. c' ≠ c}›
       and 2: ‹(p2, m2, c') ∈ {(p', m', c') ∈ tree. c' ≠ c}›
    have ‹p1 = p2 ∧ m1 = m2›
    proof(cases ‹c = c'›)
      case True
      then show ?thesis using 1 2 by auto
    next
      case False
      hence ‹(p1, m1, c') ∈ tree ∧ (p2, m2, c') ∈ tree›
        using 1 2 by blast
      then show ?thesis
        using assms by (meson unique_parent_def)
    qed
  }
  thus ?thesis by (meson unique_parent_def)
qed

lemma subset_union_unique_parent:
  assumes ‹unique_parent tree›
  shows ‹unique_parent ({(p', m', c') ∈ tree. c' ≠ c} ∪ {(p, m, c)})›
proof -
  {
    fix p1 p2 m1 m2 c'
    assume 1: ‹(p1, m1, c') ∈ {(p', m', c') ∈ tree. c' ≠ c} ∪ {(p, m, c)}›
       and 2: ‹(p2, m2, c') ∈ {(p', m', c') ∈ tree. c' ≠ c} ∪ {(p, m, c)}›
    have ‹p1 = p2 ∧ m1 = m2›
    proof(cases ‹c = c'›)
      case True
      then show ?thesis using 1 2 by auto
    next
      case False
      hence ‹(p1, m1, c') ∈ tree ∧ (p2, m2, c') ∈ tree›
        using 1 2 by blast
      then show ?thesis
        using assms by (meson unique_parent_def)
    qed
  }
  thus ?thesis by (meson unique_parent_def)
qed

lemma do_op_unique_parent:
  assumes ‹unique_parent tree1›
    and ‹do_op (Move t newp m c, tree1) = (log_oper, tree2)›
  shows ‹unique_parent tree2›
proof(cases ‹ancestor tree1 c newp ∨ c = newp›)
  case True
  hence ‹tree1 = tree2›
    using assms(2) by auto
  thus ‹unique_parent tree2›
    by (metis (full_types) assms(1))
next
  case False
  hence ‹tree2 = {(p', m', c') ∈ tree1. c' ≠ c} ∪ {(newp, m, c)}›
```

```
    using assms(2) by auto
  then show ‹unique_parent tree2›
    using subset_union_unique_parent assms(1) by fastforce
qed

lemma undo_op_unique_parent:
  assumes ‹unique_parent tree1›
    and ‹undo_op (LogMove t oldp newp m c, tree1) = tree2›
  shows ‹unique_parent tree2›
proof (cases oldp)
  case None
  hence ‹tree2 = {(p', m', c') ∈ tree1. c' ≠ c}›
    using assms(2) by auto
  then show ?thesis
    by (simp add: assms(1) subset_unique_parent)
next
  case (Some old)
  obtain oldp oldm where ‹old = (oldp, oldm)›
    by fastforce
  hence ‹tree2 = {(p', m', c') ∈ tree1. c' ≠ c} ∪ {(oldp, oldm, c)}›
    using Some assms(2) by auto
  then show ?thesis
    using subset_union_unique_parent assms(1) by fastforce
qed

corollary undo_op_unique_parent_variant:
  assumes ‹unique_parent tree1›
    and ‹undo_op (oper, tree1) = tree2›
  shows ‹unique_parent tree2›
using assms by(cases oper, auto simp add: undo_op_unique_parent)

lemma redo_op_unique_parent:
  assumes ‹unique_parent tree1›
    and ‹redo_op oper (ops1, tree1) = (ops2, tree2)›
  shows ‹unique_parent tree2›
proof -
  obtain t oldp newp m c where ‹oper = LogMove t oldp newp m c›
    using log_op.exhaust by blast
  from this obtain move2 where ‹(move2, tree2) = do_op (Move t newp m c, tree1)›
    using assms(2) by auto
  thus ‹unique_parent tree2›
    by (metis assms(1) do_op_unique_parent)
qed

lemma apply_op_unique_parent:
  assumes ‹unique_parent tree1›
    and ‹apply_op oper (ops1, tree1) = (ops2, tree2)›
  shows ‹unique_parent tree2›
using assms proof(induct ops1 arbitrary: tree1 tree2 ops2)
  case Nil
  have ‹⋀pair. snd (case pair of (p1, p2) ⇒ ([p1], p2)) = snd pair›
    by (simp add: prod.case_eq_if)
  hence ‹∃log_op. do_op (oper, tree1) = (log_op, tree2)›
    by (metis Nil.prems(2) apply_op.simps(1) prod.collapse snd_conv)
  thus ‹unique_parent tree2›
    by (metis Nil.prems(1) do_op_unique_parent operation.exhaust_sel)
next
  case step: (Cons logop ops)
  then show ‹unique_parent tree2›
  proof(cases ‹move_time oper < log_time logop›)
    case True
    moreover obtain tree1a where ‹tree1a = undo_op (logop, tree1)›
      by simp
```

```
    moreover from this have 1: ‹unique_parent tree1a›
      using undo_op_unique_parent by (metis step.prems(1) log_op.exhaust_sel)
    moreover obtain ops1b tree1b where ‹(ops1b, tree1b) = apply_op oper (ops, tree1a)›
      by (metis surj_pair)
    moreover from this have ‹unique_parent tree1b›
      using 1 by (metis step.hyps)
    ultimately show ‹unique_parent tree2›
      using redo_op_unique_parent by (metis apply_op.simps(2) step.prems(2))
  next
    case False
    hence ‹snd (do_op (oper, tree1)) = tree2›
      by (metis (mono_tags, lifting) apply_op.simps(2) prod.sel(2) split_beta step.prems(2))
    then show ‹unique_parent tree2›
      by (metis do_op_unique_parent operation.exhaust_sel prod.exhaust_sel step.prems(1))
  qed
qed

theorem apply_ops_unique_parent:
  assumes ‹apply_ops ops = (log, tree)›
  shows ‹unique_parent tree›
using assms proof(induction ops arbitrary: log tree rule: List.rev_induct)
  case Nil
  hence ‹apply_ops [] = ([], {})›
    by (simp add: apply_ops_def)
  hence ‹tree = {}›
    by (metis Nil.prems snd_conv)
  then show ?case
    by (simp add: unique_parent_def)
next
  case (snoc x xs)
  obtain log tree where apply_xs: ‹apply_ops xs = (log, tree)›
    by fastforce
  hence ‹apply_ops (xs @ [x]) = apply_op x (log, tree)›
    by (simp add: apply_ops_def)
  moreover have ‹unique_parent tree›
    by (simp add: apply_xs snoc.IH)
  ultimately show ?case
    by (metis apply_op_unique_parent snoc.prems)
qed
```

3 Move operation properties

3.1 undo-op is the inverse of do-op

```
lemma get_parent_None:
  assumes ‹∄p m. (p, m, c) ∈ tree›
  shows ‹get_parent tree c = None›
  by (meson assms get_parent_def)

lemma get_parent_Some:
  assumes ‹(p, m, c) ∈ tree›
    and ‹⋀p' m'. (p', m', c) ∈ tree ⟹ p' = p ∧ m' = m›
  shows ‹get_parent tree c = Some (p, m)›
proof -
  have ‹∃!parent. ∃!meta. (parent, meta, c) ∈ tree›
    using assms by metis
  hence ‹(THE (parent, meta). (parent, meta, c) ∈ tree) = (p, m)›
    using assms(2) by auto
  thus ‹get_parent tree c = Some (p, m)›
    using assms get_parent_def by metis
qed

lemma pred_equals_eq3:
```

```
  shows ‹(λx y z. (x, y, z) ∈ R) = (λx y z. (x, y, z) ∈ S) ⟷ R = S›
  by (simp add: set_eq_iff fun_eq_iff)

lemma do_undo_op_inv:
  assumes ‹unique_parent tree›
  shows ‹undo_op (do_op (Move t p m c, tree)) = tree›
proof(cases ‹∃par meta. (par, meta, c) ∈ tree›)
  case True
  from this obtain oldp oldm where 1: ‹(oldp, oldm, c) ∈ tree›
    by blast
  hence 2: ‹get_parent tree c = Some (oldp, oldm)›
    using assms get_parent_Some unique_parent_def by metis
  {
    fix p' m' c'
    assume 3: ‹(p', m', c') ∈ tree›
    hence ‹(p', m', c') ∈ undo_op (do_op (Move t p m c, tree))›
      using 1 2 assms unique_parent_def by (cases ‹c = c'›; fastforce)
  }
  hence ‹tree ⊆ undo_op (do_op (Move t p m c, tree))›
    by auto
  moreover have ‹undo_op (do_op (Move t p m c, tree)) ⊆ tree›
    using 1 2 by auto
  ultimately show ?thesis
    by blast
next
  case no_old_parent: False
  hence ‹get_parent tree c = None›
    using assms get_parent_None by metis
  moreover have ‹{(p', m', c') ∈ tree. c' ≠ c} = tree›
    using no_old_parent by fastforce
  moreover from this have ‹{(p', m', c') ∈ (tree ∪ {(p, m, c)}). c' ≠ c} = tree›
    by blast
  ultimately show ?thesis by simp
qed

lemma do_undo_op_inv_var:
  assumes ‹unique_parent tree›
  shows ‹undo_op (do_op (oper, tree)) = tree›
  using assms do_undo_op_inv by (metis operation.exhaust_sel)
```

3.2 Commutativity

```
lemma distinct_list_pick1:
  assumes ‹set (xs @ [x]) = set (ys @ [x] @ zs)›
    and ‹distinct (xs @ [x])› and ‹distinct (ys @ [x] @ zs)›
  shows ‹set xs = set (ys @ zs)›
using assms by (induction xs) (fastforce+)

lemma apply_op_commute_base:
  assumes ‹t1 < t2›
    and ‹unique_parent tree›
  shows ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) ([], tree)) =
         apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) ([], tree))›
proof -
  obtain tree1 where tree1: ‹do_op (Move t1 p1 m1 c1, tree) =
      (LogMove t1 (get_parent tree c1) p1 m1 c1, tree1)›
    by simp
  obtain tree12 where tree12: ‹do_op (Move t2 p2 m2 c2, tree1) =
      (LogMove t2 (get_parent tree1 c2) p2 m2 c2, tree12)›
    by simp
  obtain tree2 where tree2: ‹do_op (Move t2 p2 m2 c2, tree) =
      (LogMove t2 (get_parent tree c2) p2 m2 c2, tree2)›
    by simp
```

```
  hence undo2: ‹undo_op (LogMove t2 (get_parent tree c2) p2 m2 c2, tree2) = tree›
    using assms(2) do_undo_op_inv by fastforce
  have ‹¬ t2 < t1›
    using not_less_iff_gr_or_eq assms(1) by blast
  hence ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) ([], tree)) =
        ([LogMove t2 (get_parent tree1 c2) p2 m2 c2, LogMove t1 (get_parent tree c1) p1 m1 c1], tree12)›
    using tree1 tree12 by auto
  moreover have ‹apply_op (Move t2 p2 m2 c2) ([], tree) =
      ([LogMove t2 (get_parent tree c2) p2 m2 c2], tree2)›
    using tree2 by auto
  hence ‹apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) ([], tree)) =
         redo_op (LogMove t2 (get_parent tree c2) p2 m2 c2) ([LogMove t1 (get_parent tree c1) p1
m1 c1], tree1)›
    using tree1 undo2 assms(1) by auto
  ultimately show ?thesis
    using tree12 by auto
qed

lemma apply_op_log_cons:
  assumes ‹apply_op (Move t1 p1 m1 c1) (log, tree) = (log2, tree2)›
  shows ‹∃logop rest. log2 = logop # rest ∧ t1 ≤ log_time logop›
proof(cases log)
  case Nil
  then show ?thesis using assms by auto
next
  case (Cons logop rest)
  obtain t2 oldp2 p2 m2 c2 where logop: ‹logop = LogMove t2 oldp2 p2 m2 c2›
    using log_op.exhaust by blast
  then show ?thesis
  proof(cases ‹t1 < t2›)
    case True
    obtain tree1 log1 where tree1: ‹apply_op (Move t1 p1 m1 c1) (rest, undo_op (logop, tree)) = (log1,
tree1)›
      by fastforce
    obtain tree12 where ‹do_op (Move t2 p2 m2 c2, tree1) = (LogMove t2 (get_parent tree1 c2) p2 m2
c2, tree12)›
      by simp
    hence ‹apply_op (Move t1 p1 m1 c1) (log, tree) = (LogMove t2 (get_parent tree1 c2) p2 m2 c2 #
log1, tree12)›
      using True local.Cons tree1 logop by auto
    then show ?thesis
      using True assms by auto
  next
    case False
    obtain tree1 where tree1: ‹do_op (Move t1 p1 m1 c1, tree) = (LogMove t1 (get_parent tree c1)
p1 m1 c1, tree1)›
      by simp
    hence ‹apply_op (Move t1 p1 m1 c1) (log, tree) =
           (LogMove t1 (get_parent tree c1) p1 m1 c1 # log, tree1)›
      using False local.Cons logop by auto
    then show ?thesis
      using assms by auto
  qed
qed

lemma apply_op_commute2:
  assumes ‹t1 < t2›
    and ‹unique_parent tree›
    and ‹distinct ((map log_time log) @ [t1, t2])›
  shows ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (log, tree)) =
         apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (log, tree))›
using assms proof(induction log arbitrary: tree)
  case Nil
```

```
  then show ?case using apply_op_commute_base by metis
next
  case (Cons logop log)
  have parent0: ‹unique_parent (undo_op (logop, tree))›
    by (metis Cons.prems(2) log_op.exhaust_sel undo_op_unique_parent)
  obtain t3 oldp3 p3 m3 c3 where logop: ‹logop = LogMove t3 oldp3 p3 m3 c3›
    using log_op.exhaust by blast
  then consider (c1) ‹t3 < t1› | (c2) ‹t1 < t3 ∧ t3 < t2› | (c3) ‹t2 < t3›
    using Cons.prems(3) by force
  then show ?case
  proof(cases)
    case c1
    obtain tree1 where tree1: ‹do_op (Move t1 p1 m1 c1, tree) = (LogMove t1 (get_parent tree c1)
p1 m1 c1, tree1)›
      by simp
    obtain tree12 where tree12: ‹do_op (Move t2 p2 m2 c2, tree1) = (LogMove t2 (get_parent tree1
c2) p2 m2 c2, tree12)›
      by simp
    obtain tree2 where tree2: ‹do_op (Move t2 p2 m2 c2, tree) = (LogMove t2 (get_parent tree c2)
p2 m2 c2, tree2)›
      by simp
    hence undo2: ‹undo_op (LogMove t2 (get_parent tree c2) p2 m2 c2, tree2) = tree›
      using Cons.prems(2) do_undo_op_inv by metis
    have ‹¬ t2 < t1›
      using not_less_iff_gr_or_eq Cons.prems(1) by blast
    hence ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (logop # log, tree)) =
          ([LogMove t2 (get_parent tree1 c2) p2 m2 c2, LogMove t1 (get_parent tree c1) p1 m1 c1,
logop] @ log, tree12)›
      using tree1 tree12 logop c1 by auto
    moreover have ‹t3 < t2›
      using c1 Cons.prems(1) by auto
    hence ‹apply_op (Move t2 p2 m2 c2) (logop # log, tree) = (LogMove t2 (get_parent tree c2) p2
m2 c2 # logop # log, tree2)›
      using tree2 logop by auto
    hence ‹apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (logop # log, tree)) =
          redo_op (LogMove t2 (get_parent tree c2) p2 m2 c2) (LogMove t1 (get_parent tree c1) p1
m1 c1 # logop # log, tree1)›
      using Cons.prems(1) c1 logop tree1 undo2 by auto
    ultimately show ?thesis
      using tree12 by auto
  next
    case c2
    obtain tree1 log1 where tree1: ‹apply_op (Move t1 p1 m1 c1) (log, undo_op (logop, tree)) = (log1,
tree1)›
      by fastforce
    obtain tree13 where tree13: ‹do_op (Move t3 p3 m3 c3, tree1) = (LogMove t3 (get_parent tree1
c3) p3 m3 c3, tree13)›
      by simp
    obtain tree132 where tree132: ‹do_op (Move t2 p2 m2 c2, tree13) = (LogMove t2 (get_parent tree13
c2) p2 m2 c2, tree132)›
      by simp
    obtain tree2 where tree2: ‹do_op (Move t2 p2 m2 c2, tree) = (LogMove t2 (get_parent tree c2)
p2 m2 c2, tree2)›
      by simp
    hence undo2: ‹undo_op (LogMove t2 (get_parent tree c2) p2 m2 c2, tree2) = tree›
      by (metis Cons.prems(2) do_undo_op_inv)
    have ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (logop # log, tree)) =
          (LogMove t2 (get_parent tree13 c2) p2 m2 c2 # LogMove t3 (get_parent tree1 c3) p3 m3 c3
# log1, tree132)›
      using c2 logop tree1 tree13 tree132 by auto
    moreover have ‹apply_op (Move t2 p2 m2 c2) (logop # log, tree) =
                  (LogMove t2 (get_parent tree c2) p2 m2 c2 # logop # log, tree2)›
      using c2 logop tree2 by auto
```

```
    hence ‹apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (logop # log, tree)) =
           (LogMove t2 (get_parent tree13 c2) p2 m2 c2 # LogMove t3 (get_parent tree1 c3) p3 m3 c3
# log1, tree132)›
      using assms(1) undo2 c2 logop tree1 tree13 tree132 by auto
    ultimately show ?thesis by simp
  next
    case c3
    obtain tree1 log1 where tree1: ‹apply_op (Move t1 p1 m1 c1) (log, undo_op (logop, tree)) = (log1,
tree1)›
      by fastforce
    obtain tree13 where tree13: ‹do_op (Move t3 p3 m3 c3, tree1) = (LogMove t3 (get_parent tree1
c3) p3 m3 c3, tree13)›
      by simp
    hence undo13: ‹undo_op (LogMove t3 (get_parent tree1 c3) p3 m3 c3, tree13) = tree1›
    proof -
      have ‹unique_parent tree1›
        by (meson apply_op_unique_parent parent0 tree1)
      thus ?thesis
        using do_undo_op_inv tree13 by metis
    qed
    obtain tree12 log12 where tree12: ‹apply_op (Move t2 p2 m2 c2) (log1, tree1) = (log12, tree12)›
      by fastforce
    obtain tree123 where tree123: ‹do_op (Move t3 p3 m3 c3, tree12) = (LogMove t3 (get_parent tree12
c3) p3 m3 c3, tree123)›
      by simp
    obtain tree2 log2 where tree2: ‹apply_op (Move t2 p2 m2 c2) (log, undo_op (logop, tree)) = (log2,
tree2)›
      by fastforce
    obtain tree21 log21 where tree21: ‹apply_op (Move t1 p1 m1 c1) (log2, tree2) = (log21, tree21)›
      by fastforce
    obtain tree213 where tree213: ‹do_op (Move t3 p3 m3 c3, tree21) = (LogMove t3 (get_parent tree21
c3) p3 m3 c3, tree213)›
      by simp
    obtain tree23 where tree23: ‹do_op (Move t3 p3 m3 c3, tree2) = (LogMove t3 (get_parent tree2
c3) p3 m3 c3, tree23)›
      by simp
    hence undo23: ‹undo_op (LogMove t3 (get_parent tree2 c3) p3 m3 c3, tree23) = tree2›
    proof -
      have ‹unique_parent tree2›
        by (meson apply_op_unique_parent parent0 tree2)
      thus ?thesis
        using do_undo_op_inv tree23 by metis
    qed
    have ‹apply_op (Move t1 p1 m1 c1) (logop # log, tree) =
           (LogMove t3 (get_parent tree1 c3) p3 m3 c3 # log1, tree13)›
      using assms(1) c3 logop tree1 tree13 by auto
    hence ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (logop # log, tree)) =
           (LogMove t3 (get_parent tree12 c3) p3 m3 c3 # log12, tree123)›
      using c3 tree12 tree123 undo13 by auto
    moreover have ‹apply_op (Move t2 p2 m2 c2) (logop # log, tree) =
          (LogMove t3 (get_parent tree2 c3) p3 m3 c3 # log2, tree23)›
      using c3 logop tree2 tree23 by auto
    hence ‹apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (logop # log, tree)) =
           (LogMove t3 (get_parent tree21 c3) p3 m3 c3 # log21, tree213)›
      using assms(1) c3 undo23 tree21 tree213 by auto
    moreover have ‹apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (log, undo_op (logop,
tree))) =
                  apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (log, undo_op (logop,
tree)))›
      using Cons.IH Cons.prems(3) assms(1) parent0 by auto
    ultimately show ?thesis
      using tree1 tree12 tree123 tree2 tree21 tree213 by auto
  qed
```

```
qed

corollary apply_op_commute2I:
  assumes ‹unique_parent tree›
    and ‹distinct ((map log_time log) @ [t1, t2])›
    and ‹apply_op (Move t1 p1 m1 c1) (log, tree) = (log1, tree1)›
    and ‹apply_op (Move t2 p2 m2 c2) (log, tree) = (log2, tree2)›
  shows ‹apply_op (Move t2 p2 m2 c2) (log1, tree1) = apply_op (Move t1 p1 m1 c1) (log2, tree2)›
proof (case_tac ‹t1 < t2›, metis assms apply_op_commute2)
  assume ‹¬ t1 < t2›
  hence ‹t2 < t1›
    using assms by force
  moreover have ‹distinct ((map log_time log) @ [t2, t1])›
    using assms by force
  ultimately show ?thesis
    using assms apply_op_commute2 by metis
qed

corollary apply_op_commute2':
  assumes ‹unique_parent tree›
    and ‹distinct ((map log_time log) @
               (map move_time [oper1, oper2]))›
  shows ‹apply_op oper2 (apply_op oper1 (log, tree)) =
         apply_op oper1 (apply_op oper2 (log, tree))›
proof -
  obtain t1 p1 m1 c1 where op1: ‹oper1 = Move t1 p1 m1 c1›
    using operation.exhaust by blast
  moreover obtain t2 p2 m2 c2 where op2: ‹oper2 = Move t2 p2 m2 c2›
    using operation.exhaust by blast
  moreover have ‹distinct ((map log_time log) @ [t1, t2])›
    using assms(2) op1 op2 by auto
  moreover obtain tree1 log1 where ‹apply_op (Move t1 p1 m1 c1) (log, tree) = (log1, tree1)›
    using prod.exhaust_sel by blast
  moreover obtain tree2 log2 where ‹apply_op (Move t2 p2 m2 c2) (log, tree) = (log2, tree2)›
    using prod.exhaust_sel by blast
  moreover have ‹apply_op (Move t2 p2 m2 c2) (log1, tree1) = apply_op (Move t1 p1 m1 c1) (log2,
tree2)›
    using apply_op_commute2I assms(1) calculation by fastforce
  ultimately show ?thesis
    by simp
qed

lemma apply_op_timestamp:
  assumes ‹distinct ((map log_time log1) @ [t])›
    and ‹apply_op (Move t p m c) (log1, tree1) = (log2, tree2)›
  shows ‹distinct (map log_time log2) ∧ set (map log_time log2) = {t} ∪ set (map log_time log1)›
using assms proof(induction log1 arbitrary: tree1 log2 tree2)
  case Nil
  then show ?case by auto
next
  case (Cons logop log)
  obtain log3 tree3 where log3: ‹apply_op (Move t p m c) (log, undo_op (logop, tree1)) = (log3, tree3)›
    using prod.exhaust_sel by blast
  have ‹distinct ((map log_time log) @ [t])›
    using Cons.prems(1) by auto
  hence IH: ‹distinct (map log_time log3) ∧ set (map log_time log3) = {t} ∪ set (map log_time log)›
    using Cons.IH Cons.prems(1) log3 by auto
  then show ?case
  proof(cases ‹t < log_time logop›)
    case recursive_case: True
    obtain t2 oldp2 p2 m2 c2 where logop: ‹logop = LogMove t2 oldp2 p2 m2 c2›
      using log_op.exhaust by blast
    obtain tree4 where ‹do_op (Move t2 p2 m2 c2, tree3) = (LogMove t2 (get_parent tree3 c2) p2 m2
```

```
c2, tree4)›
      by simp
    hence ‹apply_op (Move t p m c) (logop # log, tree1) =
          (LogMove t2 (get_parent tree3 c2) p2 m2 c2 # log3, tree4)›
      using logop log3 recursive_case by auto
    moreover from this have ‹set (map log_time log2) = {t} ∪ set (map log_time (logop # log))›
      using Cons.prems(2) IH logop by fastforce
    moreover have ‹distinct (map log_time (LogMove t2 (get_parent tree3 c2) p2 m2 c2 # log3))›
      using Cons.prems(1) IH logop recursive_case by auto
    ultimately show ?thesis
      using Cons.prems(2) by auto
  next
    case cons_case: False
    obtain tree4 where ‹do_op (Move t p m c, tree1) = (LogMove t (get_parent tree1 c) p m c, tree4)›
      by simp
    hence ‹apply_op (Move t p m c) (logop # log, tree1) =
          (LogMove t (get_parent tree1 c) p m c # logop # log, tree4)›
      by (simp add: cons_case)
    moreover from this have ‹set (map log_time log2) = {t} ∪ set (map log_time (logop # log))›
      using Cons.prems(2) by auto
    moreover have ‹distinct (t # map log_time (logop # log))›
      using Cons.prems(1) by auto
    ultimately show ?thesis
      using Cons.prems(2) by auto
  qed
qed

corollary apply_op_timestampI1:
  assumes ‹apply_op (Move t p m c) (log1, tree1) = (log2, tree2)› ‹distinct ((map log_time log1) @
[t])›
  shows ‹distinct (map log_time log2)›
  using assms apply_op_timestamp by metis

corollary apply_op_timestampI2:
  assumes ‹apply_op (Move t p m c) (log1, tree1) = (log2, tree2)› ‹distinct ((map log_time log1) @
[t])›
  shows ‹set (map log_time log2) = {t} ∪ set (map log_time log1)›
  using assms apply_op_timestamp by metis

lemma apply_ops_timestamps:
  assumes ‹distinct (map move_time ops)›
    and ‹apply_ops ops = (log, tree)›
  shows ‹distinct (map log_time log) ∧ set (map move_time ops) = set (map log_time log)›
using assms proof(induction ops arbitrary: log tree rule: List.rev_induct, simp)
  case (snoc oper ops)
  obtain log1 tree1 where log1: ‹apply_ops ops = (log1, tree1)›
    by fastforce
  hence IH: ‹distinct (map log_time log1) ∧ set (map move_time ops) = set (map log_time log1)›
    using snoc by auto
  hence ‹set (map move_time (ops @ [oper])) = {move_time oper} ∪ set (map log_time log1)›
    by force
  moreover have ‹distinct (map log_time log1 @ [move_time oper])›
    using log1 snoc(1) snoc.prems(1) by force
  ultimately show ?case
    by (metis (no_types) apply_op_timestamp apply_ops_step log1 operation.exhaust_sel snoc.prems(2))
qed

lemma apply_op_commute_last:
  assumes ‹distinct ((map move_time ops) @ [t1, t2])›
  shows ‹apply_ops (ops @ [Move t1 p1 m1 c1, Move t2 p2 m2 c2]) =
        apply_ops (ops @ [Move t2 p2 m2 c2, Move t1 p1 m1 c1])›
proof -
  obtain log tree where apply_ops: ‹apply_ops ops = (log, tree)›
```

```
    by fastforce
  hence unique_parent: ‹unique_parent tree›
    by (meson apply_ops_unique_parent)
  have distinct_times: ‹distinct ((map log_time log) @ [t1, t2])›
    using assms apply_ops apply_ops_timestamps by auto
  have ‹apply_ops (ops @ [Move t1 p1 m1 c1, Move t2 p2 m2 c2]) =
        apply_op (Move t2 p2 m2 c2) (apply_op (Move t1 p1 m1 c1) (log, tree))›
    using apply_ops by (simp add: apply_ops_def)
  also have ‹... = apply_op (Move t1 p1 m1 c1) (apply_op (Move t2 p2 m2 c2) (log, tree))›
  proof(cases ‹t1 < t2›)
    case True
    then show ?thesis
      by (metis unique_parent distinct_times apply_op_commute2)
  next
    case False
    hence ‹t2 < t1›
      using assms by auto
    moreover have ‹distinct ((map log_time log) @ [t2, t1])›
      using distinct_times by auto
    ultimately show ?thesis
      by (metis unique_parent apply_op_commute2)
  qed
  also have ‹... = apply_ops (ops @ [Move t2 p2 m2 c2, Move t1 p1 m1 c1])›
    using apply_ops by (simp add: apply_ops_def)
  ultimately show ?thesis
    by presburger
qed

lemma apply_op_commute_middle:
  assumes ‹distinct (map move_time (xs @ ys @ [oper]))›
  shows ‹apply_ops (xs @ ys @ [oper]) = apply_ops (xs @ [oper] @ ys)›
using assms proof(induction ys rule: List.rev_induct, simp)
  case (snoc y ys)
  have ‹apply_ops (xs @ [oper] @ ys @ [y]) = apply_op y (apply_ops (xs @ [oper] @ ys))›
    by (metis append.assoc apply_ops_step)
  also have ‹... = apply_op y (apply_ops (xs @ ys @ [oper]))›
  proof -
    have ‹distinct (map move_time (xs @ ys @ [oper]))›
      using snoc.prems by auto
    then show ?thesis
      using snoc.IH by auto
  qed
  also have ‹... = apply_ops ((xs @ ys) @ [oper, y])›
    by (metis append.assoc append_Cons append_Nil apply_ops_step)
  also have ‹... = apply_ops ((xs @ ys) @ [y, oper])›
  proof -
    have ‹distinct ((map move_time (xs @ ys)) @ [move_time y, move_time oper])›
      using snoc.prems by auto
    thus ?thesis
      using apply_op_commute_last by (metis operation.exhaust_sel)
  qed
  ultimately show ?case
    by simp
qed

theorem apply_ops_commutes:
  assumes ‹set ops1 = set ops2›
    and ‹distinct (map move_time ops1)›
    and ‹distinct (map move_time ops2)›
  shows ‹apply_ops ops1 = apply_ops ops2›
using assms proof(induction ops1 arbitrary: ops2 rule: List.rev_induct, simp)
  case (snoc oper ops)
  then obtain pre suf where pre_suf: ‹ops2 = pre @ [oper] @ suf›
```

```
    by (metis append_Cons append_Nil in_set_conv_decomp)
  hence ‹set ops = set (pre @ suf)›
    using snoc.prems distinct_map distinct_list_pick1 by metis
  hence IH: ‹apply_ops ops = apply_ops (pre @ suf)›
    using pre_suf snoc.IH snoc.prems by auto
  moreover have ‹distinct (map move_time (pre @ suf @ [oper]))›
    using pre_suf snoc.prems(3) by auto
  moreover from this have ‹apply_ops (pre @ suf @ [oper]) = apply_ops (pre @ [oper] @ suf)›
    using apply_op_commute_middle by blast
  ultimately show ‹apply_ops (ops @ [oper]) = apply_ops ops2›
    by (metis append_assoc apply_ops_step pre_suf)
qed


end
theory Move_Acyclic
  imports Move
begin
```

4 Tree invariant 2: no cycles

```
definition acyclic :: ‹('n × 'm × 'n) set ⇒ bool› where
  ‹acyclic tree ≡ (∄n. ancestor tree n n)›

lemma acyclic_empty [simp]: ‹acyclic {}›
  by (meson acyclic_def ancestor_indcases empty_iff)

lemma acyclicE [elim]:
  assumes ‹acyclic 𝒯›
    and ‹(∄n. ancestor 𝒯 n n) ⟹ P›
  shows ‹P›
  using assms by (auto simp add: acyclic_def)

lemma ancestor_empty_False [simp]:
  shows ‹ancestor {} p c = False›
  by (meson ancestor_indcases emptyE)

lemma ancestor_superset_closed:
  assumes ‹ancestor 𝒯 p c›
    and ‹𝒯 ⊆ 𝒮›
  shows ‹ancestor 𝒮 p c›
  using assms by (induction rule: ancestor.induct) (auto intro: ancestor.intros)

lemma acyclic_subset:
  assumes ‹acyclic T›
    and ‹S ⊆ T›
  shows ‹acyclic S›
  using assms ancestor_superset_closed by (metis acyclic_def)

inductive path :: ‹('n × 'm × 'n) set ⇒ 'n ⇒ 'n ⇒ ('n × 'n) list ⇒ bool› where
  ‹⟦(b, x, e) ∈ T⟧ ⟹ path T b e [(b, e)]› |
  ‹⟦path T b m xs; (m, e) ∉ set xs; (m, x, e) ∈ T⟧ ⟹ path T b e (xs @ [(m, e)])›

inductive_cases path_indcases: ‹path T b e xs›

lemma empty_path:
  shows ‹¬ path T x y []›
  using path_indcases by fastforce

lemma singleton_path:
  assumes ‹path T b m [(p, c)]›
  shows ‹b = p ∧ m = c›
  using assms by (metis (no_types, lifting) butlast.simps(2) butlast_snoc empty_path
```

```
    list.inject path.cases prod.inject)

lemma last_path:
  assumes ‹path T b e (xs @ [(p, c)])›
  shows ‹e = c›
  using assms path.cases by force

lemma path_drop1:
  assumes ‹path T b e (xs @ [(a, e)])›
    and ‹xs ≠ []›
  shows ‹path T b a xs ∧ (a, e) ∉ set xs›
  using assms path.cases by force

lemma path_drop:
  assumes ‹path T b e (xs @ ys)›
    and ‹xs ≠ []›
  shows ‹∃m. path T b m xs›
using assms proof(induction ys arbitrary: xs, force)
  case (Cons x ys)
  from this obtain m where IH: ‹path T b m (xs @ [x])›
    by fastforce
  moreover obtain a e where ‹x = (a, e)›
    by fastforce
  moreover from this have ‹m = e›
    using IH last_path by fastforce
  ultimately show ?case
    using Cons.prems(2) path_drop1 by fastforce
qed

lemma fst_path:
  assumes ‹path T b e ((p, c) # xs)›
  shows ‹b = p›
using assms proof(induction xs arbitrary: e rule: List.rev_induct)
  case Nil then show ?case
    by (simp add: singleton_path)
next
  case (snoc x xs)
  then show ?case
    by (metis append_Cons list.distinct(1) path_drop)
qed

lemma path_split:
  assumes ‹path T m n xs›
    and ‹(p, c) ∈ set xs›
  shows ‹∃ys zs. (ys = [] ∨ path T m p ys) ∧ (zs = [] ∨ path T c n zs) ∧
                (xs = ys @ [(p, c)] @ zs) ∧ (p, c) ∉ set ys ∧ (p, c) ∉ set zs›
using assms proof(induction rule: path.induct, force)
  case step: (2 T b m xs e)
  then show ?case
  proof(cases ‹(p, c) = (m, e)›)
    case True
    then show ?thesis using step.hyps by force
  next
    case pc_xs: False
    then obtain ys zs where yszs: ‹(ys = [] ∨ path T b p ys) ∧ (zs = [] ∨ path T c m zs) ∧
        xs = ys @ [(p, c)] @ zs ∧ (p, c) ∉ set ys ∧ (p, c) ∉ set zs›
      using step.IH step.prems by auto
    have path_zs: ‹path T c e (zs @ [(m, e)])›
      by (metis (no_types, lifting) Un_iff append_Cons last_path path.simps
          self_append_conv2 set_append step.hyps(1) step.hyps(2) step.hyps(3) yszs)
    then show ?thesis
    proof(cases ‹ys = []›)
      case True
```

```
      hence ‹∃zsa. ([] = [] ∨ path T b p []) ∧ (zsa = [] ∨ path T c e zsa) ∧
              (p, c) # zs @ [(m, e)] = [] @ (p, c) # zsa ∧ (p, c) ∉ set [] ∧ (p, c) ∉ set zsa›
        using pc_xs path_zs yszs by auto
      then show ?thesis
        using yszs by force
    next
      case False
      hence ‹∃zsa. (ys = [] ∨ path T b p ys) ∧ (zsa = [] ∨ path T c e zsa) ∧
              ys @ (p, c) # zs @ [(m, e)] = ys @ (p, c) # zsa ∧ (p, c) ∉ set ys ∧ (p, c) ∉ set zsa›
        using path_zs pc_xs yszs by auto
      then show ?thesis
        using yszs by force
    qed
  qed
qed

lemma anc_path:
  assumes ‹ancestor T p c›
  shows ‹∃xs. path T p c xs›
using assms proof(induction rule: ancestor.induct)
  case (1 parent meta child tree)
  then show ?case by (meson path.intros(1))
next
  case step: (2 parent meta child tree anc)
  then obtain xs where xs: ‹path tree anc parent xs›
    by blast
  then show ?case
  proof(cases ‹(parent, child) ∈ set xs›)
    case True
    then show ?thesis
      by (metis step.hyps(1) xs append_Cons append_Nil fst_path path.intros path_split)
  next
    case False
    then show ?thesis
      by (meson path.intros(2) step.hyps(1) xs)
  qed
qed

lemma path_anc:
  assumes ‹path T p c xs›
  shows ‹ancestor T p c›
using assms by (induction rule: path.induct, auto simp add: ancestor.intros)

lemma anc_path_eq:
  shows ‹ancestor T p c ⟷ (∃xs. path T p c xs)›
  by (meson anc_path path_anc)

lemma acyclic_path_eq:
  shows ‹acyclic T ⟷ (∄n xs. path T n n xs)›
  by (meson anc_path acyclic_def path_anc)


lemma rem_edge_path:
  assumes ‹path T m n xs›
    and ‹T = insert (p, x, c) S›
    and ‹(p, c) ∉ set xs›
  shows ‹path S m n xs›
using assms by (induction rule: path.induct, auto simp add: path.intros)

lemma ancestor_transitive:
  assumes ‹ancestor 𝒮 n p› and ‹ancestor 𝒮 m n›
    shows ‹ancestor 𝒮 m p›
  using assms by (induction rule: ancestor.induct) (auto intro: ancestor.intros)
```

```
lemma cyclic_path_technical:
  assumes ‹path T m m xs›
    and ‹T = insert (p, x, c) S›
    and ‹∀n. ¬ ancestor S n n›
    and ‹c ≠ p›
  shows ‹ancestor S c p›
proof(cases ‹(p, c) ∈ set xs›)
  case True
  then obtain ys zs where yszs: ‹(ys = [] ∨ path T m p ys) ∧ (zs = [] ∨ path T c m zs) ∧
       xs = ys @ [(p, c)] @ zs ∧ (p, c) ∉ set ys ∧ (p, c) ∉ set zs›
    using assms(1) path_split by force
  then show ?thesis
  proof(cases ‹ys = []›)
    case True
    then show ?thesis using assms by (metis append_Cons append_Nil fst_path path_anc
      rem_edge_path singleton_path yszs)
  next
    case False
    then show ?thesis using assms by (metis ancestor_transitive last_path path_anc
      rem_edge_path self_append_conv yszs)
  qed
next
  case False
  then show ?thesis
    using assms by (metis path_anc rem_edge_path)
qed

lemma cyclic_ancestor:
  assumes ‹¬ acyclic (S ∪ {(p, x, c)})›
    and ‹acyclic S›
    and ‹c ≠ p›
  shows ‹ancestor S c p›
using assms anc_path acyclic_def cyclic_path_technical by fastforce

lemma do_op_acyclic:
  assumes ‹acyclic tree1›
    and ‹do_op (Move t newp m c, tree1) = (log_oper, tree2)›
  shows ‹acyclic tree2›
proof(cases ‹ancestor tree1 c newp ∨ c = newp›)
  case True
  then show ‹acyclic tree2›
    using assms by auto
next
  case False
  hence A: ‹tree2 = {(p', m', c') ∈ tree1. c' ≠ c} ∪ {(newp, m, c)}›
    using assms(2) by auto
  moreover have ‹{(p', m', c') ∈ tree1. c' ≠ c} ⊆ tree1›
    by blast
  moreover have ‹acyclic tree1›
    using assms and acyclic_def by auto
  moreover have B: ‹acyclic {(p', m', c') ∈ tree1. c' ≠ c}›
    using acyclic_subset calculation(2) calculation(3) by blast
  {
    assume ‹¬ acyclic tree2›
    hence ‹ancestor {(p', m', c') ∈ tree1. c' ≠ c} c newp›
      using cyclic_ancestor False A B by force
    from this have ‹False›
      using False ancestor_superset_closed calculation(2) by fastforce
  }
  from this show ‹acyclic tree2›
    using acyclic_def by auto
qed
```

```
lemma do_op_acyclic_var:
  assumes ‹acyclic tree1›
    and ‹do_op (oper, tree1) = (log_oper, tree2)›
  shows ‹acyclic tree2›
  using assms by (metis do_op_acyclic operation.exhaust_sel)

lemma redo_op_acyclic_var:
  assumes ‹acyclic tree1›
    and ‹redo_op (LogMove t oldp p m c) (log1, tree1) = (log2, tree2)›
  shows ‹acyclic tree2›
  using assms by (subst (asm) redo_op.simps) (rule do_op_acyclic, assumption, fastforce)

corollary redo_op_acyclic:
  assumes ‹acyclic tree1›
    and ‹redo_op logop (log1, tree1) = (log2, tree2)›
  shows ‹acyclic tree2›
  using assms by (cases logop) (metis redo_op_acyclic_var)

inductive steps :: ‹(('t, 'n, 'm) log_op list × ('n × 'm × 'n) set) list ⇒ bool› where
  ‹⟦do_op (oper, {}) = (logop, tree)⟧ ⟹ steps [([logop], tree)]› |
  ‹⟦steps (ss @ [(log, tree)]); do_op (oper, tree) = (logop, tree2)⟧ ⟹ steps (ss @ [(log, tree),
(logop # log, tree2)])›

inductive_cases steps_indcases [elim]: ‹steps ss›
inductive_cases steps_singleton_indcases [elim]: ‹steps [s]›
inductive_cases steps_snoc_indcases [elim]: ‹steps (ss@[s])›

lemma steps_empty [elim]:
  assumes ‹steps (ss @ [([], tree)])›
  shows ‹False›
  using assms by force

lemma steps_snocI:
  assumes ‹steps (ss @ [(log, tree)])›
      and ‹do_op (oper, tree) = (logop, tree2)›
      and ‹suf = [(log, tree), (logop # log, tree2)]›
    shows ‹steps (ss @ suf)›
  using assms steps.intros(2) by blast

lemma steps_unique_parent:
  assumes ‹steps ss›
  and ‹ss = ss'@[(log, tree)]›
  shows ‹unique_parent tree›
  using assms by(induction arbitrary: ss' log tree rule: steps.induct)
    (clarsimp, metis do_op_unique_parent emptyE operation.exhaust_sel unique_parentI)+


lemma apply_op_steps_exist:
  assumes ‹apply_op oper (log1, tree1) = (log2, tree2)›
    and ‹steps (ss@[(log1, tree1)])›
  shows ‹∃ss'. steps (ss'@[(log2,tree2)])›
using assms proof(induction log1 arbitrary: tree1 log2 tree2 ss)
  case Nil
  thus ?case using steps_empty by blast
next
  case (Cons logop ops)
  { assume ‹move_time oper < log_time logop›
    hence *:‹apply_op oper (logop # ops, tree1) =
            redo_op logop (apply_op oper (ops, undo_op (logop, tree1)))›
      by simp
    moreover {
      fix oper'
```

```
      assume asm: ‹do_op (oper', {}) = (logop, tree1)› ‹ss = []› ‹(logop # ops, tree1) = ([logop], 
tree1)›
      hence undo: ‹undo_op (logop, tree1) = {}›
        using asm Cons by (metis apply_ops_Nil apply_ops_unique_parent do_op.cases do_undo_op_inv 
old.prod.inject)
      obtain t oldp p m c where logmove: ‹logop = LogMove t oldp p m c›
        using log_op.exhaust by blast
      obtain logop'' tree'' where do: ‹do_op (oper, {}) = (logop'', tree'')›
        by fastforce
      hence redo: ‹redo_op logop ([logop''], tree'') = (log2, tree2)›
        using Cons.prems(1) asm undo calculation by auto
      then obtain op2 where op2: ‹do_op (Move t p m c, tree'') = (op2, tree2)›
        by (simp add: logmove)
      hence log2: ‹log2 = op2 # [logop'']›
        using logmove redo by auto
      have ‹steps ([] @ [([logop''], tree''), (op2 #  [logop''], tree2)])›
        using do op2 by (fastforce intro: steps.intros)
      hence ‹steps ([([logop''], tree'')] @ [(log2, tree2)])›
        by (simp add: log2)
      hence ‹∃ss'. steps (ss' @ [(log2, tree2)])›
        by fastforce
    } moreover {
      fix pre_ss tree' oper'
      assume asm: ‹steps (pre_ss @ [(ops, tree')])›
                  ‹do_op (oper', tree') = (logop, tree1)›
                  ‹ss = pre_ss @ [(ops, tree')]›
      hence undo: ‹undo_op (logop, tree1) = tree'›
        using do_undo_op_inv_var steps_unique_parent by metis
      obtain log'' tree'' where apply_op: ‹apply_op oper (ops, undo_op (logop, tree1)) = (log'', 
tree'')›
        by (meson surj_pair)
      moreover have ‹steps (pre_ss @ [(ops, undo_op (logop, tree1))])›
        by (simp add: undo asm)
      ultimately obtain ss' where ss': ‹steps (ss' @ [(log'', tree'')])›
        using Cons.IH by blast
      obtain t oldp p m c where logmove: ‹logop = LogMove t oldp p m c›
        using log_op.exhaust by blast
      hence redo: ‹redo_op logop (log'', tree'') = (log2, tree2)›
        using Cons.prems(1) * apply_op by auto
      then obtain op2 where op2: ‹do_op (Move t p m c, tree'') = (op2, tree2)›
        using logmove redo by auto
      hence log2: ‹log2 = op2 # log''›
        using logmove redo by auto
      hence ‹steps (ss' @ [(log'', tree''), (op2 # log'', tree2)])›
        using ss' op2 by (fastforce intro!: steps.intros)
      hence ‹steps ((ss' @ [(log'', tree'')]) @ [(log2, tree2)])›
        by (simp add: log2)
      hence ‹∃ss'. steps (ss' @ [(log2, tree2)])›
        by blast
    } ultimately have ‹∃ss'. steps (ss' @ [(log2, tree2)])›
      using Cons by auto
  } moreover {
    assume ‹¬ (move_time oper < log_time logop)›
    hence ‹apply_op oper (logop # ops, tree1) =
           (let (op2, tree2) = do_op (oper, tree1) in (op2 # logop # ops, tree2))›
      by simp
    moreover then obtain logop2 where ‹do_op (oper, tree1) = (logop2, tree2)›
      by (metis (mono_tags, lifting) Cons.prems(1) case_prod_beta' prod.collapse snd_conv)
    moreover hence ‹steps (ss @ [(logop # ops, tree1), (logop2 # logop # ops, tree2)])›
      using Cons.prems(2) steps_snocI by blast
    ultimately have ‹∃ss'. steps (ss' @ [(log2, tree2)])›
      using Cons by (metis (mono_tags) Cons_eq_appendI append_eq_appendI append_self_conv2 insert_Nil
        prod.sel(1) prod.sel(2) rotate1.simps(2) split_beta)
```

```
  } ultimately show ?case
    by auto
qed


lemma last_helper:
  assumes ‹last xs = x› ‹xs ≠ []›
  shows   ‹∃pre. xs = pre @ [x]›
  using assms by (induction xs arbitrary: x rule: rev_induct; simp)

lemma steps_exist:
  fixes log :: ‹('t::{linorder}, 'n, 'm) log_op list›
  assumes ‹apply_ops ops = (log, tree)› and ‹ops ≠ []›
  shows ‹∃ss. steps ss ∧ last ss = (log, tree)›
using assms proof(induction ops arbitrary: log tree rule: List.rev_induct, simp)
  case (snoc oper ops)
  then show ?case
  proof (cases ops)
    case Nil
    moreover obtain op2 tree2 where ‹do_op (oper, {}) = (op2, tree2)›
      by fastforce
    moreover have ‹apply_ops (ops @ [oper]) = (let (op2, tree2) = do_op (oper, {}) in ([op2], tree2))›
      by (metis apply_op.simps(1) apply_ops_Nil apply_ops_step calculation)
    moreover have ‹log = [op2]› ‹tree = tree2›
      using calculation(2) calculation(3) snoc.prems(1) by auto
    ultimately have ‹steps [(log, tree)]›
      using steps.simps  by auto
    then show ?thesis
      by force
  next
    case (Cons a list)

    obtain log1 tree1 where ‹apply_ops ops = (log1, tree1)›
      by fastforce
    moreover from this obtain ss where ‹steps ss ∧ (last ss) = (log1, tree1) ∧ ss ≠ []›
      using snoc.IH Cons by blast
    moreover then obtain pre_ss where ‹steps (pre_ss @ [(log1, tree1)]) ›
      using last_helper by fastforce
    moreover have ‹apply_op oper (log1, tree1) = (log, tree)›
      using calculation(1) snoc.prems(1) by auto
    ultimately obtain ss' where ‹steps (ss' @ [(log, tree)])›
      using apply_op_steps_exist by blast
    then show ?thesis
      by force
  qed
qed

lemma steps_remove1:
  assumes ‹steps (ss @ [s])›
  shows ‹steps ss ∨ ss = []›
using assms steps.cases by fastforce

lemma steps_singleton:
  assumes ‹steps [s]›
  shows ‹∃oper. let (logop, tree) = do_op (oper, {}) in s = ([logop], tree)›
  using assms steps_singleton_indcases
  by (metis (mono_tags, lifting) case_prodI)

lemma steps_acyclic:
  assumes ‹steps ss›
  shows ‹acyclic (snd (last ss))›
  using assms apply (induction rule: steps.induct; clarsimp)
   apply (metis acyclic_empty do_op_acyclic operation.exhaust_sel)
```

```
  using do_op_acyclic_var by auto

theorem apply_ops_acyclic:
  fixes ops :: ‹('t::{linorder}, 'n, 'm) operation list›
  assumes ‹apply_ops ops = (log, tree)›
  shows ‹acyclic tree›
proof(cases ‹ops = []›)
  case True
  then show ‹acyclic tree›
    using acyclic_def assms by fastforce
next
  case False
  then obtain ss :: ‹(('t, 'n, 'm) log_op list × ('n × 'm × 'n) set) list›
      where ‹steps ss ∧ snd (last ss) = tree›
    using assms steps_exist
    by (metis snd_conv)
  then show ‹acyclic tree›
    using steps_acyclic by blast
qed

end
theory Move_SEC
  imports Move CRDT.Network
begin
```

5 Strong eventual consistency

```
definition apply_op' :: ‹('t::{linorder}, 'n, 'm) operation ⇒ ('t, 'n, 'm) state ⇀ ('t, 'n, 'm) state›
where
  ‹apply_op' x s ≡ case s of (log, tree) ⇒
    if unique_parent tree ∧ distinct (map log_time log @ [move_time x]) then
      Some (apply_op x s)
    else None›

fun valid_move_opers :: ‹('t, 'n, 'm) state ⇒ 't ×('t, 'n, 'm) operation ⇒ bool› where
  ‹valid_move_opers _ (i, Move t _ _ _) = (i = t)›

locale move = network_with_constrained_ops _ apply_op' ‹([], {})› valid_move_opers
begin

lemma kleisli_apply_op' [iff]:
  shows ‹apply_op' (x :: ('t :: {linorder}, 'n, 'm) operation) ▷ apply_op' y = apply_op' y ▷ apply_op'
x›
proof (unfold kleisli_def, rule ext, clarify)
  fix log :: ‹('t, 'n, 'm) log_op list› and tree :: ‹('n × 'm × 'n) set›
  { assume *: ‹unique_parent tree› ‹distinct (map log_time log @ [move_time x])› ‹distinct (map log_time
log @ [move_time y])› ‹move_time x ≠ move_time y›
    obtain logx treex where 1: ‹apply_op x (log, tree) = (logx, treex)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    hence ‹set (map log_time logx) = {move_time x} ∪ set (map log_time log)›
      using * by (cases x) (rule apply_op_timestampI2; force)
    moreover have ‹distinct (map log_time logx)›
      using * 1 by (cases x) (rule apply_op_timestampI1; force)
    ultimately have 2: ‹distinct (map log_time logx @ [move_time y])›
      using * by simp
    obtain logy treey where 3: ‹apply_op y (log, tree) = (logy, treey)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    hence ‹set (map log_time logy) = {move_time y} ∪ set (map log_time log)›
      using * by (cases y) (rule apply_op_timestampI2; force)
    moreover have ‹distinct (map log_time logy)›
      using * 3 by (cases y) (rule apply_op_timestampI1, force, force)
    ultimately have 4: ‹distinct (map log_time logy @ [move_time x])›
      using * by simp
```

```
    have ‹unique_parent treex› ‹unique_parent treey›
      using * 1 3 apply_op_unique_parent by blast+
    hence ‹apply_op' x (log, tree) ≫= apply_op' y = apply_op' y (log, tree) ≫= apply_op' x›
      using * 1 2 3 4 by (cases x, cases y, clarsimp simp: apply_op'_def) (rule apply_op_commute2I;
force)
  }
  moreover {
    assume *: ‹unique_parent tree› ‹distinct (map log_time log @ [move_time x])› ‹distinct (map log_time
log @ [move_time y])› ‹move_time x = move_time y›
    obtain logx treex where 1: ‹apply_op x (log, tree) = (logx, treex)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    hence ‹set (map log_time logx) = {move_time x} ∪ set (map log_time log)›
      using * by (cases x) (rule apply_op_timestampI2; force)
    hence 2: ‹¬ distinct (map log_time logx @ [move_time y])›
      using * by simp
    obtain logy treey where 3: ‹apply_op y (log, tree) = (logy, treey)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    hence ‹ set (map log_time logy) = {move_time y} ∪ set (map log_time log)›
      using * by (cases y) (rule apply_op_timestampI2; force)
    hence 4: ‹¬ distinct (map log_time logy @ [move_time x])›
      using * by simp
    have ‹apply_op' x (log, tree) ≫= apply_op' y = apply_op' y (log, tree) ≫= apply_op' x›
      using * 1 2 3 4 by (clarsimp simp: apply_op'_def)
  }
  moreover {
    assume *: ‹unique_parent tree› ‹¬ distinct (map log_time log @ [move_time x])› ‹distinct (map
log_time log @ [move_time y])›
    then have **: ‹move_time x ∈ set (map log_time log)›
      by auto
    obtain log1 tree1 where ‹apply_op y (log, tree) = (log1, tree1)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    moreover hence ‹ set (map log_time log1) = {move_time y} ∪ set (map log_time log)›
      using * by (cases y) (rule apply_op_timestampI2; force)
    hence ‹move_time x ∈ set (map log_time log1)›
      using ** by blast
    moreover hence ‹¬ distinct (map log_time log1 @ [move_time x])›
      by simp
    ultimately have ‹apply_op' x (log, tree) ≫= apply_op' y = apply_op' y (log, tree) ≫= apply_op'
x›
      using * by (clarsimp simp: apply_op'_def)
  }
  moreover {
    assume *: ‹unique_parent tree› ‹distinct (map log_time log @ [move_time x])› ‹¬ distinct (map
log_time log @ [move_time y])›
    then have **: ‹move_time y ∈ set (map log_time log)›
      by auto
    obtain log1 tree1 where ‹apply_op x (log, tree) = (log1, tree1)›
      using * by (clarsimp simp: apply_op'_def)  (metis surj_pair)
    moreover hence ‹ set (map log_time log1) = {move_time x} ∪ set (map log_time log)›
      using * by (cases x) (rule apply_op_timestampI2; force)
    hence ‹move_time y ∈ set (map log_time log1)›
      using ** by blast
    moreover hence ‹¬ distinct (map log_time log1 @ [move_time y])›
      by simp
    ultimately have ‹apply_op' x (log, tree) ≫= apply_op' y = apply_op' y (log, tree) ≫= apply_op'
x›
      using * by (clarsimp simp: apply_op'_def)
  }
  ultimately show ‹apply_op' x (log, tree) ≫= apply_op' y = apply_op' y (log, tree) ≫= apply_op'
x›
    by (clarsimp simp: apply_op'_def) fastforce
qed
```

```
lemma concurrent_operations_commute:
  assumes ‹xs prefix of i›
  shows ‹hb.concurrent_ops_commute (node_deliver_messages xs)›
  using assms by (clarsimp simp add: hb.concurrent_ops_commute_def) (unfold interp_msg_def; simp)

corollary apply_operations_Snoc2:
  ‹hb.apply_operations (xs @ [x]) s = (hb.apply_operations xs ▷ interp_msg x) s›
  using hb.apply_operations_Snoc by auto

lemma unique_parent_empty[simp]:
  shows ‹unique_parent {}›
  by (auto simp: unique_parent_def)

lemma log_tree_invariant:
  assumes ‹xs prefix of i›  ‹apply_operations xs = Some (log, tree)›
  shows   ‹distinct (map log_time log) ∧ unique_parent tree›
using assms proof (induct xs arbitrary: log tree rule: rev_induct, clarsimp)
  case (snoc x xs)
  hence ‹apply_operations xs ≠ None›
    by (case_tac x; clarsimp simp: apply_operations_def node_deliver_messages_append kleisli_def)
       (metis (no_types, hide_lams) bind_eq_Some_conv surj_pair)
  then obtain log1 tree1 where *: ‹apply_operations xs = Some (log1, tree1)›
    by auto
  moreover have ‹xs prefix of i›
    using snoc.prems(1) by blast
  ultimately have **: ‹distinct (map log_time log1)› ‹unique_parent tree1›
    using snoc.hyps by blast+
  show ?case
  proof (case_tac x)
    fix m assume ‹x = Broadcast m›
    hence ‹apply_operations (xs @ [x]) = apply_operations xs›
      by simp
    thus ‹distinct (map log_time log) ∧ unique_parent tree›
      using ‹xs prefix of i› snoc.hyps snoc.prems(2) by presburger
  next
    fix m assume 1: ‹x = Deliver m›
    obtain t oper where 2: "m = (t, oper)"
      by force
    hence ‹interp_msg (t, oper) (log1, tree1) = Some (log, tree)›
      using ‹apply_operations xs = Some (log1, tree1)› snoc.prems(2) 1 2 by simp
    hence 4: ‹apply_op' oper (log1, tree1) = Some (log, tree)›
      by (clarsimp simp: interp_msg_def apply_op'_def)
    hence ‹distinct ((map log_time log1) @ [move_time oper])›
      by (clarsimp simp: apply_op'_def) (meson option.distinct(1))
    moreover hence 5: ‹apply_op oper (log1, tree1) = (log, tree)›
      using 4 ** by (clarsimp simp: apply_op'_def)
    ultimately have ‹distinct (map log_time log)›
      by (case_tac oper, clarsimp) (rule apply_op_timestampI1, assumption, clarsimp)
    thus ‹distinct (map log_time log) ∧ unique_parent tree›
      using ** 5 apply_op_unique_parent by blast
  qed
qed

definition indices :: "('id × ('id, 'v, 'm) operation) event list ⇒ 'id list" where
  ‹indices xs ≡ List.map_filter (λx. case x of Deliver (i, _) ⇒ Some i | _ ⇒ None) xs›

lemma indices_Nil [simp]:
  shows ‹indices [] = []›
by(auto simp: indices_def map_filter_def)

lemma indices_append [simp]:
  shows ‹indices (xs@ys) = indices xs @ indices ys›
by(auto simp: indices_def map_filter_def)
```

```
lemma indices_Broadcast_singleton [simp]:
  shows ⟨indices [Broadcast b] = []⟩
by(auto simp: indices_def map_filter_def)

lemma indices_Deliver_Insert [simp]:
  shows ⟨indices [Deliver (i, x)] = [i]⟩
  by(auto simp: indices_def map_filter_def)

lemma idx_in_elem[intro]:
  assumes ⟨Deliver (i, x) ∈ set xs⟩
  shows   ⟨i ∈ set (indices xs)⟩
using assms by(induction xs, auto simp add: indices_def map_filter_def)

lemma valid_move_oper_delivered:
  assumes ⟨xs@[Deliver (t, oper)] prefix of i⟩
  shows   ⟨move_time oper = t⟩
by (metis assms deliver_in_prefix_is_valid in_set_conv_decomp operation.set_cases(1) operation.set_sel(1)
valid_move_opers.simps)

find_theorems "apply_operations (?xs @ [?x])"

lemma apply_opers_idx_elems:
  assumes ⟨xs prefix of i⟩ ⟨apply_operations xs = Some (log, tree)⟩
  shows   ⟨set (map log_time log) = set (indices xs)⟩
using assms proof (induction xs arbitrary: log tree rule: rev_induct, force)
  case (snoc x xs)
  moreover have prefix: ⟨xs prefix of i⟩
    using snoc by force
  ultimately show ?case
  proof (cases x, force)
    case (Deliver m)
    then obtain t oper where m: ⟨m = (t, oper)⟩
      by fastforce
    from Deliver and snoc show ?thesis
    proof (cases ⟨apply_operations xs⟩, force)
      case (Some st)
      then obtain log' tree' where st: ⟨st = (log', tree')⟩
        by (meson surj_pair)
      have set_indices: ⟨log_time ` set log' = set (indices xs)⟩
        using Some prefix snoc.IH st by auto
      hence *:⟨unique_parent tree' ∧ distinct (map log_time log')⟩
        using st Some prefix by (simp add: log_tree_invariant)
      hence **: ⟨apply_operations (xs @ [x]) =
             (if move_time oper ∉ set (indices xs) then Some (apply_op (snd (t, oper)) (log', tree'))
             else None)⟩
        using Deliver Some st m set_indices by (auto simp: interp_msg_def apply_op'_def)
      hence ***: ⟨move_time oper ∉ set (indices xs)⟩
        using snoc.prems(2) by auto
      obtain t' p m c where oper: ⟨oper = Move t' p m c⟩
        using operation.exhaust by blast
      hence ⟨t = t'⟩
        using valid_move_oper_delivered Deliver m snoc.prems(1) by fastforce
      hence ⟨apply_op (Move t p m c) (log', tree') = (log, tree)⟩
        by (metis ** oper option.discI option.simps(1) prod.sel(2) snoc.prems(2))
      hence ⟨set (map log_time log) = {t} ∪ set (map log_time log')⟩
        apply (rule apply_op_timestampI2)
        using Deliver * *** m set_indices snoc.prems(1) valid_move_oper_delivered by auto
      thus ?thesis
        using Deliver m set_indices by (clarsimp simp: interp_msg_def apply_op'_def)
    qed
  qed
qed
```

```
lemma indices_distinct_aux:
  assumes ‹xs @ [Deliver (a, b)] prefix of i›
    shows ‹a ∉ set (indices xs)›
proof
  have 1: ‹xs prefix of i›
    using assms by force
  assume ‹a ∈ set (indices xs)›
  hence ‹∃x. Deliver (a, x) ∈ set xs›
    by (clarsimp simp: indices_def map_filter_def, case_tac x; force)
  then obtain c where 2: ‹Deliver (a, c) ∈ set xs›
    by auto
  moreover then obtain j where ‹Broadcast (a, c) ∈ set (history j)›
    using 1 delivery_has_a_cause prefix_elem_to_carriers by blast
  moreover obtain k where ‹Broadcast (a, b) ∈ set (history k)›
    by (meson assms delivery_has_a_cause in_set_conv_decomp prefix_elem_to_carriers)
  ultimately have ‹b = c›
    by (metis fst_conv network.msg_id_unique network_axioms old.prod.inject)
  hence ‹¬ distinct (xs @ [Deliver (a, b)])›
    by (simp add: 2)
  thus ‹False›
    using assms prefix_distinct by blast
qed


lemma indices_distinct:
  assumes ‹xs prefix of i›
  shows   ‹distinct (indices xs)›
using assms proof (induct xs rule: rev_induct, clarsimp)
  case (snoc x xs)
  hence ‹xs prefix of i›
    by force
  moreover hence ‹distinct (indices xs)›
    by (simp add: snoc.hyps)
  ultimately show ?case
    using indices_distinct_aux snoc.prems by (case_tac x; force)
qed

lemma log_time_invariant:
  assumes ‹xs@[Deliver (t, oper)] prefix of i›  ‹apply_operations xs = Some (log, tree)›
  shows   ‹move_time oper ∉ set (map log_time log)›
proof -
  have ‹xs prefix of i›
    using assms by force
  have ‹move_time oper = t›
    using assms valid_move_oper_delivered by auto
  moreover have ‹indices (xs @ [Deliver (t, oper)]) = indices xs @ [t]›
    by simp
  moreover have ‹distinct (indices (xs @ [Deliver (t, oper)]))›
    using assms indices_distinct by blast
  ultimately show ?thesis
    using apply_opers_idx_elems assms indices_distinct_aux by blast
qed

lemma apply_operations_never_fails:
  assumes ‹xs prefix of i›
  shows   ‹apply_operations xs ≠ None›
using assms proof(induct xs rule: rev_induct, clarsimp)
  case (snoc x xs)
  hence ‹apply_operations xs ≠ None›
    by blast
  then obtain log1 tree1 where *: ‹apply_operations xs = Some (log1, tree1)›
    by auto
```

```
  moreover hence ‹distinct (map log_time log1) ∧ unique_parent tree1›
    using log_tree_invariant snoc.prems by blast
  ultimately show ?case
    using log_time_invariant snoc.prems
    by (cases x; clarsimp simp: interp_msg_def) (clarsimp simp: apply_op'_def)
qed

sublocale sec: strong_eventual_consistency weak_hb hb interp_msg
  ‹λos. ∃xs i. xs prefix of i ∧ node_deliver_messages xs = os› ‹([], {})›
proof (standard; clarsimp)
  fix xsa i
  assume ‹xsa prefix of i›
  thus ‹hb.hb_consistent (node_deliver_messages xsa)›
    by(auto simp add: hb_consistent_prefix)
next
  fix xsa i
  assume ‹xsa prefix of i›
  thus ‹distinct (node_deliver_messages xsa)›
    by(auto simp add: node_deliver_messages_distinct)
next
  fix xsa i
  assume ‹xsa prefix of i›
  thus ‹hb.concurrent_ops_commute (node_deliver_messages xsa)›
    by(auto simp add: concurrent_operations_commute)
next
  fix xs a b state xsa x
  assume ‹hb.apply_operations xs ([], {}) = Some state›
         ‹node_deliver_messages xsa = xs @ [(a, b)]›
         ‹xsa prefix of x›
  moreover hence ‹apply_operations xsa ≠ None›
    using apply_operations_never_fails by blast
  ultimately show ‹∃ab bb. interp_msg (a, b) state = Some (ab, bb)›
    by (clarsimp simp: apply_operations_def kleisli_def)
next
  fix xs a b xsa x
  assume ‹node_deliver_messages xsa = xs @ [(a, b)]›
    and  ‹xsa prefix of x›
  thus ‹∃xsa. (∃x. xsa prefix of x) ∧ node_deliver_messages xsa = xs›
    using drop_last_message by blast
qed

end

end
theory
  Move_Code
imports
  Move Move_Acyclic "HOL-Library.Code_Target_Numeral" "Collections.Collections"
    "HOL-Library.Product_Lexorder"
begin
```

6 Code generation: an executable implementation

```
inductive ancestor_alt :: ‹('n × 'm × 'n) set ⇒ 'n ⇒ 'n ⇒ bool›
  where ‹get_parent T c = Some (p, m) ⟹ ancestor_alt T p c›
      | ‹get_parent T c = Some (p, m) ⟹ ancestor_alt T a p ⟹ ancestor_alt T a c›

lemma get_parent_SomeI [intro]:
  assumes ‹unique_parent T›
    and ‹(p, m, c) ∈ T›
  shows ‹get_parent T c = Some (p, m)›
using assms by(auto simp add: get_parent_def)
```

```
lemma get_parent_SomeD:
  assumes 1: ‹get_parent T c = Some (p, m)›
    and 2: ‹unique_parent T›
  shows ‹(p, m, c) ∈ T›
proof -
  {
    assume 3: ‹∃!parent. ∃!meta. (parent, meta, c) ∈ T›
    from this have ‹get_parent T c = Some (THE (parent, meta). (parent, meta, c) ∈ T)›
      by(auto simp add: get_parent_def)
    from this and 1 have ‹(THE (parent, meta). (parent, meta, c) ∈ T) = (p, m)›
      by force
    from this and 1 and 2 and 3 have ‹(p, m, c) ∈ T›
      using get_parent_SomeI by fastforce
  }
  note L = this
  {
    assume ‹¬ (∃!parent. ∃!meta. (parent, meta, c) ∈ T)›
    from this have ‹get_parent T c = None›
      by(auto simp add: get_parent_def)
    from this and 1 have ‹(p, m, c) ∈ T›
      by simp
  }
  from this and L show ?thesis
    by blast
qed

lemma get_parent_NoneD:
  assumes ‹get_parent T c = None›
    and ‹unique_parent T›
    and ‹(p, m, c) ∈ T›
  shows ‹False›
using assms by(clarsimp simp add: get_parent_def unique_parent_def split: if_split_asm; fastforce)

lemma get_parent_NoneI:
  assumes ‹unique_parent T›
    and ‹⋀p m. (p, m, c) ∉ T›
  shows ‹get_parent T c = None›
using assms by(clarsimp simp add: unique_parent_def get_parent_def)

lemma ancestor_ancestor_alt:
  assumes ‹ancestor T p c› and ‹unique_parent T›
    shows ‹ancestor_alt T p c›
using assms by(induction rule: ancestor.induct; force intro: ancestor_alt.intros)

lemma ancestor_alt_ancestor:
  assumes ‹ancestor_alt T p c› and ‹unique_parent T›
    shows ‹ancestor T p c›
using assms by(induction rule: ancestor_alt.induct; force dest: get_parent_SomeD intro: ancestor.intros)

theorem ancestor_ancestor_alt_iff [simp]:
  assumes ‹unique_parent T›
  shows ‹ancestor T p c ⟷ ancestor_alt T p c›
using assms ancestor_ancestor_alt ancestor_alt_ancestor by metis

lemma unique_parent_emptyI [intro!]:
  shows ‹unique_parent {}›
  by(auto simp add: unique_parent_def)

lemma unique_parent_singletonI [intro!]:
  shows ‹unique_parent {x}›
  by(auto simp add: unique_parent_def)

definition simulates :: ‹('n::{hashable}, 'm × 'n) hm ⇒ ('n × 'm × 'n) set ⇒ bool› (infix "≼" 50)
```

```
  where ‹simulates Rs Ss ⟷
          (∀p m c. hm.lookup c Rs = Some (m, p) ⟷ (p, m, c) ∈ Ss)›

lemma simulatesI [intro!]:
  assumes ‹⋀p m c. hm.lookup c Rs = Some (m, p) ⟹ (p, m, c) ∈ Ss›
    and ‹⋀p m c. (p, m, c) ∈ Ss ⟹ hm.lookup c Rs = Some (m, p)›
  shows ‹Rs ≼ Ss›
using assms unfolding simulates_def by meson

lemma weak_simulatesE:
  assumes ‹Rs ≼ Ss›
    and ‹(⋀p m c. hm.lookup c Rs = Some (m, p) ⟹ (p, m, c) ∈ Ss) ⟹ (⋀p m c. (p, m, c) ∈ Ss
⟹ hm.lookup c Rs = Some (m, p)) ⟹ P›
  shows P
using assms by(auto simp add: simulates_def)

lemma simulatesE [elim]:
  assumes ‹Rs ≼ Ss›
    and ‹(⋀p m c. (hm.lookup c Rs = Some (m, p)) ⟷ (p, m, c) ∈ Ss) ⟹ P›
  shows P
using assms by(auto simp add: simulates_def)

lemma empty_simulatesI [intro!]:
  shows ‹hm.empty () ≼ {}›
  by(auto simp add: hm.correct)

lemma get_parent_refinement_Some1:
  assumes ‹get_parent T c = Some (p, m)›
    and ‹unique_parent T›
    and ‹t ≼ T›
    shows ‹hm.lookup c t = Some (m, p)›
using assms by (force dest: get_parent_SomeD)

lemma get_parent_refinement_Some2:
  assumes ‹hm.lookup c t = Some (m, p)›
    and ‹unique_parent T›
    and ‹t ≼ T›
    shows ‹get_parent T c = Some (p, m)›
using assms by (force dest: get_parent_SomeI)

lemma get_parent_refinement_None1:
  assumes ‹get_parent T c = None›
    and ‹unique_parent T›
    and ‹t ≼ T›
  shows ‹hm.lookup c t = None›
proof -
  have ‹∀p m. (p, m, c) ∉ T›
    using assms by (force dest: get_parent_NoneD)
  thus ?thesis
    using assms by (force dest: get_parent_NoneD)
qed

lemma get_parent_refinement_None2:
  assumes ‹hm.lookup c t = None›
    and ‹unique_parent T›
    and ‹t ≼ T›
    shows ‹get_parent T c = None›
using assms by(force intro: get_parent_NoneI)

corollary get_parent_refinement:
  fixes T :: ‹('a::{hashable} × 'b × 'a) set›
  assumes ‹unique_parent T› and ‹t ≼ T›
  shows ‹get_parent T c = map_option (λx. (snd x, fst x)) (hm.lookup c t)›
```

```
proof (cases ‹get_parent T c›)
  case None
  then show ?thesis
    using assms by (cases ‹hm.lookup c t›; force simp: get_parent_refinement_None1)
next
  case (Some a)
  then show ?thesis
    using assms get_parent_SomeI by (cases ‹hm.lookup c t›, simp add: get_parent_refinement_None2,
fastforce)
qed

lemma set_member_refine:
  assumes ‹(p, m, c) ∈ T›
    and ‹t ≼ T›
  shows ‹hm.lookup c t = Some (m, p)›
using assms by blast

lemma ancestor_alt_simp1:
  fixes t :: ‹('n::{hashable}, 'm × 'n) hm›
  assumes ‹ancestor_alt T p c› and ‹t ≼ T› and ‹unique_parent T›
    shows ‹(case hm.lookup c t of
              None ⇒ False
            | Some (m, a) ⇒
                a = p ∨ ancestor_alt T p a)›
using assms
proof(induction rule: ancestor_alt.induct)
  case (1 T c p m)
  then show ?case by(force dest: get_parent_refinement_Some1)
next
  case (2 T c p m a)
  then show ?case by(force dest: get_parent_SomeD)
qed

lemma ancestor_alt_simp2:
  assumes ‹(case hm.lookup c t of
              None ⇒ False
            | Some (m, a) ⇒
                a = p ∨ ancestor_alt T p a)›
    and ‹t ≼ T› and ‹unique_parent T›
  shows ‹ancestor_alt T p c›
using assms by(clarsimp split: option.split_asm; force intro: ancestor_alt.intros)

theorem ancestor_alt_simp [simp]:
  fixes t :: ‹('n::{hashable}, 'm × 'n) hm›
  assumes ‹t ≼ T› and ‹unique_parent T›
  shows ‹ancestor_alt T p c ⟷
          (case hm.lookup c t of
              None ⇒ False
            | Some (m, a) ⇒
                a = p ∨ ancestor_alt T p a)›
using assms ancestor_alt_simp1 ancestor_alt_simp2 by blast

definition flip_triples :: ‹('a × 'b × 'a) list ⇒ ('a × 'b × 'a) list›
  where ‹flip_triples xs ≡ map (λ(x, y, z). (z, y, x)) xs›

definition executable_ancestor :: ‹('n::{hashable}, 'm × 'n) hm ⇒ 'n ⇒ 'n ⇒ bool›
  where ‹executable_ancestor t p c ⟷ ancestor_alt (set (flip_triples (hm.to_list t))) p c›

lemma to_list_simulates:
  shows ‹t ≼ set (flip_triples (hm.to_list t))›
proof
  fix p m c
  assume *: ‹hm.lookup c t = Some (m, p)›
```

```
  have ‹hm_invar t›
    by auto
  from this have ‹map_of (hm.to_list t) = hm.α t›
    by(auto simp add: hm.to_list_correct)
  moreover from this have ‹map_of (hm.to_list t) c = Some (m, p)›
    using * by(clarsimp simp add: hm.lookup_correct)
  from this have ‹(c, m, p) ∈ set (hm.to_list t)›
    using map_of_SomeD by metis
  from this show ‹(p, m, c) ∈ set (flip_triples (hm.to_list t))›
    by(force simp add: flip_triples_def intro: rev_image_eqI)
next
  fix p m c
  assume ‹(p, m, c) ∈ set (flip_triples (hm.to_list t))›
  from this have ‹(c, m, p) ∈ set (hm.to_list t)›
    by(force simp add: flip_triples_def)
  from this have ‹map_of (hm.to_list t) c = Some (m, p)›
    by (force intro:  map_of_is_SomeI hm.to_list_correct)+
  from this show ‹hm.lookup c t = Some (m, p)›
    by(auto simp add: hm.to_list_correct hm.lookup_correct)
qed

lemma unique_parent_to_list:
  shows ‹unique_parent (set (flip_triples (hm.to_list t)))›
  by(unfold unique_parent_def, intro allI impI conjI, elim conjE)
    (clarsimp simp add: flip_triples_def; (drule map_of_is_SomeI[rotated], force simp add: hm.to_list_correct)+

theorem executable_ancestor_simp [code]:
  shows ‹executable_ancestor t p c ⟷
          (case hm.lookup c t of
              None ⇒ False
            | Some (m, a) ⇒
                a = p ∨ executable_ancestor t p a)›
  by (unfold executable_ancestor_def)
      (auto simp: executable_ancestor_def intro!: ancestor_alt_simp unique_parent_to_list to_list_simulates)


fun executable_do_op :: ‹('t, 'n, 'm) operation × ('n::{hashable}, 'm × 'n) hm ⇒
        ('t, 'n, 'm) log_op × ('n::{hashable}, 'm × 'n) hm›
  where ‹executable_do_op (Move t newp m c, tree) =
           (LogMove t (map_option (λx. (snd x, fst x)) (hm.lookup c tree)) newp m c,
              if executable_ancestor tree c newp ∨ c = newp then tree
                else hm.update c (m, newp) tree)›

fun executable_undo_op :: ‹('t, 'n, 'm) log_op × ('n::{hashable}, 'm × 'n) hm ⇒ ('n, 'm × 'n) hm›
  where ‹executable_undo_op (LogMove t None newp m c, tree) =
          hm.delete c tree›
      | ‹executable_undo_op (LogMove t (Some (oldp, oldm)) newp m c, tree) =
          hm.update c (oldm, oldp) tree›

fun executable_redo_op :: ‹('t, 'n, 'm) log_op ⇒
            ('t, 'n, 'm) log_op list × ('n::{hashable}, 'm × 'n) hm ⇒
            ('t, 'n, 'm) log_op list × ('n, 'm × 'n) hm›
  where ‹executable_redo_op (LogMove t _ p m c) (ops, tree) =
          (let (op2, tree2) = executable_do_op (Move t p m c, tree) in
             (op2#ops, tree2))›

fun executable_apply_op :: ‹('t::{linorder}, 'n, 'm) operation ⇒
              ('t, 'n, 'm) log_op list × ('n::{hashable}, 'm × 'n) hm ⇒
            ('t, 'n, 'm) log_op list × ('n, 'm × 'n) hm›
  where ‹executable_apply_op op1 ([], tree1) =
          (let (op2, tree2) = executable_do_op (op1, tree1)
            in ([op2], tree2))›
      | ‹executable_apply_op op1 (logop#ops, tree1) =
```

```
        (if move_time op1 < log_time logop
          then executable_redo_op logop (executable_apply_op op1 (ops, executable_undo_op (logop,
tree1)))
            else let (op2, tree2) = executable_do_op (op1, tree1) in (op2 # logop # ops, tree2))⟩

definition executable_apply_ops :: ⟨('t::{linorder}, 'n::{hashable}, 'm) operation list ⇒
      ('t, 'n, 'm) log_op list × ('n::{hashable}, 'm × 'n) hm⟩
  where ⟨executable_apply_ops ops ≡
      foldl (λstate oper. executable_apply_op oper state) ([], (hm.empty ())) ops⟩
```

Any abstract set that is simulated by a hash-map must necessarily have the `unique_parent` property:

```
lemma simulates_unique_parent:
  assumes ⟨t ⪯ T⟩ shows ⟨unique_parent T⟩
using assms unfolding unique_parent_def
proof(intro allI impI, elim conjE)
  fix p1 p2 m1 m2 c
  assume ⟨(p1, m1, c) ∈ T⟩ and ⟨(p2, m2, c) ∈ T⟩
  from this have ⟨hm.lookup c t = Some (m1, p1)⟩ and ⟨hm.lookup c t = Some (m2, p2)⟩
    using assms by(auto simp add: simulates_def)
  from this show ⟨p1 = p2 ∧ m1 = m2⟩
    by force
qed
```

`hm.delete` is in relation with an explicit restrict operation on sets:

```
lemma hm_delete_refine:
  assumes ⟨t ⪯ T⟩ and ⟨S = {(p', m', c') ∈ T. c' ≠ child}⟩
  shows ⟨hm.delete child t ⪯ S⟩
using assms by(auto simp add: hm.lookup_correct hm.delete_correct restrict_map_def split!: if_split_asm)
```

`hm.restrict` is in relation with an explicit restrict operation on sets:

```
lemma hm_restrict_refine:
  assumes ⟨t ⪯ T⟩ and ⟨S = { x∈T. (P ∘ (λ(x, y, z). (z, y, x))) x }⟩
  shows ⟨hm.restrict P t ⪯ S⟩
using assms by(auto simp add: hm.lookup_correct hm.restrict_correct restrict_map_def
    simulates_unique_parent unique_parent_def split!: if_split_asm if_split)
```

`hm.update` is in relation with an explicit update operation on sets:

```
lemma hm_update_refine:
  assumes ⟨t ⪯ T⟩ and ⟨S = { (p, m, c) ∈ T. c≠x } ∪ {(z, y, x)}⟩
  shows ⟨hm.update x (y, z) t ⪯ S⟩
using assms by(auto simp add: hm.update_correct hm.lookup_correct simulates_unique_parent split:
if_split_asm)
```

Two if-then-else constructs are in relation if both of their branches are in relation:

```
lemma if_refine:
  assumes ⟨x ⟹ t ⪯ T⟩ and ⟨¬ x ⟹ u ⪯ U⟩ and ⟨x ⟷ y⟩
  shows ⟨(if x then t else u) ⪯ (if y then T else U)⟩
using assms by(case_tac x; clarsimp)
```

The `ancestor` relation can be extended "one step downwards" using `get_parent`:

```
lemma ancestor_get_parent_extend:
  assumes ⟨ancestor T a p⟩ and ⟨unique_parent T⟩
    and ⟨get_parent T c = Some (p, m)⟩
  shows ⟨ancestor T a c⟩
using assms proof(induction arbitrary: c m rule: ancestor.induct)
  case (1 parent meta child tree)
  assume 1: ⟨(parent, meta, child) ∈ tree⟩ and ⟨unique_parent tree⟩
    and ⟨get_parent tree c = Some (child, m)⟩
  from this have ⟨(child, m, c) ∈ tree⟩
    by(force simp add: unique_parent_def dest: get_parent_SomeD)
  from this and 1 show ?case
    by(blast intro: ancestor.intros)
```

```
next
  case (2 parent meta child tree anc)
  assume 1: ‹(parent, meta, child) ∈ tree› and 2: ‹unique_parent tree›
    and ‹get_parent tree c = Some (child, m)›
    and IH: ‹⋀c m. unique_parent tree ⟹ get_parent tree c = Some (parent, m) ⟹ ancestor tree
anc c›
  from this have ‹(child, m, c) ∈ tree›
    by(force dest: get_parent_SomeD)
  from this and 1 and 2 and IH show ?case
    by(blast intro: ancestor.intros(2) IH get_parent_SomeI)
qed
```

The executable and abstract `ancestor` relations agree for all ancestry queries between a prospective ancestor and child node when applied to related states:

```
lemma executable_ancestor_simulates:
  assumes ‹t ≼ T›
  shows ‹executable_ancestor t p c = ancestor T p c›
using assms proof(intro iffI)
  assume 1: ‹executable_ancestor t p c›
    and 2: ‹t ≼ T›
  obtain u where 3: ‹u = set (flip_triples (hm.to_list t))›
    by force
  from this and 1 have ‹ancestor_alt u p c›
    by(force simp add: executable_ancestor_def)
  from this and 2 and 3 show ‹ancestor T p c›
  proof(induction rule: ancestor_alt.induct)
    case (1 T' c p m)
    assume ‹get_parent T' c = Some (p, m)› and ‹T' = set (flip_triples (hm.to_list t))›
    from this have ‹(p, m, c) ∈ set (flip_triples (hm.to_list t))›
      by(force dest: get_parent_SomeD intro: unique_parent_to_list)
    from this have ‹(p, m, c) ∈ T›
      using ‹t ≼ T› by(force simp add: hm.correct hm.to_list_correct simulates_def
                flip_triples_def dest: map_of_is_SomeI[rotated])
    then show ?case
      by(force intro: ancestor.intros)
  next
    case (2 T' c p m a)
    assume 1: ‹get_parent T' c = Some (p, m)›
      and IH: ‹t ≼ T ⟹ T' = set (flip_triples (hm.to_list t)) ⟹ ancestor T a p›
      and 2: ‹t ≼ T› and 3: ‹T' = set (flip_triples (hm.to_list t))›
    from this have 4: ‹ancestor T a p›
      by auto
    from this have ‹(p, m, c) ∈ set (flip_triples (hm.to_list t))›
      using 1 and 3 by(auto dest!: get_parent_SomeD simp add: unique_parent_to_list)
    from this have ‹(c, m, p) ∈ set (hm.to_list t)›
      by(auto simp add: flip_triples_def)
    from this and 2 have ‹get_parent T c = Some (p, m)›
      by(auto intro!: get_parent_SomeI simulates_unique_parent[OF 2]
          simp add: hm.correct hm.to_list_correct dest!: map_of_is_SomeI[rotated])
    from this and 2 and 4 show ?case
      by(auto intro!: ancestor_get_parent_extend[OF 4] simulates_unique_parent)
  qed
next
  assume ‹ancestor T p c› and ‹t ≼ T›
  from this show ‹executable_ancestor t p c›
    by(induction rule: ancestor.induct) (force simp add: executable_ancestor_simp)+
qed

lemma executable_do_op_get_parent_technical:
  assumes 1: ‹t ≼ T›
  shows ‹map_option (λx. (snd x, fst x)) (hm.lookup c t) = get_parent T c›
using assms proof(cases ‹hm.lookup c t›)
  assume 2: ‹hm.lookup c t = None›
```

```
  from this have ‹map_option (λx. (snd x, fst x)) (hm.lookup c t) = None›
    by force
  moreover have ‹... = get_parent T c›
    using 1 2 get_parent_NoneI simulates_unique_parent by(metis option.simps(3) set_member_refine)
  finally show ?thesis by force
next
  fix a :: ‹'b × 'a›
  assume 2: ‹hm.lookup c t = Some a›
  {
    fix p :: 'a and m :: 'b
    assume 3: ‹a = (m, p)›
    from this and 1 and 2 have ‹(p, m, c) ∈ T›
      by auto
    moreover from 2 and 3 have ‹map_option (λx. (snd x, fst x)) (hm.lookup c t) = Some (p, m)›

      by auto
    moreover have ‹get_parent T c = Some (p, m)›
      using 1 calculation simulates_unique_parent get_parent_SomeI by auto
    ultimately have ‹map_option (λx. (snd x, fst x)) (hm.lookup c t) = get_parent T c›
      by simp
  }
  from this show ?thesis
    using prod.exhaust by blast
qed
```

The unique_parent predicate is “downward-closed” in the sense that all subsets of a set with the unique_parent property also possess this property:

```
lemma unique_parent_downward_closure:
  assumes ‹unique_parent T›
    and ‹S ⊆ T›
  shows ‹unique_parent S›
using assms by(force simp add: unique_parent_def)
```

The following is a technical lemma needed to establish the result that immediately follows:

```
lemma hm_update_refine_collapse:
  assumes ‹t ≼ T› and ‹unique_parent T›
  shows ‹hm.update child (meta, parent) t ≼
          insert (parent, meta, child) {(p, m, c). (p, m, c) ∈ T ∧ c ≠ child}›
using assms by(force simp add: hm.correct hm.update_correct hm.restrict_correct
        simulates_def unique_parent_def split!: if_split_asm)
```

The executable and abstract do_op algorithms map related concrete and abstract states to related concrete and abstract states, and produce identical logs, when fed the same operation:

```
lemma executable_do_op_simulates:
  assumes 1: ‹t ≼ T›
    and 2: ‹executable_do_op (oper, t) = (log1, u)›
    and 3: ‹do_op (oper, T) = (log2, U)›
  shows ‹log1 = log2 ∧ u ≼ U›
using assms proof(cases ‹oper›)
  case (Move time parent meta child)
  assume 4: ‹oper = Move time parent meta child›
  {
    assume 5: ‹executable_ancestor t child parent ∨ parent = child›
    from this and 1 have 6: ‹ancestor T child parent ∨ parent = child›
      using executable_ancestor_simulates by auto
    from 4 and 5 have ‹executable_do_op (oper, t) =
        (LogMove time (map_option (λx. (snd x, fst x)) (hm.lookup child t)) parent meta child, t)›
      by force
    moreover from 4 and 5 and 6 have ‹do_op (oper, T) =
        (LogMove time (get_parent T child) parent meta child, T)›
      by force
    moreover from 2 have ‹log1 = LogMove time (map_option (λx. (snd x, fst x)) (hm.lookup child
t)) parent meta child›
```

```
      and ‹u = t›
      using calculation by auto
    moreover from 3 have ‹log2 = LogMove time (get_parent T child) parent meta child› and ‹U = T›
      using calculation by auto
    ultimately have ‹log1 = log2 ∧ u ≼ U›
      using 1 by(auto simp add: executable_do_op_get_parent_technical)
  }
  note L = this
  {
    assume 5: ‹¬ (executable_ancestor t child parent ∨ parent = child)›
    from this and 1 have 6: ‹¬ (ancestor T child parent ∨ parent = child)›
      using executable_ancestor_simulates by auto
    from 4 and 5 have ‹executable_do_op (oper, t) =
      (LogMove time (map_option (λx. (snd x, fst x)) (hm.lookup child t)) parent meta child,
          hm.update child (meta, parent) t)›
      by auto
    moreover from 4 and 5 and 6 have ‹do_op (oper, T) =
        (LogMove time (get_parent T child) parent meta child,
          {(p, m, c) ∈ T. c ≠ child} ∪ {(parent, meta, child)})›
      by auto
    moreover from 2 have ‹log1 = LogMove time (map_option (λx. (snd x, fst x)) (hm.lookup child
t)) parent meta child›
        and ‹u = hm.update child (meta, parent) t›
      using calculation by auto
    moreover from 3 have ‹log2 = LogMove time (get_parent T child) parent meta child› and
          ‹U = {(p, m, c) ∈ T. c ≠ child} ∪ {(parent, meta, child)}›
      using calculation by auto
    ultimately have ‹log1 = log2 ∧ u ≼ U›
      using 1 by(clarsimp simp add: executable_do_op_get_parent_technical hm_update_refine_collapse
            simulates_unique_parent)
  }
  from this and L show ?thesis
    by auto
qed
```

The executable and abstract `redo_op` functins take related concrete and abstract states and produce identical logics and related concrete and abstract states:

```
lemma executable_redo_op_simulates:
  assumes 1: ‹t ≼ T›
    and 2: ‹executable_redo_op oper (opers, t) = (log1, u)›
    and 3: ‹redo_op oper (opers, T) = (log2, U)›
  shows ‹log1 = log2 ∧ u ≼ U›
proof(cases oper)
  case (LogMove time opt_old_parent new_parent meta child)
    assume 4: ‹oper = LogMove time opt_old_parent new_parent meta child›
    obtain entry1 and v where ‹executable_do_op (Move time new_parent meta child, t) = (entry1,
v)›
      by auto
    moreover obtain entry2 and V where ‹do_op (Move time new_parent meta child, T) = (entry2, V)›
      by auto
    moreover have 5: ‹entry1 = entry2› and 6: ‹v ≼ V›
      using calculation executable_do_op_simulates[OF 1] by blast+
    from 4 have ‹executable_redo_op oper (opers, t) = (entry1#opers, v)›
      using calculation by clarsimp
    moreover have ‹log1 = entry1#opers› and ‹u = v›
      using 2 calculation by auto
    moreover from 4 have ‹redo_op oper (opers, T) = (entry2#opers, V)›
      using calculation by simp
    moreover have ‹log2 = entry2#opers› and ‹U = V›
      using 3 calculation by auto
    ultimately show ‹?thesis›
      using 5 6 by metis
qed
```

The executable and abstract versions of `undo_op` map related concrete and abstract states to related concrete and abstract states when applied to the same operation:

```
lemma executable_undo_op_simulates:
  assumes 1: ‹t ≼ T›
  shows ‹executable_undo_op (oper, t) ≼ undo_op (oper, T)›
using assms proof(cases ‹oper›)
  case (LogMove time opt_old_parent new_parent meta child)
    assume 2: ‹oper = LogMove time opt_old_parent new_parent meta child›
    {
      assume ‹opt_old_parent = None›
      from this and 2 have 3: ‹oper = LogMove time None new_parent meta child›
        by simp
      moreover from this have ‹executable_undo_op (oper, t) = hm.delete child t›
        by force
      moreover have ‹... ≼ {(p', m', c') ∈ T. c' ≠ child}›
        by(rule hm_delete_refine[OF 1]) auto
      moreover have ‹... = undo_op (oper, T)›
        using 3 by force
      ultimately have ?thesis
        by metis
    }
    note L = this
    {
      fix old_meta old_parent
      assume ‹opt_old_parent = Some (old_parent, old_meta)›
      from this and 2 have 3: ‹oper = LogMove time (Some (old_parent, old_meta)) new_parent meta
child›
        by simp
      moreover from this have ‹executable_undo_op (oper, t) =
          hm.update child (old_meta, old_parent) t›
        by auto
      moreover have ‹... ≼ {(p, m, c) ∈ T. c ≠ child} ∪ {(old_parent, old_meta, child)}›
        by(rule hm_update_refine, rule 1, force)
      moreover have ‹... = undo_op (oper, T)›
        using 3 by auto
      ultimately have ?thesis
        by metis
    }
    from this and L show ‹?thesis›
      by(cases opt_old_parent) force+
qed
```

The executable and abstract `apply_op` algorithms map related concrete and abstract states to related concrete and abstract states when applied to the same operation and input log, and also produce identical output logs:

```
lemma executable_apply_op_simulates:
  assumes ‹t ≼ T›
    and ‹executable_apply_op oper (log, t) = (log1, u)›
    and ‹apply_op oper (log, T) = (log2, U)›
  shows ‹log1 = log2 ∧ u ≼ U›
using assms proof(induction log arbitrary: T t log1 log2 u U)
  case Nil
  assume 1: ‹t ≼ T› and 2: ‹executable_apply_op oper ([], t) = (log1, u)›
    and 3: ‹apply_op oper ([], T) = (log2, U)›
  obtain action1 action2 t' T' where 4: ‹executable_do_op (oper, t) = (action1, t')›
      and 5: ‹do_op (oper, T) = (action2, T')›
    by fastforce
  moreover from 4 and 5 have ‹action1 = action2› and ‹t' ≼ T'›
    using executable_do_op_simulates[OF 1] by blast+
  moreover from 2 and 4 have ‹log1 = [action1]› and ‹u = t'›
    by auto
  moreover from 3 and 5 have ‹log2 = [action2]› and ‹U = T'›
    by auto
```

```
  ultimately show ?case
    by auto
next
  case (Cons logop logops)
  assume 1: ‹t ≼ T› and 2: ‹executable_apply_op oper (logop # logops, t) = (log1, u)›
    and 3: ‹apply_op oper (logop # logops, T) = (log2, U)›
    and IH: ‹(⋀T t log1 log2 u U. t ≼ T ⟹ executable_apply_op oper (logops, t) = (log1, u) ⟹
              apply_op oper (logops, T) = (log2, U) ⟹ log1 = log2 ∧ u ≼ U)›
  {
    assume 4: ‹move_time oper < log_time logop›
    obtain action1 and action1' and u' and u'' and u''' where 5: ‹executable_undo_op (logop, t)
= u'› and
        6: ‹executable_apply_op oper (logops, u') = (action1, u'')› and
          7: ‹executable_redo_op logop (action1, u'') = (action1', u''')›
      by force
    obtain action2 and action2' and U' and U'' and U''' where 8: ‹undo_op (logop, T) = U'› and
        9: ‹apply_op oper (logops, U') = (action2, U'')› and
          10: ‹redo_op logop (action2, U'') = (action2', U''')›
      by force
    from 5 and 8 have ‹u' ≼ U'›
      using executable_undo_op_simulates[OF 1] by blast
    moreover from 6 and 9 have ‹action1 = action2› and ‹u'' ≼ U''›
      using IH[OF ‹u' ≼ U'›] by blast+
    moreover from this and 7 and 10 have ‹action1' = action2'› and ‹u''' ≼ U'''›
      using executable_redo_op_simulates by blast+
    moreover from 2 and 4 and 5 and 6 and 7 have ‹log1 = action1'› and ‹u = u'''›
      by auto
    moreover from 3 and 4 and 8 and 9 and 10 have ‹log2 = action2'› and ‹U = U'''›
      by auto
    ultimately have ?case
      by auto
  }
  note L = this
  {
    assume 4: ‹¬ (move_time oper < log_time logop)›
    obtain action1 action2 u' U' where 5: ‹executable_do_op (oper, t) = (action1, u')›
        and 6: ‹do_op (oper, T) = (action2, U')›
      by fastforce
    from this have ‹action1 = action2› and ‹u' ≼ U'›
      using executable_do_op_simulates[OF 1] by blast+
    moreover from 2 and 4 and 5 have ‹log1 = action1#logop#logops› and ‹u' = u›
      by auto
    moreover from 3 and 4 and 6 have ‹log2 = action2#logop#logops› and ‹U' = U›
      by auto
    ultimately have ?case
      using 1 by simp
  }
  from this and L show ?case
    by auto
qed
```

The internal workings of abstract and concrete implementations of the `apply_ops` function map related states to related states, and produce identical logs, when passed identical lists of actions to perform.

Note this lemma is necessary as the `apply_ops` function specifies a particular starting state (the empty state) to start the iterated application of the `apply_op` function from, meaning that an inductive proof using this definition directly becomes impossible, as the inductive hypothesis will be over constrained in the step case. By introducing this lemma, we show that the required property holds for any starting states (as long as they are related by the simulation relation) and then specialise to the empty starting state in the next lemma, below.

```
lemma executable_apply_ops_simulates_internal:
  assumes ‹foldl (λstate oper. executable_apply_op oper state) (log, t) xs = (log1, u)›
    and ‹foldl (λstate oper. apply_op oper state) (log, T) xs = (log2, U)›
```

```
    and ‹t ≼ T›
  shows ‹log1 = log2 ∧ u ≼ U›
using assms proof(induction xs arbitrary: log log1 log2 t T u U)
  case Nil
  assume ‹foldl (λstate oper. executable_apply_op oper state) (log, t) [] = (log1, u)›
    and ‹apply_ops' [] (log, T) = (log2, U)›
    and *: ‹t ≼ T›
  from this have ‹log = log2› and ‹T = U› and ‹log = log1› and ‹t = u›
    by auto
  from this show ‹log1 = log2 ∧ u ≼ U›
    using * by auto
next
  case (Cons x xs)
  fix xs :: ‹('a, 'b, 'c) operation list› and x log log1 log2 t T u U
  assume IH: ‹⋀log log1 log2 t T u U.
           foldl (λstate oper. executable_apply_op oper state) (log, t) xs = (log1, u) ⟹
           apply_ops' xs (log, T) = (log2, U) ⟹ t ≼ T ⟹ log1 = log2 ∧ u ≼ U›
    and 1: ‹foldl (λstate oper. executable_apply_op oper state) (log, t) (x#xs) = (log1, u)›
    and 2: ‹apply_ops' (x#xs) (log, T) = (log2, U)›
    and 3: ‹t ≼ T›
  obtain log1' log2' U' u' where 4: ‹executable_apply_op x (log, t) = (log1', u')›
      and 5: ‹apply_op x (log, T) = (log2', U')›
    by fastforce
  moreover from this have ‹log1' = log2'› and ‹u' ≼ U'›
    using executable_apply_op_simulates[OF 3] by blast+
  moreover have ‹foldl (λstate oper. executable_apply_op oper state) (log1', u') xs = (log1, u)›
    using 1 and 4 by simp
  moreover have ‹apply_ops' xs (log2', U') = (log2, U)›
    using 2 and 5 by simp
  ultimately show ‹log1 = log2 ∧ u ≼ U›
    by(auto simp add: IH)
qed
```

The executable and abstract versions of `apply_ops` produce identical operation logs and produce related concrete and abstract states:

```
lemma executable_apply_ops_simulates:
  assumes 1: ‹executable_apply_ops opers = (log1, u)›
    and 2: ‹apply_ops opers = (log2, U)›
  shows ‹log1 = log2 ∧ u ≼ U›
proof -
  have ‹hm.empty () ≼ {}›
    by auto
  moreover have ‹foldl (λstate oper. executable_apply_op oper state) ([], hm.empty ()) opers = (log1,
u)›
    using 1 by(auto simp add: executable_apply_ops_def)
  moreover have ‹foldl (λstate oper. apply_op oper state) ([], {}) opers = (log2, U)›
    using 2 by(auto simp add: apply_ops_def)
  moreover have ‹log1 = log2› and ‹u ≼ U›
    using calculation executable_apply_ops_simulates_internal by blast+
  ultimately show ‹?thesis›
    by auto
qed
```

The `executable_apply_ops` algorithm maintains an acyclic invariant similar to its abstract counterpart, namely that no node in the resulting tree hash-map is its own ancestor:

```
theorem executable_apply_ops_acyclic:
  assumes 1: ‹executable_apply_ops ops = (log, t)›
  shows ‹∄n. executable_ancestor t n n›
using assms proof(intro notI)
  assume ‹∃n. executable_ancestor t n n›
  from this obtain log2 T n where ‹apply_ops ops = (log2, T)› and ‹executable_ancestor t n n›
    by force
  moreover from this and 1 have ‹log = log2› and ‹t ≼ T›
```

```
    using executable_apply_ops_simulates by blast+
  moreover have ‹∄n. ancestor T n n›
    using apply_ops_acyclic calculation by force
  moreover have ‹ancestor T n n›
    using calculation executable_ancestor_simulates by blast
  ultimately show False
    by auto
qed
```

The main correctness theorem for the executable algorithms. This follows the ⟦set ?ops1.0 = set ?ops2.0; distinct (map move_time ?ops1.0); distinct (map move_time ?ops2.0)⟧ ⟹ apply_ops ?ops1.0 = apply_ops ?ops2.0 theorem for the abstract algorithms with one significant difference: the states obtained from interpreting the two lists of operations, ops1 and ops2, are no longer identical (the hash-maps may have a different representation in memory, for instance), but contain the same set of key-value bindings. If we take equality of finite maps (hash-maps included) to be extensional—i.e. two finite maps are equal when they contain the same key-value bindings—then this theorem coincides exactly with the ⟦set ?ops1.0 = set ?ops2.0; distinct (map move_time ?ops1.0); distinct (map move_time ?ops2.0)⟧ ⟹ apply_ops ?ops1.0 = apply_ops ?ops2.0:

```
theorem executable_apply_ops_commutes:
  assumes 1: ‹set ops1 = set ops2›
    and 2: ‹distinct (map move_time ops1)›
    and 3: ‹distinct (map move_time ops2)›
    and 4: ‹executable_apply_ops ops1 = (log1, t)›
    and 5: ‹executable_apply_ops ops2 = (log2, u)›
  shows ‹log1 = log2 ∧ hm.lookup c t = hm.lookup c u›
proof -
  from 1 2 3 have ‹apply_ops ops1 = apply_ops ops2›
    using apply_ops_commutes by auto
  from this obtain log1' log2' T U where 6: ‹apply_ops ops1 = (log1', T)›
      and 7: ‹apply_ops ops2 = (log2', U)› and 8: ‹log1' = log2'› and 9: ‹T = U›
    by fastforce
  moreover from 4 5 6 7 have ‹log1 = log1'› and ‹log2 = log2'› and ‹t ≼ T› and ‹u ≼ U›
    using executable_apply_ops_simulates by force+
  moreover from 8 have ‹log1 = log2›
    by(simp add: calculation)
  moreover have ‹hm.lookup c t = hm.lookup c u›
    using calculation by(cases ‹hm.lookup c t›; cases ‹hm.lookup c u›) (force simp add: simulates_def)+
  ultimately show ‹?thesis›
    by auto
qed
```

Testing code generation

Check that all of the executable algorithms produce executable code for all of Isabelle/HOL's code generation targets (Standard ML, Scala, OCaml, Haskell). Note that the Isabelle code generation mechanism recursively extracts all necessary material from the HOL library required to successfully compile our own definitions, here. As a result, the first section of each extraction is material extracted from the Isabelle libraries—our material is towards the bottom. (View it in the Output buffer of the Isabelle/JEdit IDE.)

```
export_code executable_ancestor executable_do_op executable_undo_op executable_redo_op
  executable_apply_op executable_apply_ops in SML file generated.SML
export_code executable_ancestor executable_do_op executable_undo_op executable_redo_op
  executable_apply_op executable_apply_ops in Scala file generated.scala
export_code executable_ancestor executable_do_op executable_undo_op executable_redo_op
  executable_apply_op executable_apply_ops in OCaml file generated.ml
export_code executable_ancestor executable_do_op executable_undo_op executable_redo_op
  executable_apply_op executable_apply_ops in Haskell

definition integer_apply_op ::
   ‹((integer × integer), integer, String.literal) operation ⇒
    ((integer × integer), integer, String.literal) log_op list ×
      (integer,  String.literal × integer) HashMap.hashmap ⇒
    ((integer × integer), integer, String.literal) log_op list ×
```

```
      (integer, String.literal × integer) HashMap.hashmap⟩
where ⟨integer_apply_op = executable_apply_op⟩

definition integer_apply_ops ::
   ⟨((integer × integer), integer, String.literal) operation list ⇒
    ((integer × integer), integer, String.literal) log_op list ×
      (integer, String.literal × integer) HashMap.hashmap⟩
   where ⟨integer_apply_ops = executable_apply_ops⟩
```

The following is an alternative version that uses `String.literal` everywhere, while the version above uses BigInt for nodes and replica identifiers. The versionthat uses strings is approximately 2.5 times slower for `do_op` and 23

```
definition string_apply_op ::
    ⟨((int × String.literal), String.literal, String.literal) operation ⇒
     ((int × String.literal), String.literal, String.literal) log_op list ×
       (String.literal, String.literal × String.literal) HashMap.hashmap ⇒
     ((int × String.literal), String.literal, String.literal) log_op list ×
       (String.literal, String.literal × String.literal) HashMap.hashmap⟩
  where ⟨string_apply_op = executable_apply_op⟩

definition string_apply_ops ::
    ⟨((int × String.literal), String.literal, String.literal) operation list ⇒
     ((int × String.literal), String.literal, String.literal) log_op list ×
       (String.literal, String.literal × String.literal) HashMap.hashmap⟩
  where ⟨string_apply_ops = executable_apply_ops⟩


export_code integer_apply_op integer_apply_ops string_apply_op string_apply_ops
  in Scala module_name generated
  file ⟨evaluation/src/main/scala/Move_Code.scala⟩
```

Without resorting to saving the generated code above to a separate file and feeding them into an SML/Scala/OCaml/Haskell compiler, as appropriate, we can show that this code compiles and executes relatively quickly from within Isabelle itself, by making use of Isabelle's quotations/anti-quotations, and its tight coupling with the underlying PolyML process.

First define a `unit_test` definition that makes use of our `executable_apply_ops` function on a variety of inputs:

```
definition unit_test :: ⟨((nat, nat, nat) log_op list × (nat, nat × nat) HashMap.hashmap) list⟩
  where ⟨unit_test ≡
          [ executable_apply_ops []
          , executable_apply_ops [Move 1 0 0 1]
          , executable_apply_ops [Move 1 0 0 0, Move 3 2 2 2, Move 2 1 1 1]
          ]⟩
```

Then, we can use **ML_val** to ask Isabelle to:

1. Generate executable code for our `unit_test` definition above, using the SML code generation target,
2. Execute this code within the underlying Isabelle/ML process, and display the resulting SML values back to us within the Isabelle/JEdit IDE.

```
ML_val⟨@{code unit_test}⟩
```

Note, there is a slight lag when performing this action as the executable code is first extracted to SML, dynamically compiled, and then the result of the computation reflected back to us. Nevertheless, on a Macbook Pro (2017 edition) this procedure takes 2 seconds, at the most.

end