Formalization of Concurrent Revisions

Roy Overbeek

October 26, 2026

Abstract

Concurrent revisions is a concurrency control model developed by Microsoft Research [1]. It has many interesting properties that distinguish it from other well-known models such as transactional memory. One of these properties is *determinacy*: programs written within the model always produce the same outcome, independent of scheduling activity. The concurrent revisions model has an operational semantics, with an informal proof of determinacy [2]. This document contains an Isabelle/HOL formalization of this semantics and the proof of determinacy. It is part of my master's thesis [3], which describes it in more detail.[1]

Contents

1 Data **4**
- 1.1 Function notations . . . 4
- 1.2 Values, expressions and execution contexts . . . 4
- 1.3 Plugging and decomposing . . . 5
- 1.4 Stores and states . . . 7

2 Occurrences **7**
- 2.1 Definitions . . . 7
  - 2.1.1 Revision identifiers . . . 7
  - 2.1.2 Location identifiers . . . 7
- 2.2 Inference rules . . . 8
  - 2.2.1 Introduction and elimination rules . . . 8
  - 2.2.2 Distributive laws . . . 9
  - 2.2.3 Miscellaneous laws . . . 13

[1]My master's thesis was partially funded by ING, and I would especially like to thank my supervisors Jasmin Blanchette (VU Amsterdam), Robbert van Dalen (ING) and Wan Fokkink (VU Amsterdam) for their useful feedback on this work.

3 Renaming **14**
3.1 Definitions . . . 15
3.2 Introduction rules . . . 16
3.3 Renaming-equivalence . . . 17
3.3.1 Identity . . . 17
3.3.2 Composition . . . 17
3.3.3 Inverse . . . 17
3.3.4 Equivalence . . . 18
3.4 Distributive laws . . . 18
3.4.1 Expression . . . 18
3.4.2 Store . . . 19
3.4.3 Global . . . 19
3.5 Miscellaneous laws . . . 19
3.6 Swaps . . . 19

4 Substitution **21**
4.1 Definition . . . 21
4.2 Trivial model . . . 21
4.3 Example model . . . 22
4.3.1 Preliminaries . . . 22
4.3.2 Definition . . . 22
4.3.3 Proof obligations . . . 23

5 Operational Semantics **24**
5.1 Definition . . . 24
5.2 Introduction lemmas for identifiers . . . 25
5.3 Domain subsumption . . . 26
5.3.1 Definitions . . . 26
5.3.2 The theorem . . . 26
5.4 Relaxed definition of the operational semantics . . . 40

6 Executions **40**
6.1 Generalizing the original transition . . . 41
6.1.1 Definition . . . 41
6.1.2 Closures . . . 41
6.2 Properties . . . 41
6.3 Invariants . . . 42
6.3.1 Inductive invariance . . . 42
6.3.2 Subsumption is invariant . . . 42
6.3.3 Finitude is invariant . . . 45
6.3.4 Reachability is invariant . . . 48

7 Determinacy **49**
7.1 Rule determinism 49
7.2 Strong local confluence 51
7.2.1 Local determinism 51
7.2.2 General principles 52
7.2.3 Case join-epsilon 52
7.2.4 Case join 53
7.2.5 Case local 55
7.2.6 Case new 59
7.2.7 Case fork 61
7.2.8 The theorem 65
7.3 Diagram Tiling 65
7.3.1 Strong local confluence diagrams 65
7.3.2 Mimicking diagrams 66
7.3.3 Strip diagram 71
7.3.4 Confluence diagram 71
7.4 Determinacy 72

1 Data

This theory defines the data types and notations, and some preliminary results about them.

theory *Data*
  imports *Main*
begin

1.1 Function notations

abbreviation $\varepsilon :: 'a \rightharpoonup 'b$ **where**
  $\varepsilon \equiv \lambda x.\ None$

fun *combine* :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ $(\langle\text{-;;-}\rangle\ 20)$ **where**
  $(f\ ;;\ g)\ x = (if\ g\ x = None\ then\ f\ x\ else\ g\ x)$

lemma *dom-combination-dom-union*: $dom\ (\tau;;\tau') = dom\ \tau \cup dom\ \tau'$
  by *auto*

1.2 Values, expressions and execution contexts

datatype $const = Unit \mid F \mid T$

datatype $(RID_V\text{: }'r,\ LID_V\text{: }'l,'v)\ val =$
  $CV\ const$
$\mid\ Var\ 'v$
$\mid\ Loc\ 'l$
$\mid\ Rid\ 'r$
$\mid\ Lambda\ 'v\ ('r,'l,'v)\ expr$
and $(RID_E\text{: }'r,\ LID_E\text{: }'l,'v)\ expr =$
  $VE\ ('r,'l,'v)\ val$
$\mid\ Apply\ ('r,'l,'v)\ expr\ ('r,'l,'v)\ expr$
$\mid\ Ite\ ('r,'l,'v)\ expr\ ('r,'l,'v)\ expr\ ('r,'l,'v)\ expr$
$\mid\ Ref\ ('r,'l,'v)\ expr$
$\mid\ Read\ ('r,'l,'v)\ expr$
$\mid\ Assign\ ('r,'l,'v)\ expr\ ('r,'l,'v)\ expr$
$\mid\ Rfork\ ('r,'l,'v)\ expr$
$\mid\ Rjoin\ ('r,'l,'v)\ expr$

datatype $(RID_C\text{: }'r,\ LID_C\text{: }'l,'v)\ cntxt =$
  $Hole\ (\langle\square\rangle)$
$\mid\ ApplyL_{\mathcal{E}}\ ('r,'l,'v)\ cntxt\ ('r,'l,'v)\ expr$
$\mid\ ApplyR_{\mathcal{E}}\ ('r,'l,'v)\ val\ ('r,'l,'v)\ cntxt$
$\mid\ Ite_{\mathcal{E}}\ ('r,'l,'v)\ cntxt\ ('r,'l,'v)\ expr\ ('r,'l,'v)\ expr$
$\mid\ Ref_{\mathcal{E}}\ ('r,'l,'v)\ cntxt$
$\mid\ Read_{\mathcal{E}}\ ('r,'l,'v)\ cntxt$
$\mid\ AssignL_{\mathcal{E}}\ ('r,'l,'v)\ cntxt\ ('r,'l,'v)\ expr$
$\mid\ AssignR_{\mathcal{E}}\ 'l\ ('r,'l,'v)\ cntxt$
$\mid\ Rjoin_{\mathcal{E}}\ ('r,'l,'v)\ cntxt$

1.3 Plugging and decomposing

fun *plug* :: ('r,'l,'v) *cntxt* ⇒ ('r,'l,'v) *expr* ⇒ ('r,'l,'v) *expr* (**infix** ‹◁› *60*) **where**
  □ ◁ *e* = *e*
| *ApplyL*$_{\mathcal{E}}$ $\mathcal{E}$ *e1* ◁ *e* = *Apply* ($\mathcal{E}$ ◁ *e*) *e1*
| *ApplyR*$_{\mathcal{E}}$ *val* $\mathcal{E}$ ◁ *e* = *Apply* (*VE val*) ($\mathcal{E}$ ◁ *e*)
| *Ite*$_{\mathcal{E}}$ $\mathcal{E}$ *e1 e2* ◁ *e* = *Ite* ($\mathcal{E}$ ◁ *e*) *e1 e2*
| *Ref*$_{\mathcal{E}}$ $\mathcal{E}$ ◁ *e* = *Ref* ($\mathcal{E}$ ◁ *e*)
| *Read*$_{\mathcal{E}}$ $\mathcal{E}$ ◁ *e* = *Read* ($\mathcal{E}$ ◁ *e*)
| *AssignL*$_{\mathcal{E}}$ $\mathcal{E}$ *e1* ◁ *e* = *Assign* ($\mathcal{E}$ ◁ *e*) *e1*
| *AssignR*$_{\mathcal{E}}$ *l* $\mathcal{E}$ ◁ *e* = *Assign* (*VE* (*Loc l*)) ($\mathcal{E}$ ◁ *e*)
| *Rjoin*$_{\mathcal{E}}$ $\mathcal{E}$ ◁ *e* = *Rjoin* ($\mathcal{E}$ ◁ *e*)

translations
  $\mathcal{E}[x]$ ⇌ $\mathcal{E}$ ◁ *x*

lemma *injective-cntxt* [*simp*]: ($\mathcal{E}$[*e1*] = $\mathcal{E}$[*e2*]) = (*e1* = *e2*) **by** (*induction* $\mathcal{E}$) *auto*

lemma *VE-empty-cntxt* [*simp*]: (*VE v* = $\mathcal{E}$[*e*]) = ($\mathcal{E}$ = □ ∧ *VE v* = *e*) **by** (*cases* $\mathcal{E}$, *auto*)

inductive *redex* :: ('r,'l,'v) *expr* ⇒ *bool* **where**
  app: *redex* (*Apply* (*VE* (*Lambda x e*)) (*VE v*))
| *iteTrue*: *redex* (*Ite* (*VE* (*CV T*)) *e1 e2*)
| *iteFalse*: *redex* (*Ite* (*VE* (*CV F*)) *e1 e2*)
| *ref*: *redex* (*Ref* (*VE v*))
| *read*: *redex* (*Read* (*VE* (*Loc l*)))
| *assign*: *redex* (*Assign* (*VE* (*Loc l*)) (*VE v*))
| *rfork*: *redex* (*Rfork e*)
| *rjoin*: *redex* (*Rjoin* (*VE* (*Rid r*)))

inductive-simps *redex-simps* [*simp*]: *redex e*
inductive-cases *redexE* [*elim*]: *redex e*

lemma *plugged-redex-not-val* [*simp*]: *redex r* ⟹ ($\mathcal{E}$ ◁ *r*) ≠ (*VE t*) **by** (*cases* $\mathcal{E}$) *auto*

inductive *decompose* :: ('r,'l,'v) *expr* ⇒ ('r,'l,'v) *cntxt* ⇒ ('r,'l,'v) *expr* ⇒ *bool* **where**
  top-redex: *redex e* ⟹ *decompose e* □ *e*
| *lapply*: ⟦ ¬*redex* (*Apply* e_1 e_2); *decompose* e_1 $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Apply* e_1 e_2) (*ApplyL*$_{\mathcal{E}}$ $\mathcal{E}$ e_2) *r*
| *rapply*: ⟦ ¬*redex* (*Apply* (*VE v*) *e*); *decompose e* $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Apply* (*VE v*) *e*) (*ApplyR*$_{\mathcal{E}}$ *v* $\mathcal{E}$) *r*
| *ite*: ⟦ ¬*redex* (*Ite* e_1 e_2 e_3); *decompose* e_1 $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Ite* e_1 e_2 e_3) (*Ite*$_{\mathcal{E}}$ $\mathcal{E}$ e_2 e_3) *r*
| *ref*: ⟦ ¬*redex* (*Ref e*); *decompose e* $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Ref e*) (*Ref*$_{\mathcal{E}}$ $\mathcal{E}$) *r*
| *read*: ⟦ ¬*redex* (*Read e*); *decompose e* $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Read e*) (*Read*$_{\mathcal{E}}$ $\mathcal{E}$) *r*
| *lassign*: ⟦ ¬*redex* (*Assign* e_1 e_2); *decompose* e_1 $\mathcal{E}$ *r* ⟧ ⟹ *decompose* (*Assign* e_1 e_2) (*AssignL*$_{\mathcal{E}}$ $\mathcal{E}$ e_2) *r*

| *rassign*: ⟦ ¬*redex* (*Assign* (*VE* (*Loc l*)) e_2); *decompose* e_2 $\mathcal{E}$ r ⟧ ⟹ *decompose* (*Assign* (*VE* (*Loc l*)) e_2) (*AssignR*$_\mathcal{E}$ *l* $\mathcal{E}$) r
| *rjoin*: ⟦ ¬*redex* (*Rjoin e*); *decompose e* $\mathcal{E}$ r ⟧ ⟹ *decompose* (*Rjoin e*) (*Rjoin*$_\mathcal{E}$ $\mathcal{E}$) r

inductive-cases *decomposeE* [*elim*]: *decompose e* $\mathcal{E}$ r

lemma *plug-decomposition-equivalence*: *redex* r ⟹ *decompose e* $\mathcal{E}$ r = ($\mathcal{E}[r]$ = e)
proof (*rule iffI*)
  assume x: *decompose e* $\mathcal{E}$ r
  show $\mathcal{E}[r]$ = e
  proof (*use* x **in** ‹*induct rule*: *decompose.induct*›)
    case (*top-redex e*)
    thus $\square[e]$ = e **by** *simp*
  next
    case (*lapply* e_1 e_2 $\mathcal{E}$ r)
    have (*ApplyL*$_\mathcal{E}$ $\mathcal{E}$ e_2) [r] = *Apply* ($\mathcal{E}[r]$) e_2 **by** *simp*
    also have ... = *Apply* e_1 e_2 **using** ‹$\mathcal{E}[r]$ = e_1› **by** *simp*
    then show *?case* **by** *simp*
  qed *simp+*
next
  assume *red*: *redex* r **and** *eq*: $\mathcal{E}[r]$ = e
 have *decompose* ($\mathcal{E}[r]$) $\mathcal{E}$ r **by** (*induct* $\mathcal{E}$) (*use red* **in** ‹*auto intro*: *decompose.intros*›)
  thus *decompose e* $\mathcal{E}$ r **by** (*simp add*: *eq*)
qed

lemma *unique-decomposition*: *decompose e* $\mathcal{E}_1$ r_1 ⟹ *decompose e* $\mathcal{E}_2$ r_2 ⟹ $\mathcal{E}_1$ = $\mathcal{E}_2$ ∧ r_1 = r_2
  by (*induct arbitrary*: $\mathcal{E}_2$ *rule*: *decompose.induct*) *auto*

lemma *completion-eq* [*simp*]:
  assumes
    red-e: *redex* r **and**
    red-e': *redex* r'
  shows ($\mathcal{E}[r]$ = $\mathcal{E}'[r']$) = ($\mathcal{E}$ = $\mathcal{E}'$ ∧ r = r')
proof (*rule iffI*)
  show $\mathcal{E}[r]$ = $\mathcal{E}'[r']$ ⟹ $\mathcal{E}$ = $\mathcal{E}'$ ∧ r = r'
  proof (*rule conjI*)
    assume *eq*: $\mathcal{E}[r]$ = $\mathcal{E}'[r']$
    have *decompose* ($\mathcal{E}[r]$) $\mathcal{E}$ r **using** *plug-decomposition-equivalence red-e* **by** *blast*
    hence *fst-decomp*:*decompose* ($\mathcal{E}'[r']$) $\mathcal{E}$ r **by** (*simp add*: *eq*)
   have *snd-decomp*: *decompose* ($\mathcal{E}'[r']$) $\mathcal{E}'$ r' **using** *plug-decomposition-equivalence red-e'* **by** *blast*
    show *cntxts-eq*: $\mathcal{E}$ = $\mathcal{E}'$ **using** *fst-decomp snd-decomp unique-decomposition* **by** *blast*
    show r = r' **using** *cntxts-eq eq* **by** *simp*
  qed
qed *simp*

1.4 Stores and states

type-synonym $('r,'l,'v)$ *store* $= 'l \rightharpoonup ('r,'l,'v)$ *val*
type-synonym $('r,'l,'v)$ *local-state* $= ('r,'l,'v)$ *store* $\times$ $('r,'l,'v)$ *store* $\times$ $('r,'l,'v)$ *expr*
type-synonym $('r,'l,'v)$ *global-state* $= 'r \rightharpoonup ('r,'l,'v)$ *local-state*

fun *doms* :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $'l$ *set* **where**
  doms $(\sigma,\tau,e) =$ *dom* $\sigma \cup$ *dom* τ

fun *LID-snapshot* :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $('r,'l,'v)$ *store* $(\langle\text{-}_\sigma\rangle\ 200)$ **where**
  LID-snapshot $(\sigma,\tau,e) = \sigma$

fun *LID-local-store* :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $('r,'l,'v)$ *store* $(\langle\text{-}_\tau\rangle\ 200)$ **where**
  LID-local-store $(\sigma,\tau,e) = \tau$

fun *LID-expression* :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $('r,'l,'v)$ *expr* $(\langle\text{-}_e\rangle\ 200)$ **where**
  LID-expression $(\sigma,\tau,e) = e$

end

2 Occurrences

This theory contains all of the definitions and laws required for reasoning about identifiers that occur in the data structures of the concurrent revisions model.

theory *Occurrences*
  imports *Data*
begin

2.1 Definitions

2.1.1 Revision identifiers

definition RID_S :: $('r,'l,'v)$ *store* $\Rightarrow$ $'r$ *set* **where**
  $RID_S\ \sigma \equiv \bigcup\ (RID_V\ `\ ran\ \sigma)$

definition RID_L :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $'r$ *set* **where**
  $RID_L\ s \equiv$ *case* s *of* $(\sigma,\ \tau,\ e) \Rightarrow RID_S\ \sigma \cup RID_S\ \tau \cup RID_E\ e$

definition RID_G :: $('r,'l,'v)$ *global-state* $\Rightarrow$ $'r$ *set* **where**
  $RID_G\ s \equiv$ *dom* $s \cup \bigcup\ (RID_L\ `\ ran\ s)$

2.1.2 Location identifiers

definition LID_S :: $('r,'l,'v)$ *store* $\Rightarrow$ $'l$ *set* **where**
  $LID_S\ \sigma \equiv$ *dom* $\sigma \cup \bigcup\ (LID_V\ `\ ran\ \sigma)$

definition LID_L :: $('r,'l,'v)$ *local-state* $\Rightarrow$ $'l$ *set* **where**

$LID_L\ s \equiv case\ s\ of\ (\sigma,\ \tau,\ e) \Rightarrow LID_S\ \sigma \cup LID_S\ \tau \cup LID_E\ e$

definition $LID_G :: ('r,'l,'v)\ global\text{-}state \Rightarrow 'l\ set$ **where**
$LID_G\ s \equiv \bigcup\ (LID_L\ `\ ran\ s)$

2.2 Inference rules

2.2.1 Introduction and elimination rules

lemma $RID_S I$ [*intro*]: $\sigma\ l = Some\ v \Longrightarrow r \in RID_V\ v \Longrightarrow r \in RID_S\ \sigma$
by (*auto simp add*: RID_S-*def ran-def*)

lemma $RID_S E$ [*elim*]: $r \in RID_S\ \sigma \Longrightarrow (\bigwedge l\ v.\ \sigma\ l = Some\ v \Longrightarrow r \in RID_V\ v \Longrightarrow P) \Longrightarrow P$
by (*auto simp add*: RID_S-*def ran-def*)

lemma $RID_L I$ [*intro*, *consumes 1*]:
assumes $s = (\sigma,\ \tau,\ e)$
shows
$r \in RID_S\ \sigma \Longrightarrow r \in RID_L\ s$
$r \in RID_S\ \tau \Longrightarrow r \in RID_L\ s$
$r \in RID_E\ e \Longrightarrow r \in RID_L\ s$
by (*auto simp add*: RID_L-*def assms*)

lemma $RID_L E$ [*elim*]:
$[\![\ r \in RID_L\ s;\ (\bigwedge \sigma\ \tau\ e.\ s = (\sigma,\ \tau,\ e) \Longrightarrow (r \in RID_S\ \sigma \Longrightarrow P) \Longrightarrow (r \in RID_S\ \tau \Longrightarrow P) \Longrightarrow (r \in RID_E\ e \Longrightarrow P) \Longrightarrow P) \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*cases s*) (*auto simp add*: RID_L-*def*)

lemma $RID_G I$ [*intro*]:
$s\ r = Some\ v \Longrightarrow r \in RID_G\ s$
$s\ r' = Some\ ls \Longrightarrow r \in RID_L\ ls \Longrightarrow r \in RID_G\ s$
apply (*simp add*: RID_G-*def domI*)
by (*metis* (*no-types*, *lifting*) RID_G-*def UN-I UnI2 ranI*)

lemma $RID_G E$ [*elim*]:
assumes
$r \in RID_G\ s$
$r \in dom\ s \Longrightarrow P$
$\bigwedge r'\ ls.\ s\ r' = Some\ ls \Longrightarrow r \in RID_L\ ls \Longrightarrow P$
shows P
using *assms* **by** (*auto simp add*: RID_G-*def ran-def*)

lemma $LID_S I$ [*intro*]:
$\sigma\ l = Some\ v \Longrightarrow l \in LID_S\ \sigma$
$\sigma\ l' = Some\ v \Longrightarrow l \in LID_V\ v \Longrightarrow l \in LID_S\ \sigma$
by (*auto simp add*: LID_S-*def ran-def*)

lemma $LID_S E$ [*elim*]:
assumes

$l \in LID_S\ \sigma$
$l \in dom\ \sigma \Longrightarrow P$
$\bigwedge l'\ v.\ \sigma\ l' = Some\ v \Longrightarrow l \in LID_V\ v \Longrightarrow P$
shows P
using *assms* **by** (*auto simp add*: LID_S*-def ran-def*)

lemma $LID_L I$ [*intro*]:
assumes $s = (\sigma, \tau, e)$
shows
$r \in LID_S\ \sigma \Longrightarrow r \in LID_L\ s$
$r \in LID_S\ \tau \Longrightarrow r \in LID_L\ s$
$r \in LID_E\ e \Longrightarrow r \in LID_L\ s$
by (*auto simp add*: LID_L*-def assms*)

lemma $LID_L E$ [*elim*]:
$[\![\ l \in LID_L\ s;\ (\bigwedge \sigma\ \tau\ e.\ s = (\sigma, \tau, e) \Longrightarrow (l \in LID_S\ \sigma \Longrightarrow P) \Longrightarrow (l \in LID_S\ \tau \Longrightarrow P) \Longrightarrow (l \in LID_E\ e \Longrightarrow P) \Longrightarrow P) \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*cases s*) (*auto simp add*: LID_L*-def*)

lemma $LID_G I$ [*intro*]: $s\ r = Some\ ls \Longrightarrow l \in LID_L\ ls \Longrightarrow l \in LID_G\ s$
by (*auto simp add*: LID_G*-def* LID_L*-def ran-def*)

lemma $LID_G E$ [*elim*]: $l \in LID_G\ s \Longrightarrow (\bigwedge r\ ls.\ s\ r = Some\ ls \Longrightarrow l \in LID_L\ ls \Longrightarrow P) \Longrightarrow P$
by (*auto simp add*: LID_G*-def ran-def*)

2.2.2 Distributive laws

lemma *ID-distr-completion* [*simp*]:
$RID_E\ (\mathcal{E}[e]) = RID_C\ \mathcal{E} \cup RID_E\ e$
$LID_E\ (\mathcal{E}[e]) = LID_C\ \mathcal{E} \cup LID_E\ e$
by (*induct rule*: *plug.induct*) *auto*

lemma *ID-restricted-store-subset-store*:
$RID_S\ (\sigma(l := None)) \subseteq RID_S\ \sigma$
$LID_S\ (\sigma(l := None)) \subseteq LID_S\ \sigma$
proof −
show $RID_S\ (\sigma(l := None)) \subseteq RID_S\ \sigma$
proof (*rule subsetI*)
fix r
assume $r \in RID_S\ (\sigma(l := None))$
then obtain $l'\ v$ **where** $(\sigma(l := None))\ l' = Some\ v$ **and** *r-v*: $r \in RID_V\ v$ **by** *blast*
have $l' \neq l$ **using** ‹$(\sigma(l := None))\ l' = Some\ v$› **by** *auto*
hence $\sigma\ l' = Some\ v$ **using** ‹$(\sigma(l := None))\ l' = Some\ v$› **by** *auto*
thus $r \in RID_S\ \sigma$ **using** *r-v* **by** *blast*
qed
next
show $LID_S\ (\sigma(l := None)) \subseteq LID_S\ \sigma$

```
  proof (rule subsetI)
    fix l'
    assume l' ∈ LID_S (σ(l := None))
    hence l' ∈ dom (σ(l := None)) ∨ (∃ l'' v. (σ(l := None)) l'' = Some v ∧ l' ∈
LID_V v) by blast
    thus l' ∈ LID_S σ
    proof (rule disjE)
      assume l' ∈ dom (σ(l := None))
      thus l' ∈ LID_S σ by auto
    next
      assume ∃ l'' v. (σ(l := None)) l'' = Some v ∧ l' ∈ LID_V v
     then obtain l'' v where (σ(l := None)) l'' = Some v and l'-in-v: l' ∈ LID_V
v by blast
      hence σ l'' = Some v by (cases l = l'', auto)
      thus l' ∈ LID_S σ using l'-in-v by blast
    qed
  qed
qed

lemma in-restricted-store-in-unrestricted-store [intro]:
  r ∈ RID_S (σ(l := None)) ⟹ r ∈ RID_S σ
  l' ∈ LID_S (σ(l := None)) ⟹ l' ∈ LID_S σ
  by (meson contra-subsetD ID-restricted-store-subset-store)+

lemma in-restricted-store-in-updated-store [intro]:
  r ∈ RID_S (σ(l := None)) ⟹ r ∈ RID_S (σ(l ↦ v))
  l' ∈ LID_S (σ(l := None)) ⟹ l' ∈ LID_S (σ(l ↦ v))
proof −
  assume r ∈ RID_S (σ(l := None))
  hence r ∈ RID_S (σ |` (− {l})) by (simp add: restrict-complement-singleton-eq)
  hence r ∈ RID_S (σ(l ↦ v) |` (− {l})) by simp
 hence r ∈ RID_S (σ(l := Some v, l := None)) by (simp add: restrict-complement-singleton-eq)
  thus r ∈ RID_S (σ(l ↦ v)) by blast
next
  assume l' ∈ LID_S (σ(l := None))
  hence l' ∈ LID_S (σ |` (− {l})) by (simp add: restrict-complement-singleton-eq)
  hence l' ∈ LID_S (σ(l ↦ v) |` (− {l})) by simp
 hence l' ∈ LID_S (σ(l := Some v, l := None)) by (simp add: restrict-complement-singleton-eq)
  thus l' ∈ LID_S (σ(l ↦ v)) by blast
qed

lemma ID-distr-store [simp]:
  RID_S (τ(l ↦ v)) = RID_S (τ(l := None)) ∪ RID_V v
  LID_S (τ(l ↦ v)) = insert l (LID_S (τ(l := None)) ∪ LID_V v)
proof −
  show RID_S (τ(l ↦ v)) = RID_S (τ(l := None)) ∪ RID_V v
  proof (rule set-eqI, rule iffI)
    fix r
    assume r ∈ RID_S (τ(l ↦ v))
```

```
    then obtain l' v' where (τ(l ↦ v)) l' = Some v' r ∈ RID_V v' by blast
    thus r ∈ RID_S (τ(l := None)) ∪ RID_V v by (cases l' = l) auto
  qed auto
next
  show LID_S (τ(l ↦ v)) = insert l (LID_S (τ(l := None)) ∪ LID_V v)
  proof (rule set-eqI, rule iffI)
    fix l'
    assume l' ∈ LID_S (τ(l ↦ v))
    hence l' ∈ dom (τ(l ↦ v)) ∨ (∃ l'' v'. (τ(l ↦ v)) l'' = Some v' ∧ l' ∈ LID_V
v') by blast
    thus l' ∈ insert l (LID_S (τ(l := None)) ∪ LID_V v)
    proof (rule disjE)
      assume l' ∈ dom (τ(l ↦ v))
      thus l' ∈ insert l (LID_S (τ(l := None)) ∪ LID_V v) by (cases l' = l) auto
    next
      assume ∃ l'' v'. (τ(l ↦ v)) l'' = Some v' ∧ l' ∈ LID_V v'
      then obtain l'' v' where (τ(l ↦ v)) l'' = Some v' l' ∈ LID_V v' by blast
      thus l' ∈ insert l (LID_S (τ(l := None)) ∪ LID_V v) by (cases l = l'') auto
    qed
  qed auto
qed

lemma ID-distr-local [simp]:
  LID_L (σ,τ,e) = LID_S σ ∪ LID_S τ ∪ LID_E e
  RID_L (σ,τ,e) = RID_S σ ∪ RID_S τ ∪ RID_E e
  by (simp add: LID_L-def RID_L-def)+

lemma ID-restricted-global-subset-unrestricted:
  LID_G (s(r := None)) ⊆ LID_G s
  RID_G (s(r := None)) ⊆ RID_G s
proof −
  show LID_G (s(r := None)) ⊆ LID_G s
  proof (rule subsetI)
    fix l
    assume l ∈ LID_G (s(r := None))
    then obtain r'' ls where (s(r := None)) r'' = Some ls and l-in-ls: l ∈ LID_L
ls by blast
    have r'' ≠ r  using ‹(s(r := None)) r'' = Some ls› by auto
    hence s r'' = Some ls using ‹(s(r := None)) r'' = Some ls› by auto
    thus l ∈ LID_G s using l-in-ls by blast
  qed
next
  show RID_G (s(r := None)) ⊆ RID_G s
  proof (rule subsetI)
    fix r'
    assume r' ∈ RID_G (s(r := None))
    hence r' ∈ dom (s(r := None)) ∨ (∃ r'' ls. (s(r := None)) r'' = Some ls ∧ r'
∈ RID_L ls) by blast
    thus r' ∈ RID_G s
```

proof (*rule disjE*)
assume $\exists\, r''\ ls.\ (s(r := None))\ r'' = Some\ ls \wedge r' \in RID_L\ ls$
then obtain $r''\ ls$ **where** $(s(r := None))\ r'' = Some\ ls$ **and** r'*-in-ls*: $r' \in RID_L\ ls$ **by** *blast*
have *neq*: $r'' \neq r$ **using** ‹$(s(r := None))\ r'' = Some\ ls$› **by** *auto*
hence $s\ r'' = Some\ ls$ **using** ‹$(s(r := None))\ r'' = Some\ ls$› **by** *auto*
thus $r' \in RID_G\ s$ **using** r'*-in-ls* **by** *blast*
qed *auto*
qed
qed

lemma *in-restricted-global-in-unrestricted-global* [*intro*]:
$r' \in RID_G\ (s(r := None)) \Longrightarrow r' \in RID_G\ s$
$l \in LID_G\ (s(r := None)) \Longrightarrow l \in LID_G\ s$
by (*simp add*: *ID-restricted-global-subset-unrestricted rev-subsetD*)+

lemma *in-restricted-global-in-updated-global* [*intro*]:
$r' \in RID_G\ (s(r := None)) \Longrightarrow r' \in RID_G\ (s(r \mapsto ls))$
$l \in LID_G\ (s(r := None)) \Longrightarrow l \in LID_G\ (s(r \mapsto ls))$
proof −
assume $r' \in RID_G\ (s(r := None))$
hence $r' \in RID_G\ (s\ |`\ (-\ \{r\}))$ **by** (*simp add*: *restrict-complement-singleton-eq*)
hence $r' \in RID_G\ (s(r \mapsto ls)\ |`\ (-\ \{r\}))$ **by** *simp*
hence $r' \in RID_G\ (s(r := Some\ ls, r := None))$ **by** (*simp add*: *restrict-complement-singleton-eq*)
thus $r' \in RID_G\ (s(r \mapsto ls))$ **by** *blast*
next
assume $l \in LID_G\ (s(r := None))$
hence $l \in LID_G\ (s\ |`\ (-\ \{r\}))$ **by** (*simp add*: *restrict-complement-singleton-eq*)
hence $l \in LID_G\ (s(r \mapsto ls)\ |`\ (-\ \{r\}))$ **by** *simp*
hence $l \in LID_G\ (s(r := Some\ ls, r := None))$ **by** (*simp add*: *restrict-complement-singleton-eq*)
thus $l \in LID_G\ (s(r \mapsto ls))$ **by** *blast*
qed

lemma *ID-distr-global* [*simp*]:
$RID_G\ (s(r \mapsto ls)) = insert\ r\ (RID_G\ (s(r := None)) \cup RID_L\ ls)$
$LID_G\ (s(r \mapsto ls)) = LID_G\ (s(r := None)) \cup LID_L\ ls$
proof −
show $LID_G\ (s(r \mapsto ls)) = LID_G\ (s(r := None)) \cup LID_L\ ls$
proof (*rule set-eqI*)
fix l
show $(l \in LID_G\ (s(r \mapsto ls))) = (l \in LID_G\ (s(r := None)) \cup LID_L\ ls)$
proof (*rule iffI*)
assume $l \in LID_G\ (s(r \mapsto ls))$
from *this* **obtain** $r'\ ls'$ **where** $(s(r \mapsto ls))\ r' = Some\ ls'\ l \in LID_L\ ls'$ **by** *auto*
thus $l \in LID_G\ (s(r := None)) \cup LID_L\ ls$ **by** (*cases* $r = r'$) *auto*
qed *auto*
qed
show $RID_G\ (s(r \mapsto ls)) = insert\ r\ (RID_G\ (s(r := None)) \cup RID_L\ ls)$

proof (*rule set-eqI*)
fix r'
show $(r' \in RID_G\ (s(r \mapsto ls))) = (r' \in insert\ r\ (RID_G\ (s(r := None)) \cup RID_L\ ls))$
proof (*rule iffI*, *clarsimp*)
assume r'-g: $r' \in RID_G\ (s(r \mapsto ls))$ **and** *neq*: $r' \neq r$ **and** r'-l: $r' \notin RID_L\ ls$
show $r' \in RID_G\ (s(r := None))$
proof (*rule* $RID_G E[OF\ r'\text{-}g]$)
assume $r' \in dom\ (s(r \mapsto ls))$
thus *?thesis* **by** (*simp add*: RID_G-*def neq*)
next
fix $ls'\ r''$
assume r'-*in-range*:$(s(r \mapsto ls))\ r'' = Some\ ls'\ r' \in RID_L\ ls'$
thus *?thesis* **by** (*cases* $r'' = r$) (*use neq* r'-l **in** *auto*)
qed
qed *auto*
qed
qed

lemma *restrictions-inwards* [*simp*]:
$x \neq x' \Longrightarrow f(x := Some\ y,\ x' := None) = (f(x' := None,\ x := Some\ y))$
by (*rule fun-upd-twist*)

2.2.3 Miscellaneous laws

lemma *ID-combination-subset-union* [*intro*]:
$RID_S\ (\sigma;;\tau) \subseteq RID_S\ \sigma \cup RID_S\ \tau$
$LID_S\ (\sigma;;\tau) \subseteq LID_S\ \sigma \cup LID_S\ \tau$
proof −
show $RID_S\ (\sigma;;\tau) \subseteq RID_S\ \sigma \cup RID_S\ \tau$
proof (*rule subsetI*)
fix r
assume $r \in RID_S\ (\sigma;;\tau)$
from *this* **obtain** $l\ v$ **where** $(\sigma;;\tau)\ l = Some\ v$ **and** $r \in RID_V\ v$ **by** *blast*
thus $r \in RID_S\ \sigma \cup RID_S\ \tau$ **by** (*cases* $l \in dom\ \tau$) *auto*
qed
show $LID_S\ (\sigma;;\tau) \subseteq LID_S\ \sigma \cup LID_S\ \tau$
proof (*rule subsetI*)
fix l
assume $l \in LID_S\ (\sigma;;\tau)$
hence $l \in dom\ (\sigma;;\tau) \vee (\exists l'\ v.\ (\sigma;;\tau)\ l' = Some\ v \wedge l \in LID_V\ v)$ **by** *blast*
thus $l \in LID_S\ \sigma \cup LID_S\ \tau$
proof (*rule disjE*)
assume $l \in dom\ (\sigma;;\tau)$
thus $l \in LID_S\ \sigma \cup LID_S\ \tau$ **by** (*cases* $l \in dom\ \tau$) *auto*
next
assume $\exists l'\ v.\ (\sigma;;\tau)\ l' = Some\ v \wedge l \in LID_V\ v$
from *this* **obtain** $l'\ v$ **where** $(\sigma;;\tau)\ l' = Some\ v\ l \in LID_V\ v$ **by** *blast*
thus $l \in LID_S\ \sigma \cup LID_S\ \tau$ **by** (*cases* $l' \in dom\ \tau$) *auto*

```
    qed
  qed
qed

lemma in-combination-in-component [intro]:
  r ∈ RID_S (σ;;τ) ⟹ r ∉ RID_S σ ⟹ r ∈ RID_S τ
  r ∈ RID_S (σ;;τ) ⟹ r ∉ RID_S τ ⟹ r ∈ RID_S σ
  l ∈ LID_S (σ;;τ) ⟹ l ∉ LID_S σ ⟹ l ∈ LID_S τ
  l ∈ LID_S (σ;;τ) ⟹ l ∉ LID_S τ ⟹ l ∈ LID_S σ
  by (meson Un-iff ID-combination-subset-union subsetCE)+

lemma in-mapped-in-component-of-combination [intro]:
  assumes
    maps-to-v: (σ;;τ) l = Some v and
    l'-in-v: l' ∈ LID_V v
  shows
    l' ∉ LID_S τ ⟹ l' ∈ LID_S σ
    l' ∉ LID_S σ ⟹ l' ∈ LID_S τ
  by (metis LID_S I(2) combine.simps l'-in-v maps-to-v)+

lemma elim-trivial-restriction [simp]: l ∉ LID_S τ ⟹ (τ(l := None)) = τ
  by (simp add: LID_S-def domIff fun-upd-idem)

lemma ID-distr-global-conditional:
  s r = Some ls ⟹ RID_G s = insert r (RID_G (s(r := None)) ∪ RID_L ls)
  s r = Some ls ⟹ LID_G s = LID_G (s(r := None)) ∪ LID_L ls
proof −
  assume s r = Some ls
  hence s-as-upd: s = (s(r ↦ ls)) by (simp add: fun-upd-idem)
  show RID_G s = insert r (RID_G (s(r := None)) ∪ RID_L ls) by (subst s-as-upd,
simp)
  show LID_G s = LID_G (s(r := None)) ∪ LID_L ls by (subst s-as-upd, simp)
qed

end
```

3 Renaming

Similar to the bound variables of lambda calculus, location and revision identifiers are meaningless names. This theory contains all of the definitions and results required for renaming data structures and proving renaming-equivalence.

```
theory Renaming
  imports Occurrences
begin
```

3.1 Definitions

abbreviation *rename-val* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ val \Rightarrow ('r,'l,'v)\ val$ $(\langle\mathcal{R}_V\rangle)$ **where**
$\mathcal{R}_V\ \alpha\ \beta\ v \equiv$ *map-val* $\alpha\ \beta$ *id* v

abbreviation *rename-expr* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ expr \Rightarrow ('r,'l,'v)$ *expr* $(\langle\mathcal{R}_E\rangle)$ **where**
$\mathcal{R}_E\ \alpha\ \beta\ e \equiv$ *map-expr* $\alpha\ \beta$ *id* e

abbreviation *rename-cntxt* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ cntxt \Rightarrow ('r,'l,'v)$ *cntxt* $(\langle\mathcal{R}_C\rangle)$ **where**
$\mathcal{R}_C\ \alpha\ \beta\ \mathcal{E} \equiv$ *map-cntxt* $\alpha\ \beta$ *id* $\mathcal{E}$

definition *is-store-renaming* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ store \Rightarrow ('r,'l,'v)$ *store* $\Rightarrow$ *bool* **where**
is-store-renaming $\alpha\ \beta\ \sigma\ \sigma' \equiv \forall\, l.$ *case* $\sigma\ l$ *of None* $\Rightarrow \sigma'\ (\beta\ l) =$ *None* | *Some* v $\Rightarrow \sigma'\ (\beta\ l) =$ *Some* $(\mathcal{R}_V\ \alpha\ \beta\ v)$

notation *Option.bind* (**infixl** $\langle\gg\!=\rangle$ *80*)

fun $\mathcal{R}_S$ :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ store \Rightarrow ('r,'l,'v)\ store$ **where**
$\mathcal{R}_S\ \alpha\ \beta\ \sigma\ l = \sigma\ (inv\ \beta\ l) \gg\!= (\lambda v.$ *Some* $(\mathcal{R}_V\ \alpha\ \beta\ v))$

lemma $\mathcal{R}_S$*-implements-renaming*: *bij* $\beta \Longrightarrow$ *is-store-renaming* $\alpha\ \beta\ \sigma\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
proof −
assume *bij* β
hence *inj* β **using** *bij-def* **by** *auto*
thus *?thesis* **by** (*auto simp add*: *is-store-renaming-def option.case-eq-if*)
qed

fun $\mathcal{R}_L$ :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ local\text{-}state \Rightarrow ('r,'l,'v)\ local\text{-}state$ **where**
$\mathcal{R}_L\ \alpha\ \beta\ (\sigma,\tau,e) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma, \mathcal{R}_S\ \alpha\ \beta\ \tau, \mathcal{R}_E\ \alpha\ \beta\ e)$

definition *is-global-renaming* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ global\text{-}state \Rightarrow ('r,'l,'v)\ global\text{-}state \Rightarrow$ *bool* **where**
is-global-renaming $\alpha\ \beta\ s\ s' \equiv \forall\, r.$ *case* $s\ r$ *of None* $\Rightarrow s'\ (\alpha\ r) =$ *None* | *Some* ls $\Rightarrow s'\ (\alpha\ r) =$ *Some* $(\mathcal{R}_L\ \alpha\ \beta\ ls)$

fun $\mathcal{R}_G$ :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ global\text{-}state \Rightarrow ('r,'l,'v)\ global\text{-}state$ **where**
$\mathcal{R}_G\ \alpha\ \beta\ s\ r = s\ (inv\ \alpha\ r) \gg\!= (\lambda ls.$ *Some* $(\mathcal{R}_L\ \alpha\ \beta\ ls))$

lemma $\mathcal{R}_G$*-implements-renaming*: *bij* $\alpha \Longrightarrow$ *is-global-renaming* $\alpha\ \beta\ s\ (\mathcal{R}_G\ \alpha\ \beta\ s)$
proof −
assume *bij* α
hence *inj* α **using** *bij-def* **by** *auto*
thus *?thesis* **by** (*auto simp add*: *is-global-renaming-def option.case-eq-if*)
qed

3.2 Introduction rules

lemma $\mathcal{R}_S I$ [*intro*]:
assumes
bij-β: *bij* β **and**
none-case: $\bigwedge l.\ \sigma\ l = None \Longrightarrow \sigma'\ (\beta\ l) = None$ **and**
some-case: $\bigwedge l\ v.\ \sigma\ l = Some\ v \Longrightarrow \sigma'\ (\beta\ l) = Some\ (\mathcal{R}_V\ \alpha\ \beta\ v)$
shows
$\mathcal{R}_S\ \alpha\ \beta\ \sigma = \sigma'$
proof (*rule ext*, *subst* $\mathcal{R}_S$*.simps*)
fix *l*
show $\sigma\ (inv\ \beta\ l) \gg\!= (\lambda v.\ Some\ (\mathcal{R}_V\ \alpha\ \beta\ v)) = \sigma'\ l$ (**is** *?lhs* $= \sigma'\ l$)
proof (*cases* $\sigma\ (inv\ \beta\ l) = None$)
case *True*
have *lhs-none*: *?lhs* $= None$ **by** (*simp add*: *True*)
have $\sigma'\ (\beta\ (inv\ \beta\ l)) = None$ **by** (*simp add*: *none-case True*)
hence *rhs-none*: $\sigma'\ l = None$ **by** (*simp add*: *bij-β bijection.intro bijection.inv-right*)
show *?thesis* **by** (*simp add*: *lhs-none rhs-none*)
next
case *False*
from *this* **obtain** *v* **where** *is-some*: $\sigma\ (inv\ \beta\ l) = Some\ v$ **by** *blast*
hence *lhs-some*: *?lhs* $= Some\ (\mathcal{R}_V\ \alpha\ \beta\ v)$ **by** *auto*
have $\sigma'\ (\beta\ (inv\ \beta\ l)) = Some\ (\mathcal{R}_V\ \alpha\ \beta\ v)$ **by** (*simp add*: *is-some some-case*)
hence *rhs-some*: $\sigma'\ l = Some\ (\mathcal{R}_V\ \alpha\ \beta\ v)$ **by** (*simp add*: *bij-β bijection.intro bijection.inv-right*)
then show *?thesis* **by** (*simp add*: *lhs-some*)
qed
qed

lemma $\mathcal{R}_G I$ [*intro*]:
assumes
bij-α: *bij* α **and**
none-case: $\bigwedge r.\ s\ r = None \Longrightarrow s'\ (\alpha\ r) = None$ **and**
some-case: $\bigwedge r\ \sigma\ \tau\ e.\ s\ r = Some\ (\sigma,\tau,e) \Longrightarrow s'\ (\alpha\ r) = Some\ (\mathcal{R}_L\ \alpha\ \beta\ (\sigma,\tau,e))$
shows
$\mathcal{R}_G\ \alpha\ \beta\ s = s'$
proof (*rule ext*, *subst* $\mathcal{R}_G$*.simps*)
fix *r*
show $s\ (inv\ \alpha\ r) \gg\!= (\lambda ls.\ Some\ (\mathcal{R}_L\ \alpha\ \beta\ ls)) = s'\ r$ (**is** *?lhs* $= s'\ r$)
proof (*cases* $s\ (inv\ \alpha\ r) = None$)
case *True*
have *lhs-none*: *?lhs* $= None$ **by** (*simp add*: *True*)
have $s'\ (\alpha\ (inv\ \alpha\ r)) = None$ **by** (*simp add*: *none-case True*)
hence *rhs-none*: $s'\ r = None$ **by** (*simp add*: *bij-α bijection.intro bijection.inv-right*)
show *?thesis* **by** (*simp add*: *lhs-none rhs-none*)
next
case *False*
from *this* **obtain** *ls* **where** $s\ (inv\ \alpha\ r) = Some\ ls$ **by** *blast*
from *this* **obtain** $\sigma\ \tau\ e$ **where** *is-some*: $s\ (inv\ \alpha\ r) = Some\ (\sigma,\ \tau,\ e)$ **by** (*cases ls*) *blast*

hence *lhs-some*: *?lhs* = *Some* ($\mathcal{R}_L$ α β (σ, τ, e)) **by** *auto*
have s' (α (*inv* α r)) = *Some* ($\mathcal{R}_L$ α β (σ, τ, e)) **by** (*simp add*: *is-some some-case*)
hence *rhs-some*: s' r = *Some* ($\mathcal{R}_L$ α β (σ, τ, e)) **by** (*simp add*: *bij-*α *bijection.intro bijection.inv-right*)
then show *?thesis* **by** (*simp add*: *lhs-some*)
qed
qed

3.3 Renaming-equivalence

3.3.1 Identity

declare *val.map-id* [*simp*]
declare *expr.map-id* [*simp*]
declare *cntxt.map-id* [*simp*]
lemma $\mathcal{R}_S$*-id* [*simp*]: $\mathcal{R}_S$ *id id* σ = σ **by** *auto*
lemma $\mathcal{R}_L$*-id* [*simp*]: $\mathcal{R}_L$ *id id ls* = *ls* **by** (*cases ls*) *simp*
lemma $\mathcal{R}_G$*-id* [*simp*]: $\mathcal{R}_G$ *id id s* = *s* **by** *auto*

3.3.2 Composition

declare *val.map-comp* [*simp*]
declare *expr.map-comp* [*simp*]
declare *cntxt.map-comp* [*simp*]
lemma $\mathcal{R}_S$*-comp* [*simp*]: ⟦ *bij* β; *bij* β' ⟧ ⟹ $\mathcal{R}_S$ α' β' ($\mathcal{R}_S$ α β s) = $\mathcal{R}_S$ ($\alpha' \circ \alpha$) ($\beta' \circ \beta$) s
by (*auto simp add*: *o-inv-distrib*)
lemma $\mathcal{R}_L$*-comp* [*simp*]: ⟦ *bij* β; *bij* β' ⟧ ⟹ $\mathcal{R}_L$ α' β' ($\mathcal{R}_L$ α β *ls*) = $\mathcal{R}_L$ ($\alpha' \circ \alpha$) ($\beta' \circ \beta$) *ls*
by (*cases ls*) *simp*
lemma $\mathcal{R}_G$*-comp* [*simp*]: ⟦ *bij* α; *bij* α'; *bij* β; *bij* β' ⟧ ⟹ $\mathcal{R}_G$ α' β' ($\mathcal{R}_G$ α β s) = $\mathcal{R}_G$ ($\alpha' \circ \alpha$) ($\beta' \circ \beta$) s
by (*rule ext*) (*auto simp add*: *o-inv-distrib*)

3.3.3 Inverse

lemma $\mathcal{R}_V$*-inv* [*simp*]: ⟦ *bij* α; *bij* β ⟧ ⟹ ($\mathcal{R}_V$ (*inv* α) (*inv* β) v' = v) = ($\mathcal{R}_V$ α β v = v')
by (*auto simp add*: *bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)
lemma $\mathcal{R}_E$*-inv* [*simp*]: ⟦ *bij* α; *bij* β ⟧ ⟹ ($\mathcal{R}_E$ (*inv* α) (*inv* β) e' = e) = ($\mathcal{R}_E$ α β e = e')
by (*auto simp add*: *bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)
lemma $\mathcal{R}_C$*-inv* [*simp*]: ⟦ *bij* α; *bij* β ⟧ ⟹ ($\mathcal{R}_C$ (*inv* α) (*inv* β) $\mathcal{E}'$ = $\mathcal{E}$) = ($\mathcal{R}_C$ α β $\mathcal{E}$ = $\mathcal{E}'$)
by (*auto simp add*: *bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)
lemma $\mathcal{R}_S$*-inv* [*simp*]: ⟦ *bij* α; *bij* β ⟧ ⟹ ($\mathcal{R}_S$ (*inv* α) (*inv* β) σ' = σ) = ($\mathcal{R}_S$ α β σ = σ')
by (*auto simp add*: *bij-imp-bij-inv bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)

lemma $\mathcal{R}_L$*-inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_L\ (inv\ \alpha)\ (inv\ \beta)\ ls' = ls) = (\mathcal{R}_L\ \alpha\ \beta\ ls = ls')$
by (*auto simp add*: *bij-imp-bij-inv bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)
lemma $\mathcal{R}_G$*-inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_G\ (inv\ \alpha)\ (inv\ \beta)\ s' = s) = (\mathcal{R}_G\ \alpha\ \beta\ s = s')$
by (*auto simp add*: *bij-imp-bij-inv bijection.intro bijection.inv-comp-right bijection.inv-comp-left*)

3.3.4 Equivalence

definition *eq-states* :: ('*r*,'*l*,'*v*) *global-state* ⇒ ('*r*,'*l*,'*v*) *global-state* ⇒ *bool* (‹- ≈ -› [*100*, *100*]) **where**
$s \approx s' \equiv \exists \alpha\ \beta.\ bij\ \alpha \wedge bij\ \beta \wedge \mathcal{R}_G\ \alpha\ \beta\ s = s'$

lemma *eq-statesI* [*intro*]:
$\mathcal{R}_G\ \alpha\ \beta\ s = s' \Longrightarrow bij\ \alpha \Longrightarrow bij\ \beta \Longrightarrow s \approx s'$
using *eq-states-def* **by** *auto*

lemma *eq-statesE* [*elim*]:
$s \approx s' \Longrightarrow (\bigwedge \alpha\ \beta.\ \mathcal{R}_G\ \alpha\ \beta\ s = s' \Longrightarrow bij\ \alpha \Longrightarrow bij\ \beta \Longrightarrow P) \Longrightarrow P$
using *eq-states-def* **by** *blast*

lemma $\alpha\beta$*-refl*: $s \approx s$ **by** (*rule eq-statesI*[*of id id s*]) *auto*

lemma $\alpha\beta$*-trans*: $s \approx s' \Longrightarrow s' \approx s'' \Longrightarrow s \approx s''$
proof −
assume $s \approx s'$
from *this* **obtain** $\alpha\ \beta$ **where** *s-s'*: $bij\ \alpha\ bij\ \beta\ \mathcal{R}_G\ \alpha\ \beta\ s = s'$ **by** *blast*
assume $s' \approx s''$
from *this* **obtain** $\alpha'\ \beta'$ **where** *s'-s''*: $bij\ \alpha'\ bij\ \beta'\ \mathcal{R}_G\ \alpha'\ \beta'\ s' = s''$ **by** *blast*
show $s \approx s''$ **by** (*rule eq-statesI*[*of* $\alpha' \circ \alpha\ \beta' \circ \beta$]) (*use s-s' s'-s'* **in** ‹*auto simp add*: *bij-comp*›)
qed

lemma $\alpha\beta$*-sym*: $s \approx s' \Longrightarrow s' \approx s$
proof −
assume $s \approx s'$
from *this* **obtain** $\alpha\ \beta$ **where** *s-s'*: $bij\ \alpha\ bij\ \beta\ \mathcal{R}_G\ \alpha\ \beta\ s = s'$ **by** *blast*
show $s' \approx s$ **by** (*rule eq-statesI*[*of inv* α *inv* β]) (*use s-s'* **in** ‹*auto simp add*: *bij-imp-bij-inv*›)
qed

3.4 Distributive laws

3.4.1 Expression

lemma *renaming-distr-completion* [*simp*]:
$\mathcal{R}_E\ \alpha\ \beta\ (\mathcal{E}[e]) = ((\mathcal{R}_C\ \alpha\ \beta\ \mathcal{E})[\mathcal{R}_E\ \alpha\ \beta\ e])$
by (*induct* $\mathcal{E}$) *simp+*

3.4.2 Store

lemma *renaming-distr-combination* [*simp*]:
$\mathcal{R}_S\ \alpha\ \beta\ (\sigma;;\tau) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma;;\mathcal{R}_S\ \alpha\ \beta\ \tau)$
by (*rule ext*) *auto*

lemma *renaming-distr-store* [*simp*]:
$bij\ \beta \Longrightarrow \mathcal{R}_S\ \alpha\ \beta\ (\sigma(l \mapsto v)) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma)(\beta\ l \mapsto \mathcal{R}_V\ \alpha\ \beta\ v)$
by (*auto simp add*: *bijection.intro bijection.inv-left-eq-iff*)

3.4.3 Global

lemma *renaming-distr-global* [*simp*]:
$bij\ \alpha \Longrightarrow \mathcal{R}_G\ \alpha\ \beta\ (s(r \mapsto ls)) = (\mathcal{R}_G\ \alpha\ \beta\ s)(\alpha\ r \mapsto \mathcal{R}_L\ \alpha\ \beta\ ls)$
$bij\ \alpha \Longrightarrow \mathcal{R}_G\ \alpha\ \beta\ (s(r := None)) = (\mathcal{R}_G\ \alpha\ \beta\ s)(\alpha\ r := None)$
by (*auto simp add*: *bijection.intro bijection.inv-left-eq-iff*)

3.5 Miscellaneous laws

lemma *rename-empty* [*simp*]:
$\mathcal{R}_S\ \alpha\ \beta\ \varepsilon = \varepsilon$
$\mathcal{R}_G\ \alpha\ \beta\ \varepsilon = \varepsilon$
by *auto*

3.6 Swaps

lemma *swap-bij*:
$bij\ (id(x := x',\ x' := x))$ (**is** *bij ?f*)
proof (*rule bijI*)
show *inj ?f* **by** (*simp add*: *inj-on-def*)
show *surj ?f*
proof
show $UNIV \subseteq range\ (id(x := x',\ x' := x))$
proof (*rule subsetI*)
fix *y*
assume $y \in (UNIV :: 'a\ set)$
show $y \in range\ (id(x := x',\ x' := x))$ **by** (*cases* $y = x$; *cases* $y = x'$) *auto*
qed
qed *simp*
qed

lemma *id-trivial-update* [*simp*]: $id(x := x) = id$ **by** *auto*

lemma *eliminate-renaming-val-expr* [*simp*]:
fixes
$v :: ('r,'l,'v)\ val$ **and**
$e :: ('r,'l,'v)\ expr$
shows
$l \notin LID_V\ v \Longrightarrow \mathcal{R}_V\ \alpha\ (\beta(l := l'))\ v = \mathcal{R}_V\ \alpha\ \beta\ v$
$l \notin LID_E\ e \Longrightarrow \mathcal{R}_E\ \alpha\ (\beta(l := l'))\ e = \mathcal{R}_E\ \alpha\ \beta\ e$

$r \notin RID_V\ v \Longrightarrow \mathcal{R}_V\ (\alpha(r := r'))\ \beta\ v = \mathcal{R}_V\ \alpha\ \beta\ v$
$r \notin RID_E\ e \Longrightarrow \mathcal{R}_E\ (\alpha(r := r'))\ \beta\ e = \mathcal{R}_E\ \alpha\ \beta\ e$
proof −
have $(\forall\, \alpha\ \beta\ r\ r'.\ r \notin RID_V\ v \longrightarrow \mathcal{R}_V\ (\alpha(r := r'))\ \beta\ v = \mathcal{R}_V\ \alpha\ \beta\ v) \wedge$
$(\forall\, \alpha\ \beta\ r\ r'.\ r \notin RID_E\ e \longrightarrow \mathcal{R}_E\ (\alpha(r := r'))\ \beta\ e = \mathcal{R}_E\ \alpha\ \beta\ e)$
by (*induct rule*: *val-expr.induct*) *simp+*
thus
$r \notin RID_V\ v \Longrightarrow \mathcal{R}_V\ (\alpha(r := r'))\ \beta\ v = \mathcal{R}_V\ \alpha\ \beta\ v$
$r \notin RID_E\ e \Longrightarrow \mathcal{R}_E\ (\alpha(r := r'))\ \beta\ e = \mathcal{R}_E\ \alpha\ \beta\ e$
by *simp+*
have $(\forall\, \alpha\ \beta\ l\ l'.\ l \notin LID_V\ v \longrightarrow \mathcal{R}_V\ \alpha\ (\beta(l := l'))\ v = \mathcal{R}_V\ \alpha\ \beta\ v) \wedge$
$(\forall\, \alpha\ \beta\ l\ l'.\ l \notin LID_E\ e \longrightarrow \mathcal{R}_E\ \alpha\ (\beta(l := l'))\ e = \mathcal{R}_E\ \alpha\ \beta\ e)$
by (*induct rule*: *val-expr.induct*) *simp+*
thus
$l \notin LID_V\ v \Longrightarrow \mathcal{R}_V\ \alpha\ (\beta(l := l'))\ v = \mathcal{R}_V\ \alpha\ \beta\ v$ **and**
$l \notin LID_E\ e \Longrightarrow \mathcal{R}_E\ \alpha\ (\beta(l := l'))\ e = \mathcal{R}_E\ \alpha\ \beta\ e$
by *simp+*
qed

lemma *eliminate-renaming-cntxt* [*simp*]:
$r \notin RID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ (\alpha(r := r'))\ \beta\ \mathcal{E} = \mathcal{R}_C\ \alpha\ \beta\ \mathcal{E}$
$l \notin LID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ \alpha\ (\beta(l := l'))\ \mathcal{E} = \mathcal{R}_C\ \alpha\ \beta\ \mathcal{E}$
by (*induct* $\mathcal{E}$ *rule*: *cntxt.induct*) *auto*

lemma *eliminate-swap-val* [*simp*, *intro*]:
$r \notin RID_V\ v \Longrightarrow r' \notin RID_V\ v \Longrightarrow \mathcal{R}_V\ (id(r := r', r' := r))\ id\ v = v$
$l \notin LID_V\ v \Longrightarrow l' \notin LID_V\ v \Longrightarrow \mathcal{R}_V\ id\ (id(l := l', l' := l))\ v = v$
by *simp+*

lemma *eliminate-swap-expr* [*simp*, *intro*]:
$r \notin RID_E\ e \Longrightarrow r' \notin RID_E\ e \Longrightarrow \mathcal{R}_E\ (id(r := r', r' := r))\ id\ e = e$
$l \notin LID_E\ e \Longrightarrow l' \notin LID_E\ e \Longrightarrow \mathcal{R}_E\ id\ (id(l := l', l' := l))\ e = e$
by *simp+*

lemma *eliminate-swap-cntxt* [*simp*, *intro*]:
$r \notin RID_C\ \mathcal{E} \Longrightarrow r' \notin RID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ (id(r := r', r' := r))\ id\ \mathcal{E} = \mathcal{E}$
$l \notin LID_C\ \mathcal{E} \Longrightarrow l' \notin LID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ id\ (id(l := l', l' := l))\ \mathcal{E} = \mathcal{E}$
by *simp+*

lemma *eliminate-swap-store-rid* [*simp*, *intro*]:
$r \notin RID_S\ \sigma \Longrightarrow r' \notin RID_S\ \sigma \Longrightarrow \mathcal{R}_S\ (id(r := r', r' := r))\ id\ \sigma = \sigma$
by (*rule* $\mathcal{R}_S I$) (*auto simp add*: *swap-bij* RID_S*-def domIff ranI*)

lemma *eliminate-swap-store-lid* [*simp*, *intro*]:
$l \notin LID_S\ \sigma \Longrightarrow l' \notin LID_S\ \sigma \Longrightarrow \mathcal{R}_S\ id\ (id(l := l', l' := l))\ \sigma = \sigma$
by (*rule* $\mathcal{R}_S I$) (*auto simp add*: *swap-bij* LID_S*-def domIff ranI*)

lemma *eliminate-swap-global-rid* [*simp*, *intro*]:
$r \notin RID_G\ s \Longrightarrow r' \notin RID_G\ s \Longrightarrow \mathcal{R}_G\ (id(r := r', r' := r))\ id\ s = s$

by (*rule* $\mathcal{R}_G I[OF\ swap\text{-}bij]$, ((*rule sym*, *auto*)[*1*])+)

lemma *eliminate-swap-global-lid* [*simp*, *intro*]:
$l \notin LID_G\ s \Longrightarrow l' \notin LID_G\ s \Longrightarrow \mathcal{R}_G\ id\ (id(l := l', l' := l))\ s = s$
by (*rule* $\mathcal{R}_G I$) (*auto simp add*: *ID-distr-global-conditional*)

end

4 Substitution

This theory introduces the substitution operation using a locale, and provides two models.

theory *Substitution*
imports *Renaming*
begin

4.1 Definition

locale *substitution* =
fixes *subst* :: $('r,'l,'v)\ expr \Rightarrow 'v \Rightarrow ('r,'l,'v)\ expr \Rightarrow ('r,'l,'v)\ expr$
assumes
renaming-distr-subst: $\mathcal{R}_E\ \alpha\ \beta\ (subst\ e\ x\ e') = subst\ (\mathcal{R}_E\ \alpha\ \beta\ e)\ x\ (\mathcal{R}_E\ \alpha\ \beta\ e')$
and
subst-introduces-no-rids: $RID_E\ (subst\ e\ x\ e') \subseteq RID_E\ e \cup RID_E\ e'$ **and**
subst-introduces-no-lids: $LID_E\ (subst\ e\ x\ e') \subseteq LID_E\ e \cup LID_E\ e'$
begin

lemma *rid-substE* [*dest*]: $r \in RID_E\ (subst\ (VE\ v)\ x\ e) \Longrightarrow r \notin RID_E\ e \Longrightarrow r \in RID_V\ v$
using *subst-introduces-no-rids* **by** *fastforce*

lemma *lid-substE* [*dest*]: $l \in LID_E\ (subst\ (VE\ v)\ x\ e) \Longrightarrow l \notin LID_E\ e \Longrightarrow l \in LID_V\ v$
using *subst-introduces-no-lids* **by** *fastforce*

end

4.2 Trivial model

fun *constant-function* :: $('r,'l,'v)\ expr \Rightarrow 'v \Rightarrow ('r,'l,'v)\ expr \Rightarrow ('r,'l,'v)\ expr$
where
$constant\text{-}function\ e\ x\ e' = VE\ (CV\ Unit)$

lemma *constant-function-models-substitution*:
substitution constant-function **by** (*auto simp add*: *substitution-def*)

4.3 Example model

4.3.1 Preliminaries

notation *set3-val* (‹$\mathcal{V}_V$›)
notation *set3-expr* (‹$\mathcal{V}_E$›)

abbreviation *rename-vars-val* :: $('v \Rightarrow 'v) \Rightarrow ('r,'l,'v)\ val \Rightarrow ('r,'l,'v)\ val$ (‹$\mathcal{RV}_V$›)
where
$\mathcal{RV}_V\ \zeta \equiv$ *map-val id id* ζ

abbreviation *rename-vars-expr* :: $('v \Rightarrow 'v) \Rightarrow ('r,'l,'v)\ expr \Rightarrow ('r,'l,'v)\ expr$ (‹$\mathcal{RV}_E$›) **where**
$\mathcal{RV}_E\ \zeta \equiv$ *map-expr id id* ζ

lemma *var-renaming-preserves-size*:
fixes
$v :: ('r,'l,'v)\ val$ **and**
$e :: ('r,'l,'v)\ expr$ **and**
$\alpha :: 'r \Rightarrow 'r'$ **and**
$\beta :: 'l \Rightarrow 'l'$ **and**
$\zeta :: 'v \Rightarrow 'v'$
shows
size (*map-val* $\alpha\ \beta\ \zeta\ v$) = *size* v
size (*map-expr* $\alpha\ \beta\ \zeta\ e$) = *size* e
proof −
have $(\forall(\alpha :: 'r \Rightarrow 'r')\ (\beta :: 'l \Rightarrow 'l')\ (\zeta :: 'v \Rightarrow 'v').$ *size* (*map-val* $\alpha\ \beta\ \zeta\ v$) = *size* $v) \wedge$
$(\forall(\alpha :: 'r \Rightarrow 'r')\ (\beta :: 'l \Rightarrow 'l')\ (\zeta :: 'v \Rightarrow 'v').$ *size* (*map-expr* $\alpha\ \beta\ \zeta\ e$) = *size* e)
by (*induct rule*: *val-expr.induct*) *auto*
thus
size (*map-val* $\alpha\ \beta\ \zeta\ v$) = *size* v
size (*map-expr* $\alpha\ \beta\ \zeta\ e$) = *size* e
by *auto*
qed

4.3.2 Definition

function
$nat\text{-}subst_V :: ('r,'l,nat)\ expr \Rightarrow nat \Rightarrow ('r,'l,nat)\ val \Rightarrow ('r,'l,nat)\ expr$ **and**
$nat\text{-}subst_E :: ('r,'l,nat)\ expr \Rightarrow nat \Rightarrow ('r,'l,nat)\ expr \Rightarrow ('r,'l,nat)\ expr$
where
$nat\text{-}subst_V\ e\ x\ (CV\ const) = VE\ (CV\ const)$
$\mid nat\text{-}subst_V\ e\ x\ (Var\ x') = (if\ x = x'\ then\ e\ else\ VE\ (Var\ x'))$
$\mid nat\text{-}subst_V\ e\ x\ (Loc\ l) = VE\ (Loc\ l)$
$\mid nat\text{-}subst_V\ e\ x\ (Rid\ r) = VE\ (Rid\ r)$
$\mid nat\text{-}subst_V\ e\ x\ (Lambda\ y\ e') = VE\ ($
if $x = y$ *then*
$Lambda\ y\ e'$

else
let z = *Suc* (*Max* ($\mathcal{V}_E$ e' ∪ $\mathcal{V}_E$ e)) *in*
Lambda z (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ (*id*(y := z)) e')))
| *nat-subst*$_E$ e x (*VE* v') = *nat-subst*$_V$ e x v'
| *nat-subst*$_E$ e x (*Apply* l r) = *Apply* (*nat-subst*$_E$ e x l) (*nat-subst*$_E$ e x r)
| *nat-subst*$_E$ e x (*Ite e1 e2 e3*) = *Ite* (*nat-subst*$_E$ e x *e1*) (*nat-subst*$_E$ e x *e2*) (*nat-subst*$_E$ e x *e3*)
| *nat-subst*$_E$ e x (*Ref* e') = *Ref* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Read* e') = *Read* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Assign* l r) = *Assign* (*nat-subst*$_E$ e x l) (*nat-subst*$_E$ e x r)
| *nat-subst*$_E$ e x (*Rfork* e') = *Rfork* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Rjoin* e') = *Rjoin* (*nat-subst*$_E$ e x e')
by *pat-completeness auto*
termination
by (*relation measure* (λ*x*. *case x of Inl* (*e*,*x*,*v*) ⇒ *size v* | *Inr* (*e*,*x*,*e'*) ⇒ *size e'*))
(*auto simp add*: *var-renaming-preserves-size*(*2*))

4.3.3 Proof obligations

lemma *nat-subst*$_E$*-distr*:
fixes e :: ('*r*,'*l*,*nat*) *expr*
shows $\mathcal{R}_E$ α β (*nat-subst*$_E$ e x e') = *nat-subst*$_E$ ($\mathcal{R}_E$ α β e) x ($\mathcal{R}_E$ α β e')
proof −
fix v' :: ('*r*,'*l*,*nat*) *val*
have
(∀ α β x e ζ. $\mathcal{R}_E$ α β (*nat-subst*$_V$ e x ($\mathcal{RV}_V$ ζ v')) = *nat-subst*$_V$ ($\mathcal{R}_E$ α β e) x ($\mathcal{R}_V$ α β ($\mathcal{RV}_V$ ζ v'))) ∧
(∀ α β x e ζ. $\mathcal{R}_E$ α β (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ ζ e')) = *nat-subst*$_E$ ($\mathcal{R}_E$ α β e) x ($\mathcal{R}_E$ α β ($\mathcal{RV}_E$ ζ e')))
by (*induct rule*: *val-expr.induct*) (*auto simp add*: *expr.set-map*(*3*) *fun.map-ident*)
hence $\mathcal{R}_E$ α β (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ *id* e')) = *nat-subst*$_E$ ($\mathcal{R}_E$ α β e) x ($\mathcal{R}_E$ α β ($\mathcal{RV}_E$ *id* e')) **by** *blast*
thus *?thesis* **by** *simp*
qed

lemma *nat-subst*$_E$*-introduces-no-rids*:
fixes e' :: ('*r*,'*l*,*nat*) *expr*
shows *RID*$_E$ (*nat-subst*$_E$ e x e') ⊆ *RID*$_E$ e ∪ *RID*$_E$ e'
proof −
fix v' :: ('*r*,'*l*,*nat*) *val*
have
(∀ x e. ∀ ζ. *RID*$_E$ (*nat-subst*$_V$ e x ($\mathcal{RV}_V$ ζ v')) ⊆ *RID*$_E$ e ∪ *RID*$_V$ ($\mathcal{RV}_V$ ζ v')) ∧
(∀ x e. ∀ ζ. *RID*$_E$ (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ ζ e')) ⊆ *RID*$_E$ e ∪ *RID*$_E$ ($\mathcal{RV}_E$ ζ e'))
by (*induct rule*: *val-expr.induct*) (*auto 0 4 simp add*: *expr.set-map*(*1*))
hence *RID*$_E$ (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ *id* e')) ⊆ *RID*$_E$ e ∪ *RID*$_E$ ($\mathcal{RV}_E$ *id* e') **by** *blast*
thus *?thesis* **by** *simp*
qed

lemma *nat-subst$_E$-introduces-no-lids*:
fixes e' :: ('r,'l,nat) *expr*
shows LID_E (*nat-subst$_E$* e x e') $\subseteq$ LID_E e $\cup$ LID_E e'
proof −
fix v' :: ('r,'l,nat) *val*
have
$(\forall x\ e.\ \forall \zeta.\ LID_E$ (*nat-subst$_V$* e x ($\mathcal{RV}_V$ ζ v')) $\subseteq$ LID_E e $\cup$ LID_V ($\mathcal{RV}_V$ ζ v'))
$\wedge$
$(\forall x\ e.\ \forall \zeta.\ LID_E$ (*nat-subst$_E$* e x ($\mathcal{RV}_E$ ζ e')) $\subseteq$ LID_E e $\cup$ LID_E ($\mathcal{RV}_E$ ζ e'))
by (*induct rule*: *val-expr.induct*) (*auto 0 4 simp add*: *expr.set-map*(2))
hence LID_E (*nat-subst$_E$* e x ($\mathcal{RV}_E$ *id* e')) $\subseteq$ LID_E e $\cup$ LID_E ($\mathcal{RV}_E$ *id* e') **by** *blast*
thus *?thesis* **by** *simp*
qed

lemma *nat-subst$_E$-models-substitution*: *substitution nat-subst$_E$*
by (*simp add*: *nat-subst$_E$-distr nat-subst$_E$-introduces-no-lids nat-subst$_E$-introduces-no-rids substitution-def*)

end

5 Operational Semantics

This theory defines the operational semantics of the concurrent revisions model. It also introduces a relaxed formulation of the operational semantics, and proves the main result required for establishing their equivalence.

theory *OperationalSemantics*
imports *Substitution*
begin

context *substitution*
begin

5.1 Definition

inductive *revision-step* :: 'r ⇒ ('r,'l,'v) *global-state* ⇒ ('r,'l,'v) *global-state* ⇒ *bool*
where
app: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$[*Apply* (*VE* (*Lambda* x e)) (*VE* v)]) ⟹ *revision-step* r s (s(r ↦ (σ, τ, $\mathcal{E}$[*subst* (*VE* v) x e])))
| *ifTrue*: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$[*Ite* (*VE* (*CV T*)) *e1 e2*]) ⟹ *revision-step* r s (s(r ↦ (σ, τ, $\mathcal{E}$[*e1*])))
| *ifFalse*: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$[*Ite* (*VE* (*CV F*)) *e1 e2*]) ⟹ *revision-step* r s (s(r ↦ (σ, τ, $\mathcal{E}$[*e2*])))

| *new*: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$[*Ref* (*VE* v)]) ⟹ $l \notin LID_G\ s$ ⟹ *revision-step* r s (s(r ↦ (σ, τ(l ↦ v), $\mathcal{E}$[*VE* (*Loc* l)])))

| *get*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Read\ (VE\ (Loc\ l))])\ \Longrightarrow\ l \in dom\ (\sigma;;\tau)\ \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[VE\ (the\ ((\sigma;;\tau)\ l))])))$
| *set*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Assign\ (VE\ (Loc\ l))\ (VE\ v)])\ \Longrightarrow\ l \in dom\ (\sigma;;\tau)\ \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau(l \mapsto v),\ \mathcal{E}[VE\ (CV\ Unit)])))$

| *fork*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rfork\ e])\ \Longrightarrow\ r' \notin RID_G\ s\ \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[VE\ (Rid\ r')]),\ r' \mapsto (\sigma;;\tau,\ \varepsilon,\ e)))$
| *join*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rjoin\ (VE\ (Rid\ r'))])\ \Longrightarrow\ s\ r' = Some\ (\sigma',\ \tau',\ VE\ v)$ $\Longrightarrow$ *revision-step* $r\ s\ (s(r := Some\ (\sigma,\ (\tau;;\tau'),\ \mathcal{E}[VE\ (CV\ Unit)]),\ r' := None))$
| $join_\varepsilon$: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rjoin\ (VE\ (Rid\ r'))])\ \Longrightarrow\ s\ r' = None\ \Longrightarrow$ *revision-step* $r\ s\ \varepsilon$

inductive-cases *revision-stepE* [*elim*, *consumes 1*, *case-names app ifTrue ifFalse new get set fork join* $join_\varepsilon$]:
revision-step r s s'

5.2 Introduction lemmas for identifiers

lemma *only-new-introduces-lids* [*intro*, *dest*]:
assumes
step: *revision-step r s s'* **and**
not-new: $\bigwedge\sigma\ \tau\ \mathcal{E}\ v.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Ref\ (VE\ v)])$
shows $LID_G\ s' \subseteq LID_G\ s$
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case *fork*
thus *?thesis* **by** (*auto simp add*: *fun-upd-twist ID-distr-global-conditional*)
next
case (*join - - - r' - - -*)
hence $r \neq r'$ **by** *auto*
thus *?thesis* **using** *join* **by** (*auto simp add*: *fun-upd-twist dest*!: *in-combination-in-component*)
qed (*auto simp add*: *not-new fun-upd-twist ID-distr-global-conditional dest*: $LID_S I(2)$)

lemma *only-fork-introduces-rids* [*intro*, *dest*]:
assumes
step: *revision-step r s s'* **and**
not-fork: $\bigwedge\sigma\ \tau\ \mathcal{E}\ e.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Rfork\ e])$
shows $RID_G\ s' \subseteq RID_G\ s$
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
next
case *get*
then show *?thesis* **by** (*auto simp add*: *ID-distr-global-conditional*)
next
case *fork*
then show *?thesis* **by** (*simp add*: *not-fork*)
next
case (*join - - -r' - - -*)
hence $r \neq r'$ **by** *auto*
then show *?thesis* **using** *join* **by** (*auto simp add*: *fun-upd-twist dest*!: *in-combination-in-component*)
qed (*auto simp add*: *ID-distr-global-conditional*)

lemma *only-fork-introduces-rids'* [*dest*]:
assumes
step: *revision-step r s s'* **and**
not-fork: $\bigwedge\sigma\ \tau\ \mathcal{E}\ e.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Rfork\ e])$
shows $r' \notin RID_G\ s \Longrightarrow r' \notin RID_G\ s'$
using *assms* **by** *blast*

lemma *only-new-introduces-lids'* [*dest*]:
assumes
step: *revision-step r s s'* **and**
not-new: $\bigwedge\sigma\ \tau\ \mathcal{E}\ v.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Ref\ (VE\ v)])$
shows $l \notin LID_G\ s \Longrightarrow l \notin LID_G\ s'$
using *assms* **by** *blast*

5.3 Domain subsumption

5.3.1 Definitions

definition *domains-subsume* :: $('r,'l,'v)$ *local-state* $\Rightarrow$ *bool* $(\langle\mathcal{S}\rangle)$ **where**
$\mathcal{S}\ ls = (LID_L\ ls \subseteq doms\ ls)$

definition *domains-subsume-globally* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* $(\langle\mathcal{S}_G\rangle)$ **where**
$\mathcal{S}_G\ s = (\forall r\ ls.\ s\ r = Some\ ls \longrightarrow \mathcal{S}\ ls)$

lemma *domains-subsume-globallyI* [*intro*]:
$(\bigwedge r\ \sigma\ \tau\ e.\ s\ r = Some\ (\sigma,\tau,e) \Longrightarrow \mathcal{S}\ (\sigma,\tau,e)) \Longrightarrow$ *domains-subsume-globally s*
using *domains-subsume-globally-def* **by** *auto*

definition *subsumes-accessible* :: $'r \Rightarrow 'r \Rightarrow ('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* $(\langle\mathcal{A}\rangle)$ **where**
$\mathcal{A}\ r_1\ r_2\ s = (r_2 \in RID_L\ (the\ (s\ r_1)) \longrightarrow (LID_S\ ((the\ (s\ r_2))_\sigma) \subseteq doms\ (the\ (s\ r_1))))$

lemma *subsumes-accessibleI* [*intro*]:
$(r_2 \in RID_L\ (the\ (s\ r_1)) \Longrightarrow LID_S\ ((the\ (s\ r_2))_\sigma) \subseteq doms\ (the\ (s\ r_1))) \Longrightarrow \mathcal{A}\ r_1\ r_2\ s$
using *subsumes-accessible-def* **by** *auto*

definition *subsumes-accessible-globally* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* $(\langle\mathcal{A}_G\rangle)$ **where**
$\mathcal{A}_G\ s = (\forall r_1\ r_2.\ r_1 \in dom\ s \longrightarrow r_2 \in dom\ s \longrightarrow \mathcal{A}\ r_1\ r_2\ s)$

lemma *subsumes-accessible-globallyI* [*intro*]:
$(\bigwedge r_1\ \sigma_1\ \tau_1\ e_1\ r_2\ \sigma_2\ \tau_2\ e_2.\ s\ r_1 = Some\ (\sigma_1,\tau_1,e_1) \Longrightarrow s\ r_2 = Some\ (\sigma_2,\tau_2,e_2) \Longrightarrow \mathcal{A}\ r_1\ r_2\ s) \Longrightarrow \mathcal{A}_G\ s$
using *subsumes-accessible-globally-def* **by** *auto*

5.3.2 The theorem

lemma $\mathcal{S}_G$*-imp-*$\mathcal{A}$*-refl*:

assumes
$\mathcal{S}_G$-*s*: $\mathcal{S}_G\ s$ **and**
r-in-dom: $r \in dom\ s$
shows $\mathcal{A}\ r\ r\ s$
using *assms* **by** (*auto simp add*: *domains-subsume-def domains-subsume-globally-def subsumes-accessibleI*)

lemma *step-preserves-*$\mathcal{S}_G$*-and-*$\mathcal{A}_G$:
assumes
step: *revision-step r s s'* **and**
$\mathcal{S}_G$-*s*: $\mathcal{S}_G\ s$ **and**
$\mathcal{A}_G$-*s*: $\mathcal{A}_G\ s$
shows $\mathcal{S}_G\ s'\ \mathcal{A}_G\ s'$
proof −
show $\mathcal{S}_G\ s'$
proof (*rule domains-subsume-globallyI*)
fix $r'\ \sigma\ \tau\ e$
assume s'-*r*: $s'\ r' = Some\ (\sigma,\tau,e)$
show $\mathcal{S}\ (\sigma,\tau,e)$
proof (*cases* $s'\ r' = s\ r'$)
case *True*
then show *?thesis* **using** $\mathcal{S}_G$-*s domains-subsume-globally-def* s'-*r* **by** *auto*
next
case r'-*was-updated*: *False*
show *?thesis*
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case (*app* $\sigma'\ \tau'\ \mathcal{E}'$ - $e'\ v'$)
have $r = r'$ **by** (*metis fun-upd-apply app(1)* r'-*was-updated*)
have $LID_L\ (the\ (s'\ r)) \subseteq LID_S\ \sigma' \cup LID_S\ \tau' \cup LID_C\ \mathcal{E}' \cup LID_E\ e' \cup LID_V\ v'$ **using** *app(1)* **by** *auto*
also have ... = $LID_L\ (the\ (s\ r))$ **using** *app(2)* **by** *auto*
also have ... ⊆ *doms* (*the* (*s r*))
by (*metis* $\mathcal{S}_G$-*s domains-subsume-def domains-subsume-globally-def local.app(2) option.sel*)
also have ... = *doms* (*the* ($s'\ r$)) **using** *app* **by** *simp*
finally have $\mathcal{S}$ (*the* ($s'\ r$)) **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› s'-*r*)
next
case *ifTrue*
have $r = r'$ **by** (*metis fun-upd-apply ifTrue(1)* r'-*was-updated*)
have $LID_L\ (the\ (s'\ r)) \subseteq LID_L\ (the\ (s\ r))$ **using** *ifTrue* **by** *auto*
also have ... ⊆ *doms* (*the* (*s r*))
by (*metis* $\mathcal{S}_G$-*s domains-subsume-def domains-subsume-globally-def ifTrue(2) option.sel*)
also have ... = *doms* (*the* ($s'\ r$)) **by** (*simp add*: *ifTrue*)
finally have $\mathcal{S}$ (*the* ($s'\ r$)) **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› s'-*r*)
next
case *ifFalse*

have $r = r'$ **by** (*metis fun-upd-apply ifFalse(1) r'-was-updated*)
have $LID_L\ (the\ (s'\ r)) \subseteq LID_L\ (the\ (s\ r))$ **using** *ifFalse* **by** *auto*
also have ... $\subseteq$ *doms* (*the* (*s r*))
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def ifFalse(2) option.sel*)
also have ... = *doms* (*the* (*s' r*)) **by** (*simp add*: *ifFalse*)
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› *s'-r*)
next
case (*new* $\sigma'\ \tau'\ \mathcal{E}'\ v'\ l'$)
have $r = r'$ **by** (*metis fun-upd-apply new(1) r'-was-updated*)
have $LID_L\ (the\ (s'\ r)) = insert\ l'\ (LID_S\ \sigma' \cup LID_S\ \tau' \cup LID_V\ v' \cup LID_C\ \mathcal{E}')$
proof −
have $l' \notin LID_S\ \tau'$ **using** *new(2−3)* **by** *auto*
thus *?thesis* **using** *new(1)* **by** *auto*
qed
also have ... = $insert\ l'\ (LID_L\ (the\ (s\ r)))$ **using** *new* **by** *auto*
also have ... $\subseteq$ $insert\ l'\ (doms\ (the\ (s\ r)))$
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def insert-mono new(2) option.sel*)
also have ... = *doms* (*the* (*s' r*)) **using** *new* **by** *auto*
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› *s'-r*)
next
case *get*
have $r = r'$ **by** (*metis fun-upd-apply get(1) r'-was-updated*)
have $LID_L\ (the\ (s'\ r)) = LID_L\ (the\ (s\ r))$ **using** *get* **by** *auto*
also have ... $\subseteq$ *doms* (*the* (*s r*))
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def get(2) option.sel*)
also have ... = *doms* (*the* (*s' r*)) **by** (*simp add*: *get(1−2)*)
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› *s'-r*)
next
case *set*
have $r = r'$ **by** (*metis fun-upd-apply set(1) r'-was-updated*)
have $LID_L\ (the\ (s'\ r)) \subseteq LID_L\ (the\ (s\ r))$ **using** *set(1−2)* **by** *auto*
also have ... $\subseteq$ *doms* (*the* (*s r*))
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def set(2) option.sel*)
also have ... $\subseteq$ *doms* (*the* (*s' r*)) **using** *set(1−2)* **by** *auto*
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› *s'-r*)
next
case (*fork* $\sigma'\ \tau'$ - - r'')
have $r = r' \vee r'' = r'$ **using** *fork r'-was-updated* **by** *auto*
then show *?thesis*
proof (*rule disjE*)

assume $r = r'$
have $LID_L\ (the\ (s'\ r)) \subseteq LID_L\ (the\ (s\ r))$ **using** *fork(1−2)* **by** *auto*
also have $... \subseteq doms\ (the\ (s\ r))$
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def fork(2) option.sel*)
also have $... = doms\ (the\ (s'\ r))$ **using** *fork* **by** *auto*
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r = r'$› *s'-r*)
next
assume $r'' = r'$
have $LID_L\ (the\ (s'\ r'')) \subseteq LID_L\ (the\ (s\ r))$ **using** *fork(1−2)* **by** *auto*
also have $... \subseteq doms\ (the\ (s\ r))$
by (*metis* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def fork(2) option.sel*)
also have $... = dom\ \sigma' \cup dom\ \tau'$ **using** *fork* **by** *simp*
also have $... = dom\ (\sigma';;\tau')$ **by** (*simp add*: *dom-combination-dom-union*)
also have $... = doms\ (the\ (s'\ r''))$ **using** *fork* **by** *simp*
finally have $\mathcal{S}\ (the\ (s'\ r''))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **by** (*simp add*: ‹$r'' = r'$› *s'-r*)
qed
next
case (*join* $\sigma'\ \tau'$ - $r''\ \sigma''\ \tau''$ -)
have $r' = r$ **by** (*metis fun-upd-def join(1) option.simps(3) r'-was-updated s'-r*)
have $LID_L\ (the\ (s'\ r)) \subseteq LID_L\ (the\ (s\ r)) \cup LID_S\ \tau''$ **using** *join* **by** *auto*
also have $... \subseteq doms\ (the\ (s\ r)) \cup LID_S\ \tau''$
by (*metis Un-mono* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def join(2) option.sel subset-refl*)
also have $... \subseteq doms\ (the\ (s\ r)) \cup LID_L\ (the\ (s\ r''))$ **using** *join(3)* **by** *auto*
also have $... \subseteq doms\ (the\ (s\ r)) \cup doms\ (the\ (s\ r''))$
by (*metis* (*no-types*, *lifting*) *Un-absorb* $\mathcal{S}_G$*-s domains-subsume-def domains-subsume-globally-def join(3) option.sel sup.orderI sup-mono*)
also have $... = dom\ \sigma' \cup dom\ \tau' \cup dom\ \sigma'' \cup dom\ \tau''$ **using** *join* **by** *auto*
also have $... \subseteq dom\ \sigma' \cup dom\ \tau' \cup LID_S\ \sigma'' \cup dom\ \tau''$ **by** *auto*
also have $... \subseteq dom\ \sigma' \cup dom\ \tau' \cup dom\ \sigma' \cup dom\ \tau' \cup dom\ \tau''$
proof −
have *r-r''*: $\mathcal{A}\ r\ r''\ s$ **using** $\mathcal{A}_G$*-s join(2−3) subsumes-accessible-globally-def* **by** *auto*
have *r-accesses-r''*: $r'' \in RID_L\ (the\ (s\ r))$ **using** *join* **by** *auto*
have $LID_S\ \sigma'' \subseteq dom\ \sigma' \cup dom\ \tau'$ **using** *join subsumes-accessible-def r-r'' r-accesses-r''* **by** *auto*
thus *?thesis* **by** *auto*
qed
also have $... = dom\ \sigma' \cup dom\ \tau' \cup dom\ \tau''$ **by** *auto*
also have $... = dom\ \sigma' \cup dom\ (\tau';;\tau'')$ **by** (*auto simp add*: *dom-combination-dom-union*)
also have $... = doms\ (the\ (s'\ r))$ **using** *join* **by** *auto*
finally have $\mathcal{S}\ (the\ (s'\ r))$ **by** (*simp add*: *domains-subsume-def*)
thus *?thesis* **using** ‹$r' = r$› *s'-r* **by** *auto*
next

```
      case join_ε
      then show ?thesis using s'-r by blast
    qed
  qed
 qed
 show A_G s'
 proof (rule subsumes-accessible-globallyI)
  fix r_1 σ_1 τ_1 e_1 r_2 σ_2 τ_2 e_2
  assume
    s'-r_1: s' r_1 = Some (σ_1, τ_1, e_1) and
    s'-r_2: s' r_2 = Some (σ_2, τ_2, e_2)
  show A r_1 r_2 s'
  proof (cases r_1 = r_2)
    case True
    then show ?thesis using S_G-imp-A-refl ‹S_G s'› s'-r_1 by blast
  next
    case r_1-neq-r_2: False
     have r_1-nor-r_2-updated-implies-thesis: s' r_1 = s r_1 ⟹ s' r_2 = s r_2 ⟹
?thesis
    proof −
      assume r_1-unchanged: s' r_1 = s r_1 and r_2-unchanged: s' r_2 = s r_2
      have A r_1 r_2 s
       by (metis A_G-s domIff option.discI r_1-unchanged r_2-unchanged s'-r_1 s'-r_2
subsumes-accessible-globally-def)
    show ?thesis using ‹A r_1 r_2 s› r_1-unchanged r_2-unchanged subsumes-accessible-def
by auto
    qed
    have r_1-or-r_2-updated-implies-thesis: s' r_1 ≠ s r_1 ∨ s' r_2 ≠ s r_2 ⟹ ?thesis
    proof −
      assume r_1-or-r_2-updated: s' r_1 ≠ s r_1 ∨ s' r_2 ≠ s r_2
      show ?thesis
      proof (use step in ‹cases rule: revision-stepE›)
        case app
        have r_1 = r ∨ r_2 = r by (metis fun-upd-other app(1) r_1-or-r_2-updated)
        then show ?thesis
        proof (rule disjE)
          assume r_1-eq-r: r_1 = r
          show A r_1 r_2 s'
          proof (rule subsumes-accessibleI)
            assume r_2-in-s'-r_1: r_2 ∈ RID_L (the (s' r_1))
            have LID_S ((the (s' r_2))_σ) ⊆ LID_S ((the (s r_2))_σ) using app by auto
            also have ... ⊆ doms (the (s r_1))
            proof −
             have r_2-in-s-r_1: r_2 ∈ RID_L (the (s r_1)) using app r_2-in-s'-r_1 r_1-eq-r
by auto
              have A r_1 r_2 s
                    by (metis A_G-s domI fun-upd-other app r_1-eq-r s'-r_2 sub-
sumes-accessible-globally-def)
             show ?thesis using ‹A r_1 r_2 s› r_2-in-s-r_1 subsumes-accessible-def
```

by *blast*
qed
also have ... $\subseteq$ *doms* (*the* (s' r_1)) **using** *app* **by** *auto*
finally show LID_S (*the* (s' r_2)$_\sigma$) $\subseteq$ *doms* (*the* (s' r_1)) **by** *simp*
qed
next
assume r_2-*eq-r*: $r_2 = r$
show $\mathcal{A}$ r_1 r_2 s'
proof (*rule subsumes-accessibleI*)
assume r_2-*in-s'-r*$_1$: $r_2 \in RID_L$ (*the* (s' r_1))
have LID_S (*the* (s' r_2)$_\sigma$) = LID_S (*the* (s r_2)$_\sigma$) **using** *app* **by** *auto*
also have ... $\subseteq$ *doms* (*the* (s r_1))
proof −
have r_2-*in-s-r*$_1$: $r_2 \in RID_L$ (*the* (s r_1)) **using** *app*(*1*) r_1-*neq-r*$_2$ r_2-*eq-r* r_2-*in-s'-r*$_1$ **by** *auto*
have $\mathcal{A}$ r_1 r_2 s
by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$-*s domIff fun-upd-other app option.discI* r_2-*eq-r* s'-r_1 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}$ r_1 r_2 s› r_2-*in-s-r*$_1$ *subsumes-accessible-def* **by** *blast*
qed
also have ... $\subseteq$ *doms* (*the* (s' r_1)) **by** (*simp add*: *app*)
finally show LID_S (*the* (s' r_2)$_\sigma$) $\subseteq$ *doms* (*the* (s' r_1)) **by** *simp*
qed
qed
next
case *ifTrue*
have $r_1 = r \vee r_2 = r$ **by** (*metis fun-upd-other ifTrue*(*1*) r_1-*or-r*$_2$-*updated*)
then show *?thesis*
proof (*rule disjE*)
assume r_1-*eq-r*: $r_1 = r$
show $\mathcal{A}$ r_1 r_2 s'
proof (*rule subsumes-accessibleI*)
assume r_2-*in-s'-r*$_1$: $r_2 \in RID_L$ (*the* (s' r_1))
have LID_S ((*the* (s' r_2))$_\sigma$) $\subseteq$ LID_S ((*the* (s r_2))$_\sigma$) **using** *ifTrue* **by** *auto*
also have ... $\subseteq$ *doms* (*the* (s r_1))
proof −
have r_2-*in-s-r*$_1$: $r_2 \in RID_L$ (*the* (s r_1)) **using** *ifTrue* r_2-*in-s'-r*$_1$ r_1-*eq-r* **by** *auto*
have $\mathcal{A}$ r_1 r_2 s
by (*metis* $\mathcal{A}_G$-*s domI fun-upd-other ifTrue* r_1-*eq-r* s'-r_2 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}$ r_1 r_2 s› r_2-*in-s-r*$_1$ *subsumes-accessible-def* **by** *blast*
qed
also have ... $\subseteq$ *doms* (*the* (s' r_1)) **using** *ifTrue* **by** *auto*
finally show LID_S (*the* (s' r_2)$_\sigma$) $\subseteq$ *doms* (*the* (s' r_1)) **by** *simp*
qed

```
      next
        assume r2-eq-r: r2 = r
        show 𝒜 r1 r2 s'
        proof (rule subsumes-accessibleI)
          assume r2-in-s'-r1: r2 ∈ RID_L (the (s' r1))
          have LID_S (the (s' r2)_σ) = LID_S (the (s r2)_σ) using ifTrue by auto
          also have ... ⊆ doms (the (s r1))
          proof −
            have r2-in-s-r1: r2 ∈ RID_L (the (s r1)) using ifTrue(1) r1-neq-r2
r2-eq-r r2-in-s'-r1 by auto
            have 𝒜 r1 r2 s
              by (metis (no-types, lifting) 𝒜_G-s domIff fun-upd-other ifTrue
option.discI r2-eq-r s'-r1 subsumes-accessible-globally-def)
            show ?thesis using ‹𝒜 r1 r2 s› r2-in-s-r1 subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r1)) by (simp add: ifTrue)
          finally show LID_S (the (s' r2)_σ) ⊆ doms (the (s' r1)) by simp
        qed
      qed
    next
      case ifFalse
    have r1 = r ∨ r2 = r by (metis fun-upd-other ifFalse(1) r1-or-r2-updated)
      then show ?thesis
      proof (rule disjE)
        assume r1-eq-r: r1 = r
        show 𝒜 r1 r2 s'
        proof (rule subsumes-accessibleI)
          assume r2-in-s'-r1: r2 ∈ RID_L (the (s' r1))
          have LID_S ((the (s' r2))_σ) ⊆ LID_S ((the (s r2))_σ) using ifFalse by
auto
          also have ... ⊆ doms (the (s r1))
          proof −
            have r2-in-s-r1: r2 ∈ RID_L (the (s r1)) using ifFalse r2-in-s'-r1
r1-eq-r by auto
            have 𝒜 r1 r2 s
              by (metis 𝒜_G-s domI fun-upd-other ifFalse r1-eq-r s'-r2 sub-
sumes-accessible-globally-def)
            show ?thesis using ‹𝒜 r1 r2 s› r2-in-s-r1 subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r1)) using ifFalse by auto
          finally show LID_S (the (s' r2)_σ) ⊆ doms (the (s' r1)) by simp
        qed
      next
        assume r2-eq-r: r2 = r
        show 𝒜 r1 r2 s'
        proof (rule subsumes-accessibleI)
          assume r2-in-s'-r1: r2 ∈ RID_L (the (s' r1))
```

```
          have LID_S (the (s' r_2)_σ) = LID_S (the (s r_2)_σ) using ifFalse by auto
          also have ... ⊆ doms (the (s r_1))
          proof −
            have r_2-in-s-r_1: r_2 ∈ RID_L (the (s r_1)) using ifFalse(1) r_1-neq-r_2
r_2-eq-r r_2-in-s'-r_1 by auto
           have A r_1 r_2 s
                by (metis (no-types, lifting) A_G-s domIff fun-upd-other ifFalse
option.discI r_2-eq-r s'-r_1 subsumes-accessible-globally-def)
            show ?thesis using ‹A r_1 r_2 s› r_2-in-s-r_1 subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r_1)) by (simp add: ifFalse)
          finally show LID_S (the (s' r_2)_σ) ⊆ doms (the (s' r_1)) by simp
        qed
      qed
    next
      case new
      have r_1 = r ∨ r_2 = r by (metis fun-upd-other new(1) r_1-or-r_2-updated)
      then show ?thesis
      proof (rule disjE)
        assume r_1-eq-r: r_1 = r
        show A r_1 r_2 s'
        proof (rule subsumes-accessibleI)
          assume r_2-in-s'-r_1: r_2 ∈ RID_L (the (s' r_1))
         have LID_S ((the (s' r_2))_σ) ⊆ LID_S ((the (s r_2))_σ) using new by auto
          also have ... ⊆ doms (the (s r_1))
          proof −
           have r_2-in-s-r_1: r_2 ∈ RID_L (the (s r_1)) using new r_2-in-s'-r_1 r_1-eq-r
by auto
            have A r_1 r_2 s
                 by (metis A_G-s domI fun-upd-other new(1−2) r_1-eq-r s'-r_2
subsumes-accessible-globally-def)
            show ?thesis using ‹A r_1 r_2 s› r_2-in-s-r_1 subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r_1)) using new by auto
          finally show LID_S (the (s' r_2)_σ) ⊆ doms (the (s' r_1)) by simp
        qed
      next
        assume r_2-eq-r: r_2 = r
        show A r_1 r_2 s'
        proof (rule subsumes-accessibleI)
          assume r_2-in-s'-r_1: r_2 ∈ RID_L (the (s' r_1))
          have LID_S (the (s' r_2)_σ) = LID_S (the (s r_2)_σ) using new by auto
          also have ... ⊆ doms (the (s r_1))
          proof −
            have r_2-in-s-r_1: r_2 ∈ RID_L (the (s r_1)) using new(1) r_1-neq-r_2
r_2-eq-r r_2-in-s'-r_1 by auto
           have A r_1 r_2 s
```

by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$*-s domIff fun-upd-other new*(*1*−*2*) *option.discI* r_2*-eq-r* s'-r_1 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}$ r_1 r_2 s› r_2*-in-s-*r_1 *subsumes-accessible-def* **by** *blast*
qed
also have ... ⊆ *doms* (*the* (s' r_1)) **by** (*auto simp add*: *new*)
finally show LID_S (*the* (s' r_2)$_\sigma$) ⊆ *doms* (*the* (s' r_1)) **by** *simp*
qed
qed
next
case *get*
have $r_1 = r \vee r_2 = r$ **by** (*metis fun-upd-other get*(*1*) r_1*-or-*r_2*-updated*)
then show *?thesis*
proof (*rule disjE*)
assume r_1*-eq-r*: $r_1 = r$
show $\mathcal{A}$ r_1 r_2 s'
proof (*rule subsumes-accessibleI*)
assume r_2*-in-s'-*r_1: $r_2 \in RID_L$ (*the* (s' r_1))
have LID_S ((*the* (s' r_2))$_\sigma$) ⊆ LID_S ((*the* (s r_2))$_\sigma$) **using** *get* **by** *auto*
also have ... ⊆ *doms* (*the* (s r_1))
proof −
have r_2*-in-s-*r_1: $r_2 \in RID_L$ (*the* (s r_1)) **using** *get* r_2*-in-s'-*r_1 r_1*-eq-r* **apply** *auto*
by (*meson* $RID_S I$)
have $\mathcal{A}$ r_1 r_2 s
by (*metis* $\mathcal{A}_G$*-s domI fun-upd-other get*(*1*−*2*) r_1*-eq-r* s'-r_2 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}$ r_1 r_2 s› r_2*-in-s-*r_1 *subsumes-accessible-def* **by** *blast*
qed
also have ... ⊆ *doms* (*the* (s' r_1)) **using** *get* **by** *auto*
finally show LID_S (*the* (s' r_2)$_\sigma$) ⊆ *doms* (*the* (s' r_1)) **by** *simp*
qed
next
assume r_2*-eq-r*: $r_2 = r$
show $\mathcal{A}$ r_1 r_2 s'
proof (*rule subsumes-accessibleI*)
assume r_2*-in-s'-*r_1: $r_2 \in RID_L$ (*the* (s' r_1))
have LID_S (*the* (s' r_2)$_\sigma$) = LID_S (*the* (s r_2)$_\sigma$) **using** *get* **by** *auto*
also have ... ⊆ *doms* (*the* (s r_1))
proof −
have r_2*-in-s-*r_1: $r_2 \in RID_L$ (*the* (s r_1)) **using** *get*(*1*) r_1*-neq-*r_2 r_2*-eq-r* r_2*-in-s'-*r_1 **by** *auto*
have $\mathcal{A}$ r_1 r_2 s
by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$*-s domIff fun-upd-other get*(*1*−*2*) *option.discI* r_2*-eq-r* s'-r_1 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}$ r_1 r_2 s› r_2*-in-s-*r_1 *subsumes-accessible-def* **by** *blast*
qed

```
          also have ... ⊆ doms (the (s' r₁)) by (simp add: get)
          finally show LID_S (the (s' r₂)_σ) ⊆ doms (the (s' r₁)) by simp
        qed
      qed
    next
      case set
      have r₁ = r ∨ r₂ = r by (metis fun-upd-other set(1) r₁-or-r₂-updated)
      then show ?thesis
      proof (rule disjE)
        assume r₁-eq-r: r₁ = r
        show 𝒜 r₁ r₂ s'
        proof (rule subsumes-accessibleI)
          assume r₂-in-s'-r₁: r₂ ∈ RID_L (the (s' r₁))
          have LID_S ((the (s' r₂))_σ) ⊆ LID_S ((the (s r₂))_σ) using set by auto
          also have ... ⊆ doms (the (s r₁))
          proof −
            have r₂-in-s-r₁: r₂ ∈ RID_L (the (s r₁)) using set r₂-in-s'-r₁ r₁-eq-r
by auto
            have 𝒜 r₁ r₂ s
              by (metis 𝒜_G-s domI fun-upd-other set(1−2) r₁-eq-r s'-r₂
subsumes-accessible-globally-def)
            show ?thesis using ‹𝒜 r₁ r₂ s› r₂-in-s-r₁ subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r₁)) using set by auto
          finally show LID_S (the (s' r₂)_σ) ⊆ doms (the (s' r₁)) by simp
        qed
      next
        assume r₂-eq-r: r₂ = r
        show 𝒜 r₁ r₂ s'
        proof (rule subsumes-accessibleI)
          assume r₂-in-s'-r₁: r₂ ∈ RID_L (the (s' r₁))
          have LID_S (the (s' r₂)_σ) = LID_S (the (s r₂)_σ) using set by auto
          also have ... ⊆ doms (the (s r₁))
          proof −
            have r₂-in-s-r₁: r₂ ∈ RID_L (the (s r₁)) using set(1) r₁-neq-r₂ r₂-eq-r
r₂-in-s'-r₁ by auto
            have 𝒜 r₁ r₂ s
              by (metis (no-types, lifting) 𝒜_G-s domIff fun-upd-other set(1−2)
option.discI r₂-eq-r s'-r₁ subsumes-accessible-globally-def)
            show ?thesis using ‹𝒜 r₁ r₂ s› r₂-in-s-r₁ subsumes-accessible-def
by blast
          qed
          also have ... ⊆ doms (the (s' r₁)) by (auto simp add: set)
          finally show LID_S (the (s' r₂)_σ) ⊆ doms (the (s' r₁)) by simp
        qed
      qed
    next
      case (fork σ τ ℰ e r')
```

have s'-r: $s'\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r')])$ **using** *fork* **by** *auto*
have s'-r': $s'\ r' = Some\ (\sigma;;\tau,\ \varepsilon,\ e)$
by (*simp add*: *local.fork*(*1*))
have *case1*: $r_1 = r \Longrightarrow r_2 \neq r \Longrightarrow r_2 \neq r' \Longrightarrow$ *?thesis*
proof (*rule subsumes-accessibleI*)
assume $r_1 = r\ r_2 \neq r\ r_2 \neq r'$
assume r_2-*in-s'*-r_1: $r_2 \in RID_L\ (the\ (s'\ r_1))$
have $LID_S\ ((the\ (s'\ r_2))_\sigma) \subseteq LID_S\ ((the\ (s\ r_2))_\sigma)$ **using** *fork*(*1*−*2*) **by** (*simp add*: ‹$r_2 \neq r'$›)
also have ... $\subseteq$ *doms* (*the* (*s* r_1))
proof −
have r_2-*in-s*-r_1: $r_2 \in RID_L\ (the\ (s\ r_1))$ **using** *fork* ‹$r_1 = r$› ‹$r_2 \neq r'$› r_2-*in-s'*-r_1 s'-r **by** *auto*
have $\mathcal{A}\ r_1\ r_2\ s$
by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$-*s* ‹$r_1 = r$› ‹$r_2 \neq r'$› *domIff fun-upd-other fork*(*1*−*2*) *option.discI s'*-r_2 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}\ r_1\ r_2\ s$› r_2-*in-s*-r_1 *subsumes-accessible-def* **by** *blast*
qed
also have ... $\subseteq$ *doms* (*the* (*s'* r_1)) **by** (*simp add*: ‹$r_1 = r$› *fork*(*2*) s'-r)
finally show $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq$ *doms* (*the* (*s'* r_1)) **by** *simp*
qed
have *case2*: $r_1 \neq r \Longrightarrow r_1 \neq r' \Longrightarrow r_2 = r \Longrightarrow$ *?thesis*
proof (*rule subsumes-accessibleI*)
assume $r_1 \neq r\ r_1 \neq r'\ r_2 = r$
assume r_2-*in-s'*-r_1: $r_2 \in RID_L\ (the\ (s'\ r_1))$
have $LID_S\ ((the\ (s'\ r_2))_\sigma) \subseteq LID_S\ ((the\ (s\ r_2))_\sigma)$
using ‹$r_1 \neq r'$› ‹$r_1 \neq r$› *fork* r_2-*in-s'*-r_1 s'-r_1 **by** *auto*
also have ... $\subseteq$ *doms* (*the* (*s* r_1))
proof −
have r_2-*in-s*-r_1: $r_2 \in RID_L\ (the\ (s\ r_1))$ **using** ‹$r_1 \neq r'$› ‹$r_1 \neq r$› *fork*(*1*) r_2-*in-s'*-r_1 **by** *auto*
have $\mathcal{A}\ r_1\ r_2\ s$
by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$-*s* ‹$r_1 \neq r'$› ‹$r_2 = r$› *domIff fun-upd-other fork*(*1*−*2*) *option.discI s'*-r_1 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}\ r_1\ r_2\ s$› r_2-*in-s*-r_1 *subsumes-accessible-def* **by** *auto*
qed
also have ... $\subseteq$ *doms* (*the* (*s'* r_1)) **by** (*simp add*: ‹$r_1 \neq r'$› ‹$r_1 \neq r$› *fork*(*1*))
finally show $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq$ *doms* (*the* (*s'* r_1)) **by** *simp*
qed
have *case3*: $r_1 = r' \Longrightarrow r_2 \neq r \Longrightarrow r_2 \neq r' \Longrightarrow$ *?thesis*
proof (*rule subsumes-accessibleI*)
fix *l*
assume $r_1 = r'\ r_2 \neq r\ r_2 \neq r'$
assume $r_2 \in RID_L\ (the\ (s'\ r_1))$
hence $r_2 \in RID_L\ (the\ (s\ r))$ **using** $RID_L I(3)$ ‹$r_1 = r'$› *fork*(*2*) s'-r' **by** *auto*

have $s\ r_2 = s'\ r_2$ **by** (*simp add*: ‹$r_2 \neq r'$› ‹$r_2 \neq r$› *fork(1)*)
hence $\mathcal{A}\ r\ r_2\ s$ **using** $\mathcal{A}_G$*-s fork(2) s'-r*$_2$ *subsumes-accessible-globally-def*
by *auto*
hence $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq doms\ (the\ (s\ r))$
by (*simp add*: ‹$r_2 \in RID_L\ (the\ (s\ r))$› ‹$s\ r_2 = s'\ r_2$› *subsumes-accessible-def*)
also have $... = dom\ \sigma \cup dom\ \tau$ **by** (*simp add*: *fork(2)*)
also have $... = dom\ (\sigma;;\tau)$ **by** (*simp add*: *dom-combination-dom-union*)
also have $... = doms\ (the\ (s'\ r'))$ **by** (*simp add*: *s'-r'*)
finally show $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq doms\ (the\ (s'\ r_1))$ **using** ‹$r_1 = r'$›
by *blast*
qed
have *case4*: $r_1 \neq r \Longrightarrow r_1 \neq r' \Longrightarrow r_2 = r' \Longrightarrow$ *?thesis*
proof −
assume $r_1 \neq r\ r_1 \neq r'\ r_2 = r'$
have $r_2 \notin RID_L\ (the\ (s\ r_1))$ **using** ‹$r_1 \neq r'$› ‹$r_1 \neq r$› ‹$r_2 = r'$› *fork(1,3) s'-r*$_1$ **by** *auto*
hence $r_2 \notin RID_L\ (the\ (s'\ r_1))$ **by** (*simp add*: ‹$r_1 \neq r'$› ‹$r_1 \neq r$› *fork(1)*)
thus *?thesis* **by** *blast*
qed
have *case5*: $r_1 = r \Longrightarrow r_2 = r' \Longrightarrow$ *?thesis*
proof (*rule subsumes-accessibleI*)
assume $r_1 = r\ r_2 = r'$
have $LID_S\ ((the\ (s'\ r_2))_\sigma) = LID_S\ (\sigma;;\tau)$ **by** (*simp add*: ‹$r_2 = r'$› *s'-r'*)
also have $... \subseteq LID_S\ \sigma \cup LID_S\ \tau$ **by** *auto*
also have $... \subseteq LID_L\ (the\ (s'\ r_1))$ **by** (*simp add*: ‹$r_1 = r$› *s'-r*)
also have $... \subseteq doms\ (the\ (s'\ r_1))$
by (*metis* ‹$\mathcal{S}_G\ s'$› ‹$r_1 = r$› *domains-subsume-def domains-subsume-globally-def option.sel s'-r*)
finally show $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq doms\ (the\ (s'\ r_1))$ **by** *simp*
qed
have *case6*: $r_1 = r' \Longrightarrow r_2 = r \Longrightarrow$ *?thesis*
proof (*rule subsumes-accessibleI*)
assume $r_1 = r'\ r_2 = r\ r_2 \in RID_L\ (the\ (s'\ r_1))$
have $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq LID_L\ (the\ (s'\ r_2))$ **by** (*simp add*: *s'-r*$_2$ *subsetI*)
also have $... \subseteq doms\ (the\ (s'\ r_2))$
using ‹$\mathcal{S}_G\ s'$› *domains-subsume-def domains-subsume-globally-def s'-r*$_2$
by *auto*
also have $... = dom\ \sigma \cup dom\ \tau$ **by** (*simp add*: ‹$r_2 = r$› *s'-r*)
also have $... = dom\ (\sigma;;\tau)$ **by** (*simp add*: *dom-combination-dom-union*)
finally show $LID_S\ (the\ (s'\ r_2)_\sigma) \subseteq doms\ (the\ (s'\ r_1))$
using ‹$r_1 = r'$› *s'-r'* **by** *auto*
qed
show *?thesis* **using** *case1 case2 case3 case4 case5 case6 fork(1) r*$_1$*-neq-r*$_2$ *r*$_1$*-nor-r*$_2$*-updated-implies-thesis* **by** *fastforce*
next
case (*join* $\sigma\ \tau\ \mathcal{E}\ r'\ \sigma'\ \tau'\ v$)
have $r_1 = r \vee r_2 = r$ **by** (*metis fun-upd-def join(1) option.simps(3)*

```
r1-or-r2-updated s'-r1 s'-r2)
        then show ?thesis
        proof (rule disjE)
          assume r1 = r
          show 𝒜 r1 r2 s'
          proof (rule subsumes-accessibleI)
            assume r2-in-s'-r1: r2 ∈ RID_L (the (s' r1))
            show LID_S (the (s' r2)_σ) ⊆ doms (the (s' r1))
            proof (cases r2 ∈ RID_S τ')
              case r2-in-τ': True
              have LID_S (the (s' r2)_σ) = LID_S (the (s r2)_σ)
                by (metis ‹r1 = r› fun-upd-def join(1) option.distinct(1) r1-neq-r2
s'-r2)
              also have ... ⊆ doms (the (s r'))
              proof −
                  have r2-in-s-r': r2 ∈ RID_L (the (s r')) by (simp add: join(3)
r2-in-τ')
                have 𝒜 r' r2 s
               by (metis 𝒜_G-s ‹r1 = r› domI fun-upd-def join(1) join(3) r1-neq-r2
s'-r2 subsumes-accessible-globally-def)
                  show ?thesis using ‹𝒜 r' r2 s› r2-in-s-r' subsumes-accessible-def
by blast
              qed
              also have ... = dom σ' ∪ dom τ' by (simp add: join(3))
              also have ... ⊆ LID_S σ' ∪ dom τ' by auto
              also have ... ⊆ dom σ ∪ dom τ ∪ dom τ'
              proof −
                have r' ∈ RID_L (the (s r)) by (simp add: join(2))
              have 𝒜 r r' s using 𝒜_G-s join(2−3) subsumes-accessible-globally-def
by auto
                  show ?thesis using ‹𝒜 r r' s› join(2−3) subsumes-accessible-def
by auto
              qed
                    also have ... = dom σ ∪ dom (τ;;τ') by (auto simp add:
dom-combination-dom-union)
             also have ... = doms (the (s' r1)) using join by (auto simp add: ‹r1
= r›)
              finally show ?thesis by simp
            next
              case r2-nin-τ': False
              have LID_S (the (s' r2)_σ) = LID_S (the (s r2)_σ)
                by (metis ‹r1 = r› fun-upd-def join(1) option.distinct(1) r1-neq-r2
s'-r2)
              also have ... ⊆ doms (the (s r1))
              proof −
                have r2-in-s-r1: r2 ∈ RID_L (the (s r))
                proof −
                  have RID_L (the (s' r1)) = RID_S σ ∪ RID_S (τ;;τ') ∪ RID_C ℰ
                by (metis (no-types, lifting) ID-distr-completion(1) ID-distr-local(2)
```

‹$r_1 = r$› *expr.simps(153) fun-upd-apply local.join(1) option.discI option.sel* s'-r_1 *sup-bot.right-neutral val.simps(66)*)
hence $r_2 \in RID_S\ \sigma \cup RID_S\ \tau \cup RID_C\ \mathcal{E}$ **using** r_2-*in-*s'-r_1 r_2-*nin-*τ' **by** *auto*
thus *?thesis* **by** (*simp add*: *join(2)*)
qed
have $\mathcal{A}\ r_1\ r_2\ s$ **by** (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$-*s* ‹$r_1 = r$› *join(1−2) domIff fun-upd-def option.discI* s'-r_2 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}\ r_1\ r_2\ s$› r_2-*in-s-*r_1 *subsumes-accessible-def* ‹$r_1 = r$› **by** *blast*
qed
also have ... = *dom* $\sigma \cup$ *dom* τ **by** (*simp add*: ‹$r_1 = r$› *join(2)*)
also have ... $\subseteq$ *dom* $\sigma \cup$ *dom* $(\tau;;\tau')$ **by** (*auto simp add*: *dom-combination-dom-union*)
also have ... = *doms* (*the* $(s'\ r_1)$) **using** *join* ‹$r_1 = r$› **by** *auto*
finally show *?thesis* **by** *simp*
qed
qed
next
assume $r_2 = r$
show $\mathcal{A}\ r_1\ r_2\ s'$
proof (*rule subsumes-accessibleI*)
assume r_2-*in-*s'-r_1: $r_2 \in RID_L$ (*the* $(s'\ r_1)$)
have LID_S (*the* $(s'\ r_2)_\sigma$) = LID_S (*the* $(s\ r_2)_\sigma$)
by (*metis* (*no-types*, *lifting*) *LID-snapshot.simps fun-upd-apply join(1−2) option.discI option.sel* s'-r_2)
also have ... $\subseteq$ *doms* (*the* $(s\ r_1)$)
proof −
have r_2-*in-s-*r_1: $r_2 \in RID_L$ (*the* $(s\ r_1)$)
by (*metis* ‹$r_2 = r$› *fun-upd-apply local.join(1) option.discI* r_1-*neq-*r_2 r_2-*in-*s'-r_1 s'-r_1)
have $\mathcal{A}\ r_1\ r_2\ s$
by (*metis* (*no-types*, *lifting*) $\mathcal{A}_G$-*s* ‹$r_2 = r$› *domIff fun-upd-apply join(1−2) option.discI* s'-r_1 *subsumes-accessible-globally-def*)
show *?thesis* **using** ‹$\mathcal{A}\ r_1\ r_2\ s$› r_2-*in-s-*r_1 *subsumes-accessible-def* **by** *blast*
qed
also have ... $\subseteq$ *doms* (*the* $(s'\ r_1)$)
by (*metis* ‹$r_2 = r$› *eq-refl fun-upd-def local.join(1) option.distinct(1)* r_1-*neq-*r_2 s'-r_1)
finally show LID_S (*the* $(s'\ r_2)_\sigma$) $\subseteq$ *doms* (*the* $(s'\ r_1)$) **by** *simp*
qed
qed
next
case $join_\varepsilon$
thus *?thesis* **using** s'-r_1 **by** *blast*
qed
qed
show $\mathcal{A}\ r_1\ r_2\ s'$ **using** r_1-*nor-*r_2-*updated-implies-thesis* r_1-*or-*r_2-*updated-implies-thesis*

by *blast*
qed
qed
qed

5.4 Relaxed definition of the operational semantics

inductive *revision-step-relaxed* :: $'r \Rightarrow ('r,'l,'v)$ *global-state* $\Rightarrow ('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* **where**
app: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Apply\ (VE\ (Lambda\ x\ e))\ (VE\ v)]) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[subst\ (VE\ v)\ x\ e])))$
| *ifTrue*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Ite\ (VE\ (CV\ T))\ e1\ e2]) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[e1])))$
| *ifFalse*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Ite\ (VE\ (CV\ F))\ e1\ e2]) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[e2])))$

| *new*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Ref\ (VE\ v)]) \Longrightarrow l \notin \bigcup \{\ doms\ ls \mid ls.\ ls \in ran\ s\ \} \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE\ (Loc\ l)])))$
| *get*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Read\ (VE\ (Loc\ l))]) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (the\ ((\sigma;;\tau)\ l))])))$
| *set*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Assign\ (VE\ (Loc\ l))\ (VE\ v)]) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE\ (CV\ Unit)])))$

| *fork*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Rfork\ e]) \Longrightarrow r' \notin RID_G\ s \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (Rid\ r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e)))$
| *join*: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Rjoin\ (VE\ (Rid\ r'))]) \Longrightarrow s\ r' = Some\ (\sigma', \tau', VE\ v) \Longrightarrow$ *revision-step-relaxed* $r\ s\ (s(r := Some\ (\sigma, (\tau;;\tau'), \mathcal{E}[VE\ (CV\ Unit)]), r' := None))$
| $join_\varepsilon$: $s\ r = Some\ (\sigma, \tau, \mathcal{E}[Rjoin\ (VE\ (Rid\ r'))]) \Longrightarrow s\ r' = None \Longrightarrow$ *revision-step-relaxed* $r\ s\ \varepsilon$

inductive-cases *revision-step-relaxedE* [*elim*, *consumes 1*, *case-names app ifTrue ifFalse new get set fork join* $join_\varepsilon$]:
revision-step-relaxed $r\ s\ s'$

end

end

6 Executions

This section contains all definitions required for reasoning about executions in the concurrent revisions model. It also contains a number of proofs for inductive variants. One of these proves the equivalence of the two definitions of the operational semantics. The others are required for proving determinacy.

theory *Executions*
imports *OperationalSemantics*

begin

context *substitution*
begin

6.1 Generalizing the original transition

6.1.1 Definition

definition *steps* :: $('r,'l,'v)$ *global-state rel* $(\langle[\rightsquigarrow]\rangle)$ **where**
steps = { (s,s') | s s'. $\exists\, r$. *revision-step* r s s' }

abbreviation *valid-step* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* (**infix** $\langle\rightsquigarrow\rangle$ *60*) **where**
$s \rightsquigarrow s' \equiv (s,s') \in [\rightsquigarrow]$

lemma *valid-stepI* [*intro*]:
revision-step r s $s' \Longrightarrow s \rightsquigarrow s'$
using *steps-def* **by** *auto*

lemma *valid-stepE* [*dest*]:
$s \rightsquigarrow s' \Longrightarrow \exists\, r$. *revision-step* r s s'
by (*simp add*: *steps-def*)

6.1.2 Closures

abbreviation *refl-trans-step-rel* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool*(**infix** $\langle\rightsquigarrow^*\rangle$ *60*) **where**
$s \rightsquigarrow^* s' \equiv (s,s') \in [\rightsquigarrow]^*$

abbreviation *refl-step-rel* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* (**infix** $\langle\rightsquigarrow^=\rangle$ *60*) **where**
$s \rightsquigarrow^= s' \equiv (s,s') \in [\rightsquigarrow]^=$

lemma *refl-rewritesI* [*intro*]: $s \rightsquigarrow s' \Longrightarrow s \rightsquigarrow^= s'$ **by** *blast*

6.2 Properties

abbreviation *program-expr* :: $('r,'l,'v)$ *expr* $\Rightarrow$ *bool* **where**
program-expr $e \equiv LID_E\ e = \{\} \wedge RID_E\ e = \{\}$

abbreviation *initializes* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *expr* $\Rightarrow$ *bool* **where**
initializes s $e \equiv \exists\, r$. $s = (\varepsilon(r \mapsto(\varepsilon,\varepsilon,e))) \wedge$ *program-expr* e

abbreviation *initial-state* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* **where**
initial-state $s \equiv \exists\, e$. *initializes* s e

definition *execution* :: $('r,'l,'v)$ *expr* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* **where**
execution e s $s' \equiv$ *initializes* s $e \wedge s \rightsquigarrow^* s'$

definition *maximal-execution* :: $('r,'l,'v)$ *expr* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* **where**
maximal-execution e s s' $\equiv$ *execution e s s'* $\wedge$ $(\nexists s''.\ s' \rightsquigarrow s'')$

definition *reachable* :: $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* **where**
reachable s $\equiv \exists e\ s'.$ *execution e s' s*

definition *terminates-in* :: $('r,'l,'v)$ *expr* $\Rightarrow$ $('r,'l,'v)$ *global-state* $\Rightarrow$ *bool* (**infix** $\langle\downarrow\rangle$ *60*) **where**
$e \downarrow s' \equiv \exists s.$ *maximal-execution e s s'*

6.3 Invariants

6.3.1 Inductive invariance

definition *inductive-invariant* :: $(('r,'l,'v)$ *global-state* $\Rightarrow$ *bool*$) \Rightarrow$ *bool* **where**
inductive-invariant P $\equiv$ $(\forall s.$ *initial-state s* $\longrightarrow$ *P s*$) \wedge (\forall s\ s'.\ s \rightsquigarrow s' \longrightarrow P\ s \longrightarrow P\ s')$

lemma *inductive-invariantI* [*intro*]:
$(\bigwedge s.$ *initial-state s* $\Longrightarrow$ *P s*$) \Longrightarrow (\bigwedge s\ s'.\ s \rightsquigarrow s' \Longrightarrow P\ s \Longrightarrow P\ s') \Longrightarrow$ *inductive-invariant P*
by (*auto simp add*: *inductive-invariant-def*)

lemma *inductive-invariant-is-execution-invariant*: *reachable s* $\Longrightarrow$ *inductive-invariant P* $\Longrightarrow$ *P s*
proof −
assume *reach*: *reachable s* **and** *ind-inv*: *inductive-invariant P*
then obtain *e initial n* **where** *initializes*: *initializes initial e* **and** *trace*: (*initial*,*s*) $\in [\rightsquigarrow]^{\frown}n$
by (*metis execution-def reachable-def rtrancl-power*)
thus *P s*
proof (*induct n arbitrary*: *s*)
case *0*
have *initial* = *s* **using** *0.prems*(*2*) **by** *auto*
hence *initial-state s* **using** *initializes* **by** *blast*
then show *?case* **using** *ind-inv inductive-invariant-def* **by** *auto*
next
case (*Suc n*)
obtain *s'* **where** *nfold*: (*initial*, *s'*) $\in [\rightsquigarrow]^{\frown}n$ **and** *step*: *s'* $\rightsquigarrow$ *s* **using** *Suc.prems*(*2*) **by** *auto*
have *P s'* **using** *Suc*(*1*) *nfold initializes* **by** *blast*
then show *?case* **using** *ind-inv step inductive-invariant-def* **by** *auto*
qed
qed

6.3.2 Subsumption is invariant

lemma *nice-ind-inv-is-inductive-invariant*: *inductive-invariant* $(\lambda s.\ \mathcal{S}_G\ s \wedge \mathcal{A}_G\ s)$

proof (*rule inductive-invariantI*)
fix s
assume *initial-state* s
then obtain e r **where** s: $s = \varepsilon(r \mapsto (\varepsilon,\ \varepsilon,\ e))$ **and** *prog-expr-e*: *program-expr* e **by** *blast*
show $\mathcal{S}_G\ s \wedge \mathcal{A}_G\ s$
proof (*rule conjI*)
show $\mathcal{S}_G\ s$
proof (*rule domains-subsume-globallyI*)
fix $r'\ \sigma'\ \tau'\ e'$
assume *s-r'*: $s\ r' = Some\ (\sigma',\tau',e')$
have $r' = r$ **using** s *s-r'* *prog-expr-e* **by** (*meson domI domIff fun-upd-other*)
hence $LID_L\ (\sigma',\tau',e') = LID_L\ (\varepsilon,\ \varepsilon,\ e)$ **using** s *s-r'* **by** *auto*
also have ... $= \{\}$ **using** *prog-expr-e* **by** *auto*
also have ... $= dom\ \sigma' \cup dom\ \tau'$ **using** ‹$r' = r$› s *s-r'* **by** *auto*
finally show $\mathcal{S}\ (\sigma',\ \tau',\ e')$ **by** (*simp add*: *domains-subsume-def*)
qed
show $\mathcal{A}_G\ s$
proof (*rule subsumes-accessible-globallyI*)
fix $r_1\ \sigma_1\ \tau_1\ e_1\ r_2\ \sigma_2\ \tau_2\ e_2$
assume *s-r1*: $s\ r_1 = Some\ (\sigma_1,\ \tau_1,\ e_1)$ **and** *s-r2*: $s\ r_2 = Some\ (\sigma_2,\ \tau_2,\ e_2)$
have $r_2 = r$ **using** s *s-r2* *prog-expr-e* **by** (*meson domI domIff fun-upd-other*)
hence $\sigma_2 = \varepsilon$ **using** s *s-r2* **by** *auto*
hence $LID_S\ \sigma_2 = \{\}$ **by** *auto*
thus $\mathcal{A}\ r_1\ r_2\ s$ **using** *s-r2* **by** *auto*
qed
qed
qed (*use step-preserves-*$\mathcal{S}_G$*-and-*$\mathcal{A}_G$ **in** *auto*)

corollary *reachable-imp-*$\mathcal{S}_G$: *reachable* $s \Longrightarrow \mathcal{S}_G\ s$
proof −
assume *reach*: *reachable* s
have $\mathcal{S}_G\ s \wedge \mathcal{A}_G\ s$ **by** (*rule inductive-invariant-is-execution-invariant*[*OF reach nice-ind-inv-is-inductive-invariant*])
thus *?thesis* **by** *auto*
qed

lemma *transition-relations-equivalent*: *reachable* $s \Longrightarrow$ *revision-step* $r\ s\ s' =$ *revision-step-relaxed* $r\ s\ s'$
proof −
assume *reach*: *reachable* s
have *doms-sub-local*: $\mathcal{S}_G\ s$ **by** (*rule reachable-imp-*$\mathcal{S}_G$[*OF reach*])
show *revision-step* $r\ s\ s' =$ *revision-step-relaxed* $r\ s\ s'$
proof (*rule iffI*)
assume *step*: *revision-step* $r\ s\ s'$
show *revision-step-relaxed* $r\ s\ s'$
proof (*use step* **in** ‹*induct rule*: *revision-stepE*›)
case (*new* $\sigma\ \tau\ \mathcal{E}\ v\ l$)
have *revision-step-relaxed* $r\ s\ (s(r \mapsto (\sigma,\ \tau(l \mapsto v),\ \mathcal{E}\ [VE\ (Loc\ l)])))$

```
      proof (rule revision-step-relaxed.new)
        show l ∉ ⋃ { doms ls | ls. ls ∈ ran s}
        proof
          assume l ∈ ⋃ { doms ls | ls. ls ∈ ran s}
           then obtain ls where in-ran: ls ∈ ran s and in-doms: l ∈ doms ls by
blast
          from in-doms have l ∈ LID_L ls by (cases ls) auto
          have l ∈ LID_G s
          proof −
            have ls ∈ {ls. ∃ r. s r = Some ls} by (metis (full-types) in-ran ran-def)
             then show ?thesis using ‹l ∈ LID_L ls› by blast
          qed
          thus False using new by auto
        qed
      qed (simp add: new.hyps(2))
      thus ?thesis using new.hyps(1) by blast
    qed (use revision-step-relaxed.intros in simp)+
  next
    assume step: revision-step-relaxed r s s'
    show revision-step r s s'
    proof (use step in ‹induct rule: revision-step-relaxedE›)
      case (new σ τ ℰ v l)
      have revision-step r s (s(r ↦ (σ, τ(l ↦ v), ℰ [VE (Loc l)])))
      proof (rule revision-step.new)
        show s r = Some (σ, τ, ℰ [Ref (VE v)]) by (simp add: new.hyps(2))
        show l ∉ LID_G s
        proof
          assume l ∈ LID_G s
          then obtain r' σ' τ' e' where s-r': s r' = Some (σ',τ',e') and l-in-local:
l ∈ LID_L (σ',τ',e') by auto
          hence l ∈ dom σ' ∪ dom τ'
         by (metis (no-types, lifting) domains-subsume-def domains-subsume-globally-def
doms.simps doms-sub-local rev-subsetD)
          thus False by (meson s-r' new.hyps(3) ranI)
        qed
      qed
      then show ?case using new.hyps(1) by blast
    next
      case (get σ τ ℰ l)
      have revision-step r s (s(r ↦ (σ, τ, ℰ [VE (the ((σ;;τ) l))])))
      proof
       show s r = Some (σ, τ, ℰ [Read (VE (Loc l))]) by (simp add: get.hyps(2))
       show l ∈ dom (σ;;τ)
       proof −
         have l ∈ LID_L (σ, τ, ℰ [Read (VE (Loc l))])  by simp
         hence l ∈ dom σ ∪ dom τ
           using domains-subsume-def domains-subsume-globally-def doms-sub-local
get.hyps(2) by fastforce
         thus l ∈ dom (σ;;τ) by (simp add: dom-combination-dom-union)
```

```
      qed
    qed
    then show ?case using get.hyps(1) by auto
   next
    case (set σ τ ℰ l v)
    have revision-step r s (s(r ↦ (σ, τ(l ↦ v), ℰ [VE (CV Unit)])))
    proof
      show s r = Some (σ, τ, ℰ [Assign (VE (Loc l)) (VE v)]) by (simp add:
set.hyps(2))
     show l ∈ dom (σ;;τ)
     proof −
      have l ∈ LID_L (σ, τ, ℰ [Assign (VE (Loc l)) (VE v)]) by simp
      hence l ∈ dom σ ∪ dom τ
       using domains-subsume-def domains-subsume-globally-def doms-sub-local
set.hyps(2) by fastforce
      thus l ∈ dom (σ;;τ) by (simp add: dom-combination-dom-union)
     qed
    qed
    then show ?case using set.hyps(1) by blast
   qed (simp add: revision-step.intros)+
  qed
qed
```

6.3.3 Finitude is invariant

```
lemma finite-occurrences-val-expr [simp]:
  fixes
    v :: ('r,'l,'v) val and
    e :: ('r,'l,'v) expr
  shows
  finite (RID_V v)
  finite (RID_E e)
  finite (LID_V v)
  finite (LID_E e)
proof −
  have (finite (RID_V v) ∧ finite (LID_V v)) ∧ finite (RID_E e) ∧ finite (LID_E e)
    by (induct rule: val-expr.induct) auto
  thus
    finite (RID_V v)
    finite (RID_E e)
    finite (LID_V v)
    finite (LID_E e)
    by auto
qed

lemma store-finite-upd [intro]:
  finite (RID_S τ) ⟹ finite (RID_S (τ(l := None)))
  finite (LID_S τ) ⟹ finite (LID_S (τ(l := None)))
  apply (meson ID-restricted-store-subset-store(1) finite-subset)
```

by (*simp add*: *ID-restricted-store-subset-store*(*2*) *rev-finite-subset*)

lemma *finite-state-imp-restriction-finite* [*intro*]:
finite (*RID*$_G$ *s*) ⟹ *finite* (*RID*$_G$ (*s*(*r* := *None*)))
finite (*LID*$_G$ *s*) ⟹ *finite* (*LID*$_G$ (*s*(*r* := *None*)))
proof −
assume *finite* (*RID*$_G$ *s*)
thus *finite* (*RID*$_G$ (*s*(*r* := *None*))) **by** (*meson infinite-super ID-restricted-global-subset-unrestricted*)
next
assume *fin*: *finite* (*LID*$_G$ *s*)
have *LID*$_G$ (*s*(*r* := *None*)) ⊆ *LID*$_G$ *s* **by** *auto*
thus *finite* (*LID*$_G$ (*s*(*r* := *None*))) **using** *fin finite-subset* **by** *auto*
qed

lemma *local-state-of-finite-restricted-global-state-is-finite* [*intro*]:
s r′ = *Some ls* ⟹ *finite* (*RID*$_G$ (*s*(*r* := *None*))) ⟹ *r* ≠ *r*′ ⟹ *finite* (*RID*$_L$ *ls*)
s r′ = *Some ls* ⟹ *finite* (*LID*$_G$ (*s*(*r* := *None*))) ⟹ *r* ≠ *r*′ ⟹ *finite* (*LID*$_L$ *ls*)
apply (*metis* (*no-types*, *lifting*) *ID-distr-global*(*1*) *finite-Un finite-insert fun-upd-triv fun-upd-twist*)
by (*metis ID-distr-global*(*2*) *finite-Un fun-upd-triv fun-upd-twist*)

lemma *empty-map-finite* [*simp*]:
finite (*RID*$_S$ *ε*)
finite (*LID*$_S$ *ε*)
finite (*RID*$_G$ *ε*)
finite (*LID*$_G$ *ε*)
by (*simp add*: *RID*$_S$*-def LID*$_S$*-def RID*$_G$*-def LID*$_G$*-def*)+

lemma *finite-combination* [*intro*]:
finite (*RID*$_S$ *σ*) ⟹ *finite* (*RID*$_S$ *τ*) ⟹ *finite* (*RID*$_S$ (*σ*;;*τ*))
finite (*LID*$_S$ *σ*) ⟹ *finite* (*LID*$_S$ *τ*) ⟹ *finite* (*LID*$_S$ (*σ*;;*τ*))
by (*meson finite-UnI rev-finite-subset ID-combination-subset-union*)+

lemma *RID*$_G$*-finite-invariant*:
assumes
step: *revision-step r s s*′ **and**
fin: *finite* (*RID*$_G$ *s*)
shows
finite (*RID*$_G$ *s*′)
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case (*join σ τ ℰ r*′ *σ*′ *τ*′ *v*)
hence *r* ≠ *r*′ **by** *auto*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *ID-distr-global*(*1*) *ID-distr-local*(*2*) *fin finite-Un finite-combination*(*1*) *finite-insert finite-occurrences-val-expr*(*2*) *finite-state-imp-restriction-finite*(*1*) *join local-state-of-finite-restricted-global-state-is-finite*(*1*))
qed (*use fin* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)

lemma *RID_L-finite-invariant*:
assumes
step: *revision-step r s s'* **and**
fin: *finite* (*LID_G s*)
shows
finite (*LID_G s'*)
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case (*join σ τ ℰ r' σ' τ' v*)
hence *r* ≠ *r'* **by** *auto*
then show *?thesis*
using *join assms*
by (*metis* (*mono-tags*, *lifting*) *ID-distr-global*(*2*) *ID-distr-local*(*1*) *fin finite-Un finite-combination*(*2*) *finite-occurrences-val-expr*(*4*) *finite-state-imp-restriction-finite*(*2*) *join local-state-of-finite-restricted-global-state-is-finite*(*2*))
qed (*use fin* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)

lemma *reachable-imp-identifiers-finite*:
assumes *reach*: *reachable s*
shows
finite (*RID_G s*)
finite (*LID_G s*)
proof −
from *reach* **obtain** *e r* **where** *exec*: *execution e* (*ε*(*r* ↦ (*ε*,*ε*,*e*))) *s* **using** *reachable-def execution-def* **by** *auto*
hence *prog-exp*: *program-expr e* **by** (*meson execution-def*)
obtain *n* **where** *n-reachable*: (*ε*(*r* ↦ (*ε*,*ε*,*e*)), *s*) ∈ [⇝]^^*n* **using** *exec* **by** (*meson execution-def rtrancl-imp-relpow*)
hence *finite* (*RID_G s*) ∧ *finite* (*LID_G s*)
proof (*induct n arbitrary*: *s*)
case *0*
hence *s*: *s* = *ε*(*r* ↦ (*ε*, *ε*, *e*)) **by** *auto*
hence *rid-dom*: *dom s* = {*r*} **by** *auto*
hence *rid-ran*: ⋃ (*RID_L* ‘ *ran s*) = {} **using** *s* **by** (*auto simp add*: *prog-exp*)
have *rids*: *RID_G s* = {*r*} **by** (*unfold RID_G-def*, *use rid-dom rid-ran* **in** *auto*)
have *lid-ran*: ⋃ (*LID_L* ‘ *ran s*) = {} **using** *s* **by** (*auto simp add*: *prog-exp*)
hence *lids*: *LID_G s* = {} **by** (*unfold LID_G-def*, *simp*)
thus *?case* **using** *rids lids* **by** *simp*
next
case (*Suc n*)
then obtain *s'* **where**
n-steps: (*ε*(*r* ↦ (*ε*, *ε*, *e*)), *s'*) ∈ [⇝]^^*n* **and**
step: *s'* ⇝ *s*
by (*meson relpow-Suc-E*)
have *fin-rid*: *finite* (*RID_G s'*) **using** *Suc.hyps n-steps* **by** *blast*
have *fin-lid*: *finite* (*LID_G s'*) **using** *Suc.hyps n-steps* **by** *blast*
thus *?case* **by** (*meson RID_G-finite-invariant RID_L-finite-invariant fin-rid local.step valid-stepE*)
qed
thus *finite* (*RID_G s*) *finite* (*LID_G s*) **by** *auto*

qed

lemma *reachable-imp-identifiers-available*:
assumes
reachable (s :: ($'r$,$'l$,$'v$) *global-state*)
shows
infinite (*UNIV* :: $'r$ *set*) $\Longrightarrow \exists\, r.\ r \notin RID_G\ s$
infinite (*UNIV* :: $'l$ *set*) $\Longrightarrow \exists\, l.\ l \notin LID_G\ s$
by (*simp add*: *assms ex-new-if-finite reachable-imp-identifiers-finite*)+

6.3.4 Reachability is invariant

lemma *initial-state-reachable*:
assumes *program-expr e*
shows *reachable* ($\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))$)
proof −
have *initializes* ($\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))$) *e* **using** *assms* **by** *auto*
hence *execution e* ($\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))$) ($\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))$) **by** (*simp add*: *execution-def*)
thus *?thesis* **using** *reachable-def* **by** *blast*
qed

lemma *reachability-closed-under-execution-step*:
assumes
reach: *reachable s* **and**
step: *revision-step r s s′*
shows *reachable s′*
proof −
obtain *init e* **where** *exec*: *execution e init s* **using** *reach reachable-def* **by** *blast*
hence *init-s*:*init* $\rightsquigarrow^*$ *s* **by** (*simp add*: *execution-def*)
have *s-s′*: *s* $\rightsquigarrow$ *s′* **using** *step* **by** *blast*
have *init* $\rightsquigarrow^*$ *s′* **using** *init-s s-s′* **by** *auto*
hence *execution e init s′* **using** *exec* **by** (*simp add*: *execution-def*)
thus *?thesis* **using** *reachable-def* **by** *auto*
qed

lemma *reachability-closed-under-execution*: *reachable s* $\Longrightarrow$ *s* $\rightsquigarrow^*$ *s′* $\Longrightarrow$ *reachable s′*
proof −
assume *reach*: *reachable s* **and** *s* $\rightsquigarrow^*$ *s′*
then obtain *n* **where** (*s*, *s′*) $\in [\rightsquigarrow]^{\frown}n$ **using** *rtrancl-imp-relpow* **by** *blast*
thus *reachable s′*
proof (*induct n arbitrary*: *s′*)
case *0*
thus *?case* **using** *reach* **by** *auto*
next
case (*Suc n*)
obtain *s″* **where** (*s*,*s″*) $\in [\rightsquigarrow]^{\frown}n$ *s″* $\rightsquigarrow$ *s′* **using** *Suc.prems* **by** *auto*
have *reachable s″* **by** (*simp add*: *Suc.hyps* ‹(*s*, *s″*) $\in [\rightsquigarrow]^{\frown}n$›)
then show *?case* **using** ‹*s″* $\rightsquigarrow$ *s′*› *reachability-closed-under-execution-step* **by**

blast
qed
qed

end

end

7 Determinacy

This section proves that the concurrent revisions model is determinate modulo renaming-equivalence.

theory *Determinacy*
imports *Executions*
begin

context *substitution*
begin

7.1 Rule determinism

lemma *app-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Apply* (*VE* (*Lambda x e*)) (*VE v*)])
shows (*revision-step r s s′*) = (*s′* = (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*subst* (*VE v*) *x e*])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*simp add*: *s-r revision-step.app*)

lemma *ifTrue-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Ite* (*VE* (*CV T*)) *e1 e2*])
shows (*revision-step r s s′*) = (*s′* = (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*e1*])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*simp add*: *s-r revision-step.ifTrue*)

lemma *ifFalse-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Ite* (*VE* (*CV F*)) *e1 e2*])
shows (*revision-step r s s′*) = (*s′* = (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*e2*])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*simp add*: *s-r revision-step.ifFalse*)

lemma *new-pseudodeterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Ref* (*VE v*)])
shows (*revision-step r s s′*) = (∃ *l*. *l* ∉ *LID*$_G$ *s* ∧ *s′* = (*s*(*r* ↦ (*σ*, *τ*(*l* ↦ *v*), *ℰ* [*VE* (*Loc l*)])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*auto simp add*: *s-r revision-step.new*)

lemma *get-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Read* (*VE* (*Loc l*))])
shows (*revision-step r s s′*) = (*l* ∈ *dom* (*σ*;;*τ*) ∧ *s′* = (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*VE* (*the* ((*σ*;;*τ*) *l*))])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*use revision-step.get* **in** ‹*auto simp add*: *s-r*›)

lemma *set-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Assign* (*VE* (*Loc l*)) (*VE v*)])
shows (*revision-step r s s′*) = (*l* ∈ *dom* (*σ*;;*τ*) ∧ *s′* = (*s*(*r* ↦ (*σ*, *τ*(*l* ↦ *v*), *ℰ* [*VE* (*CV Unit*)])))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *?l*
thus *?r* **by** (*cases rule*: *revision-stepE*) (*use s-r* **in** *auto*)
qed (*auto simp add*: *s-r revision-step.set*)

lemma *fork-pseudodeterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Rfork e*])
shows (*revision-step r s s′*) = (∃ *r′*. *r′* ∉ *RID*$_G$ (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*Rfork e*]))) ∧ *s′* = (*s*(*r* ↦ (*σ*, *τ*, *ℰ* [*VE* (*Rid r′*)]), *r′* ↦ (*σ*;;*τ*, *ε*, *e*)))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *step*: *?l*
show *?r*
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case (*fork σ τ ℰ e r′*)
show *?thesis* **by** (*rule exI*[**where** *x*=*r′*]) (*use fork s-r* **in** *auto*)
qed (*auto simp add*: *s-r*)
qed (*auto simp add*: *s-r revision-step.fork map-upd-triv*)

lemma *rjoin-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Rjoin* (*VE* (*Rid r′*))]) **and**
s-r′: *s r′* = *Some* (*σ′*, *τ′*, *VE v*)

shows (*revision-step r s s'*) = (*s'* = (*s*(*r* := *Some* (*σ*, *τ*;;*τ'*, *ℰ* [*VE* (*CV Unit*)]), *r'* := *None*))) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *step*: *?l*
show *?r* **by** (*cases rule*: *revision-stepE*[*OF step*]) (*use s-r s-r'* **in** *auto*)
qed (*meson s-r s-r' revision-step.join*)

lemma *rjoin$_\varepsilon$-deterministic* [*simp*]:
assumes
s-r: *s r* = *Some* (*σ*, *τ*, *ℰ* [*Rjoin* (*VE* (*Rid r'*))]) **and**
s-r': *s r'* = *None*
shows (*revision-step r s s'*) = (*s'* = *ε*) (**is** *?l* = *?r*)
proof (*rule iffI*)
assume *step*: *?l*
show *?r* **by** (*cases rule*: *revision-stepE*[*OF step*]) (*use s-r s-r'* **in** *auto*)
qed (*simp add*: *revision-step.join$_\varepsilon$ s-r s-r'*)

7.2 Strong local confluence

7.2.1 Local determinism

lemma *local-determinism*:
assumes
left: *revision-step r* s_1 s_2 **and**
right: *revision-step r* s_1 $s_2{}'$
shows $s_2 \approx s_2{}'$
proof (*use left* **in** ‹*induct rule*:*revision-stepE*›)
case (*new σ τ ℰ v l*)
from *new*(*2*) *right* **obtain** *l'* **where**
side: *l'* ∉ *LID$_G$* s_1 **and**
$s_2{}'$: $s_2{}'$ = s_1(*r* ↦ (*σ*, *τ*(*l'* ↦ *v*), *ℰ*[*VE* (*Loc l'*)]))
by *auto*
let *?β* = *id*(*l* := *l'*, *l'* := *l*)
have *bij-β*: *bij ?β* **by** (*rule swap-bij*)
have *renaming*: *ℛ$_G$ id ?β* s_2 = $s_2{}'$
by (*use new side* $s_2{}'$ *bij-β* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
show *?case* **by** (*rule eq-statesI*[*OF renaming bij-id bij-β*])
next
case (*fork σ τ ℰ e r'*)
from *fork*(*2*) *right* **obtain** *r''* **where**
side: *r''* ∉ *RID$_G$* s_1 **and**
$s_2{}'$: $s_2{}'$ = s_1(*r* ↦ (*σ*, *τ*, *ℰ* [*VE* (*Rid r''*)]), *r''* ↦ (*σ*;;*τ*, *ε*, *e*))
by (*auto simp add*: *ID-distr-global-conditional*)
let *?α* = *id*(*r'* := *r''*, *r''* := *r'*)
have *bij-α*: *bij ?α* **by** (*rule swap-bij*)
have *renaming*: *ℛ$_G$?α id* s_2 = $s_2{}'$
by (*use fork side* $s_2{}'$ *bij-α* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
show *?case* **by** (*rule eq-statesI*[*OF renaming bij-α bij-id*])
qed ((*rule eq-statesI*[*of id id*], *use assms* **in** *auto*)[*1*])+

7.2.2 General principles

lemma *SLC-sym*:
$\exists s_3'\ s_3.\ s_3' \approx s_3 \wedge$ (*revision-step* $r'\ s_2\ s_3' \vee s_2 = s_3'$) $\wedge$ (*revision-step* $r\ s_2'\ s_3 \vee s_2' = s_3$) $\Longrightarrow$
$\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge$ (*revision-step* $r\ s_2'\ s_3 \vee s_2' = s_3$) $\wedge$ (*revision-step* $r'\ s_2\ s_3' \vee s_2 = s_3'$)
by (*metis* $\alpha\beta$-*sym*)

lemma *SLC-commute*:
$[\![s_3 = s_3'$; *revision-step* $r'\ s_2\ s_3$; *revision-step* $r\ s_2'\ s_3']\!] \Longrightarrow$
$s_3 \approx s_3' \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ s_2'\ s_3' \vee s_2' = s_3'$)
using $\alpha\beta$-*refl* **by** *auto*

7.2.3 Case join-epsilon

lemma *SLC-join*$_\varepsilon$:
assumes
s_1-r: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}[Rjoin\ (VE\ (Rid\ r''))])$ **and**
s_2: $s_2 = \varepsilon$ **and**
side: $s_1\ r'' = None$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ s_2'\ s_3' \vee s_2' = s_3'$)
proof −
have *right-collapsed-case*: $s_2' = \varepsilon \Longrightarrow$ *?thesis*
by (*rule exI*[**where** x=ε], *rule exI*[**where** x=ε], *use* s_2 **in** *auto*)
have *left-step-still-available-case*: $s_2' \neq \varepsilon \Longrightarrow s_2'\ r = s_1\ r \Longrightarrow s_2'\ r'' = None \Longrightarrow$ *?thesis*
by (*rule exI*[**where** x=ε], *rule exI*[**where** x=ε]) (*use assms* **in** *auto*)
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case (*join - - - right-joinee*)
have *r-unchanged-left*: $s_2'\ r = s_1\ r$ **using** *join assms* **by** *auto*
have r'-*unchanged-right*: $s_2'\ r'' = None$ **using** *join assms* **by** *auto*
have *right-joinee* $\neq r'$ **using** *join*(*2*−*3*) **by** *auto*
hence s_2'-*nonempty*: $s_2' \neq \varepsilon$ **using** *assms join* **by** (*auto simp add*: *fun-upd-twist*)
show *?thesis* **by** (*rule left-step-still-available-case*[*OF* s_2'-*nonempty r-unchanged-left* r'-*unchanged-right*])
next
case *join*$_\varepsilon$
show *?thesis* **by** (*rule right-collapsed-case*, *use join*$_\varepsilon$(*2*−*3*) *right* **in** *auto*)
qed ((*rule left-step-still-available-case*, *use side neq* s_1-r *right* **in** *auto*)[*1*])+
qed

7.2.4 Case join

lemma *join-and-local-commute*:
assumes
$s_2 = s_1(r := Some\ (\sigma,\ \tau;;\tau',\ \mathcal{E}[VE\ (CV\ Unit)]),\ r'' := None)$
$s_2{}' = s_1(r' \mapsto ls)$
$r \neq r'$
$r' \neq r''$
revision-step $r'\ s_2$ ($s_1(r := Some\ (\sigma,\ \tau;;\tau',\ \mathcal{E}[VE\ (CV\ Unit)]),\ r'' := None,\ r' := Some\ ls)$)
$s_2{}'\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Rjoin\ (VE\ (Rid\ r''))])$
$s_2{}'\ r'' = Some\ (\sigma',\ \tau',\ VE\ v)$
shows
$\exists\, s_3\ s_3{}'.\ s_3 \approx s_3{}' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2{}'\ s_3{}' \vee s_2{}' = s_3{}')$
apply (*rule exI*[**where** $x{=}s_1(r := Some\ (\sigma,\ \tau;;\tau',\ \mathcal{E}[VE\ (CV\ Unit)]),\ r'' := None,\ r' := Some\ ls)$])
apply (*rule exI*[**where** $x{=}s_1(r' := Some\ ls,\ r := Some\ (\sigma,\ \tau;;\tau',\ \mathcal{E}[VE\ (CV\ Unit)]),\ r'' := None)$])
by (*rule SLC-commute*, *use assms* **in** *auto*)

lemma *SLC-join*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rjoin\ (VE\ (Rid\ r''))])$ **and**
s_2: $s_2 = (s_1(r := Some\ (\sigma,\ (\tau;;\tau'),\ \mathcal{E}[VE\ (CV\ Unit)]),\ r'' := None))$ **and**
side: $s_1\ r'' = Some\ (\sigma',\ \tau',\ VE\ v)$ **and**
right: *revision-step* $r'\ s_1\ s_2{}'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3{}'.\ s_3 \approx s_3{}' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2{}'\ s_3{}' \vee s_2{}' = s_3{}')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1-*r* s_2 *side* **by** *auto*
have r'-*not-joined*: $r' \neq r''$ **using** *right side* **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case (*new* - - - - *l*)
have *l-fresh-left*: $l \notin LID_G\ s_2$ **by** (*rule only-new-introduces-lids'*[*OF left-step*]) (*use new right* s_1-*r* **in** *auto*)
show *?thesis* **by** (*rule join-and-local-commute*, *use assms* r'-*not-joined new l-fresh-left* **in** *auto*)
next
case (*fork* - - - - r''')
have r'-*unchanged-left*: $s_2\ r' = s_1\ r'$ **using** *fork assms* **by** *auto*
have r'''-*fresh-left*: $r''' \notin RID_G\ s_2$ **using** *left-step fork*(*3*) *only-fork-introduces-rids'* s_1-*r* **by** *auto*
have *r-unchanged-right*: $s_2{}'\ r = s_1\ r$ **using** *fork assms* **by** *auto*
have r''-*unchanged-right*: $s_2{}'\ r'' = s_1\ r''$ **using** *fork assms* **by** *auto*
let $?s_3 = s_2(r' := s_2{}'\ r',\ r''' := s_2{}'\ r''')$
let $?s_3{}' = s_2{}'(r := s_2\ r,\ r'' := None)$

show *?thesis*
proof (*rule exI*[**where** *x*=*?s*$_3$],*rule exI*[**where** *x*=*?s*$_3$′], *rule SLC-commute*)
show *?s*$_3$ = *?s*$_3$′ **using** *fork*(*1*) *fork*(*3*) *neq r*′*-not-joined s*$_1$*-r s*$_2$ **by** (*auto simp add*: *ID-distr-global-conditional*)
show *revision-step r*′ *s*$_2$ *?s*$_3$ **using** *fork*(*1*−*2*) *r*′*-unchanged-left r*′′′*-fresh-left* **by** (*auto simp add*: *ID-distr-global-conditional*)
show *revision-step r s*$_2$′ *?s*$_3$′ **using** *r*′′*-unchanged-right r-unchanged-right s*$_1$*-r s*$_2$ *side* **by** *auto*
qed
next
case (*join - - - r*′′′)
have *r*′*-unchanged-left*: *s*$_2$ *r*′ = *s*$_1$ *r*′ **using** *join*(*2*) *neq r*′*-not-joined s*$_2$ **by** *auto*
have *r-unchanged-right*: *s*$_2$′ *r* = *s*$_1$ *r* **using** *join*(*1*,*3*) *neq s*$_1$*-r* **by** *auto*
show *?thesis*
proof (*cases r*′′ = *r*′′′)
case *True*
have *r*′′′*-none-left*: *s*$_2$ *r*′′′ = *None* **by** (*simp add*: *True s*$_2$)
have *r*′′*-none-right*: *s*$_2$′ *r*′′ = *None* **by** (*simp add*: *True join*(*1*))
show *?thesis*
proof (*rule exI*[**where** *x*=*ε*], *rule exI*[**where** *x*=*ε*], *rule SLC-commute*)
show *ε* = *ε* **by** (*rule refl*)
show *revision-step r*′ *s*$_2$ *ε* **using** *r*′*-unchanged-left r*′′′*-none-left join*(*2*) **by** *auto*
show *revision-step r s*$_2$′ *ε* **using** *r-unchanged-right r*′′*-none-right s*$_1$*-r* **by** *auto*
qed
next
case *False*
have *r*′′′*-unchanged-left*: *s*$_2$ *r*′′′ = *s*$_1$ *r*′′′ **using** *False join*(*1*,*3*) *s*$_2$ *r-unchanged-right* **by** *auto*
have *r*′′*-unchanged-right*′: *s*$_2$′ *r*′′ = *s*$_1$ *r*′′ **using** *False join*(*1*) *r*′*-not-joined side* **by** *auto*
let *?s*$_3$ = *s*$_2$(*r*′ := *s*$_2$′ *r*′, *r*′′′ := *None*)
let *?s*$_3$′ = *s*$_2$′(*r* := *s*$_2$ *r*, *r*′′ := *None*)
show *?thesis*
proof (*rule exI*[**where** *x*=*?s*$_3$], *rule exI*[**where** *x*=*?s*$_3$′], *rule SLC-commute*)
show *?s*$_3$ = *?s*$_3$′ **using** *join*(*1*) *neq r*′*-not-joined r-unchanged-right s*$_1$*-r s*$_2$ *s*$_1$*-r* **by** *fastforce*
show *revision-step r*′ *s*$_2$ *?s*$_3$ **by** (*simp add*: *join r*′′′*-unchanged-left r*′*-unchanged-left*)
show *revision-step r s*$_2$′ *?s*$_3$′ **using** *r*′′*-unchanged-right*′ *r-unchanged-right s*$_1$*-r side s*$_2$ **by** *auto*
qed
qed
next
case *join*$_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*$_\varepsilon$, *use left-step neq right join*$_\varepsilon$ **in** *auto*)
qed ((*rule join-and-local-commute*, *use assms r*′*-not-joined* **in** *auto*)[*1*])+

qed

7.2.5 Case local

lemma *local-steps-commute*:
assumes
$s_2 = s_1(r \mapsto x)$
$s_2' = s_1(r' \mapsto y)$
revision-step r' $(s_1(r \mapsto x))$ $(s_1(r \mapsto x,\ r' \mapsto y))$
revision-step r $(s_1(r' \mapsto y))$ $(s_1(r' \mapsto y,\ r \mapsto x))$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ s_2'\ s_3' \vee s_2' = s_3'$)
by (*metis* (*no-types*, *lifting*) *assms fun-upd-twist fun-upd-upd local-determinism*)

lemma *local-and-fork-commute*:
assumes
$s_2 = s_1(r \mapsto x)$
$s_2' = s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e))$
$s_2\ r' = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rfork\ e])$
$r'' \notin RID_G\ s_2$
revision-step $r\ s_2'$ $(s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e),\ r \mapsto x))$
$r \neq r'$
$r \neq r''$
shows
$\exists\, s_3\ s_3'.\ (s_3 \approx s_3') \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ s_2'\ s_3' \vee s_2' = s_3'$)
proof(*rule exI*[**where** x=$s_1(r \mapsto x,\ r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e))$],
rule exI[**where** x=$s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e),\ r \mapsto x)$],
rule SLC-commute)
show $s_1(r \mapsto x,\ r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e)) =$
$s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e),\ r \mapsto x)$
using *assms revision-step.fork* **by** *auto*
show *revision-step* $r'\ s_2$ $(s_1(r \mapsto x,\ r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e)))$
using *assms(1) assms(3) assms(4) revision-step.fork* **by** *blast*
show *revision-step* $r\ s_2'$ $(s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e),\ r \mapsto x))$
using *assms(5)* **by** *blast*
qed

lemma *SLC-app*:
assumes
s_1-r: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Apply\ (VE\ (Lambda\ x\ e))\ (VE\ v)])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[subst\ (VE\ v)\ x\ e]))$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$

shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1-$r\ s_2$ **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case *new*: (*new* - - - - *l*)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: s_1-r *new*(*3*))+
show *?thesis* **by** (*rule local-steps-commute*) (*use new l-fresh-left assms* **in** *auto*)
next
case (*fork* - - - - r'')
have r''*-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids'*[*OF left-step*]) (*simp add*: s_1-r *fork*(*3*))+
show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork*(*1*)]) (*use fork neq* s_2 r''*-fresh-left* s_1-r **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step right neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step right neq* **in** *auto*)
qed ((*rule local-steps-commute*[*OF* s_2], *use assms* **in** *auto*)[*1*])+
qed

lemma *SLC-ifTrue*:
assumes
s_1-r: $s_1\ r = \textit{Some}\ (\sigma, \tau, \mathcal{E}[\textit{Ite}\ (\textit{VE}\ (\textit{CV}\ T))\ e1\ e2])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[e1]))$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1-$r\ s_2$ **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case (*new* - - - - *l*)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: s_1-r *new*(*3*))+
show *?thesis* **by** (*rule local-steps-commute*) (*use new l-fresh-left assms* **in** *auto*)
next
case (*fork* - - - - r'')
have r''*-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids'*[*OF left-step*]) (*simp add*: s_1-r *fork*(*3*))+

show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork*(*1*)]) (*use fork neq* s_2 r''*-fresh-left* s_1*-r* **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step right neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step right neq* **in** *auto*)
qed ((*rule local-steps-commute*[*OF* s_2], *use assms* **in** *auto*)[*1*])+
qed

lemma *SLC-ifFalse*:
assumes
s_1*-r*: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}[Ite\ (VE\ (CV\ F))\ e1\ e2])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[e2]))$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2'\ s_3' \vee s_2' = s_3')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1*-r* s_2 **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
next
case (*new - - - - l*)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: s_1*-r new*(*3*))+
show *?thesis* **by** (*rule local-steps-commute*) (*use new l-fresh-left assms* **in** *auto*)
next
case (*fork - - - - r''*)
have r''*-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids'*[*OF left-step*]) (*simp add*: s_1*-r fork*(*3*))+
show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork*(*1*)]) (*use fork neq* s_2 r''*-fresh-left* s_1*-r* **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step right neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step right neq* **in** *auto*)
qed ((*rule local-steps-commute*[*OF* s_2], *use assms* **in** *auto*)[*1*])+
qed

lemma *SLC-set*:

assumes
s_1-r: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Assign\ (VE\ (Loc\ l))\ (VE\ v)])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma,\ \tau(l \mapsto v),\ \mathcal{E}[VE\ (CV\ Unit)]))$ **and**
side: $l \in dom\ (\sigma;;\tau)$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2'\ s_3' \vee s_2' = s_3')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1-$r\ s_2$ *side* **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case (*new* - - - - l)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: s_1-r *new*(*3*))+
show *?thesis* **by** (*rule local-steps-commute*) (*use new l-fresh-left assms* **in** *auto*)
next
case (*fork* - - - - r'')
have r''*-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids'*[*OF left-step*]) (*simp add*: s_1-r *fork*(*3*))+
show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork*(*1*)]) (*use fork neq* s_2 r''*-fresh-left* s_1-r *side* **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step neq* **in** *auto*)
qed ((*rule local-steps-commute*[*OF* s_2], *use assms* **in** *auto*)[*1*])+
qed

lemma *SLC-get*:
assumes
s_1-r: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Read\ (VE\ (Loc\ l))])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[VE\ (the\ ((\sigma;;\tau)\ l))]))$ **and**
side: $l \in dom\ (\sigma;;\tau)$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2'\ s_3' \vee s_2' = s_3')$
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1-$r\ s_2$ *side* **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case (*new* - - - - l)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: s_1-r *new*(*3*))+

show *?thesis* **by** (*rule local-steps-commute*) (*use new l-fresh-left assms* **in** *auto*)
next
case (*fork - - - - r''*)
have *r''-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids'*[*OF left-step*]) (*simp add*: s_1*-r fork*(*3*))+
show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork*(*1*)]) (*use fork neq* s_2 *r''-fresh-left* s_1*-r side* **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step neq* **in** *auto*)
qed ((*rule local-steps-commute*[*OF* s_2], *use assms* **in** *auto*)[*1*])+
qed

7.2.6 Case new

lemma *SLC-new*:
assumes
s_1*-r*: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}[Ref\ (VE\ v)])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}\ [VE\ (Loc\ l)]))$ **and**
side: $l \notin LID_G\ s_1$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$ **and**
reach: *reachable* (s_1 :: ('*r*,'*l*,'*v*) *global-state*) **and**
lid-inf: *infinite* (*UNIV* :: '*l set*)
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ s_2'\ s_3' \vee s_2' = s_3'$)
proof −
have *left-step*: *revision-step* $r\ s_1\ s_2$ **using** s_1*-r* s_2 *side* **by** *auto*
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case *app*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-app*) (*use app assms*(*1−5*) **in** *auto*)
next
case *ifTrue*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-ifTrue*) (*use ifTrue assms*(*1−5*) **in** *auto*)
next
case *ifFalse*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-ifFalse*) (*use ifFalse assms*(*1−5*) **in** *auto*)
next
case (*new* $\sigma'\ \tau'\ \mathcal{E}'\ v'\ l'$)
have *r'-unchanged-left*: $s_2\ r' = s_1\ r'$ **using** *new*(*2*) *neq* s_2 **by** *auto*
have *r-unchanged-right*: $s_2'\ r = s_1\ r$ **by** (*simp add*: *new*(*1*) *neq* s_1*-r*)
show *?thesis*

proof (*cases* $l = l'$)
case *True*
obtain l'' **where** l''*-fresh-left*: $l'' \notin LID_G\ s_2$
by (*meson ex-new-if-finite left-step lid-inf reach* RID_L*-finite-invariant reachable-imp-identifiers-finite*(*2*))
hence *l-l''*: $l \neq l''$ **by** (*auto simp add*: s_2)
have l''*-fresh-*s_1: $l'' \notin LID_G\ s_1$ **using** *assms True new* l''*-fresh-left* **by** (*auto simp add*: *ID-distr-global-conditional*)
hence l''*-fresh-right'*: $l'' \notin LID_G\ s_2'$ **using** *True l-l'' new*(*1−2*) **by** *auto*
let *?β* = *id*($l := l''$, $l'' := l$)
have *bij-β*: *bij ?β* **by** (*simp add*: *swap-bij*)
let *?*s_3 = $s_2(r' \mapsto (\sigma', \tau'(l'' \mapsto v'), \mathcal{E}'\ [VE\ (Loc\ l'')]))$
have *left-convergence*: *revision-step* $r'\ s_2$ *?*s_3
using l''*-fresh-left new*(*2*) r'*-unchanged-left* **by** *auto*
let *?*s_3' = $s_2'(r \mapsto (\sigma, \tau(l'' \mapsto v), \mathcal{E}\ [VE\ (Loc\ l'')]))$
have *right-convergence*: *revision-step* $r\ s_2'$ *?*s_3'
using l''*-fresh-right' new*(*1*) *neq* s_1*-r* **by** *auto*
have *equiv*: *?*$s_3 \approx$ *?*s_3'
proof (*rule eq-statesI*[*of id ?β*])
show $\mathcal{R}_G$ *id ?β ?*s_3 = *?*s_3'
proof −
have s_1: $\mathcal{R}_G$ *id ?β* $s_1 = s_1$ **using** l''*-fresh-*s_1 *side* **by** *auto*
have σ: $\mathcal{R}_S$ *id ?β* $\sigma = \sigma$ **using** l''*-fresh-*s_1 s_1*-r side* **by** *auto*
have $\mathcal{E}$: $\mathcal{R}_C$ *id ?β* $\mathcal{E} = \mathcal{E}$
proof
show $l \notin LID_C\ \mathcal{E}$ **using** s_1*-r side* **by** *auto*
show $l'' \notin LID_C\ \mathcal{E}$ **using** l''*-fresh-left* s_2 **by** *auto*
qed
have *τlv*: $\mathcal{R}_S$ *id* (*id*($l := l''$, $l'' := l$)) $(\tau(l \mapsto v)) = (\tau(l'' \mapsto v))$
proof −
have τ: $\mathcal{R}_S$ *id ?β* $\tau = \tau$ **using** l''*-fresh-*s_1 s_1*-r side* **by** *auto*
have v: $\mathcal{R}_V$ *id ?β* $v = v$
proof
show $l \notin LID_V\ v$ **using** s_1*-r side* **by** *auto*
show $l'' \notin LID_V\ v$ **using** l''*-fresh-*s_1 s_1*-r* **by** *auto*
qed
show *?thesis* **by** (*simp add*: τ v *bij-β*)
qed
have σ': $\mathcal{R}_S$ *id ?β* $\sigma' = \sigma'$ **using** l''*-fresh-*s_1 *new*(*2−3*) **by** (*auto simp add*: *True ID-distr-global-conditional*)
have $\mathcal{E}'$: $\mathcal{R}_C$ *id ?β* $\mathcal{E}' = \mathcal{E}'$ **using** l''*-fresh-*s_1 *new*(*2−3*) **by** (*auto simp add*: *True ID-distr-global-conditional*)
have *τl''v*: $\mathcal{R}_S$ *id* (*id*($l := l''$, $l'' := l$)) $(\tau'(l'' \mapsto v')) = (\tau'(l \mapsto v'))$
proof −
have τ': $\mathcal{R}_S$ *id ?β* $\tau' = \tau'$ **using** *new*(*2−3*) l''*-fresh-*s_1 **by** (*auto simp add*: *True ID-distr-global-conditional*)
have v': $\mathcal{R}_V$ *id ?β* $v' = v'$ **using** *new*(*2−3*) l''*-fresh-*s_1 **by** (*auto simp add*: *True ID-distr-global-conditional*)
show *?thesis* **by** (*simp add*: τ' v' *bij-β*)

qed
show *?thesis* **using** *True neq* s_1 σ $\mathcal{E}$ τlv σ' $\mathcal{E}'$ $\tau l'' v$ s_2 *l-l'' new(1)* **by** *auto*
qed
qed (*simp add*: *bij-β*)+
show *?thesis* **using** *left-convergence right-convergence equiv* **by** *blast*
next
case *False*
have *l-fresh-left*: $l \notin LID_G\ s_2'$
by (*rule revision-stepE*[*OF left-step*]) (*use False side new(1−2)* **in** ‹*auto simp add*: s_1*-r*›)
have *l′-fresh-right*: $l' \notin LID_G\ s_2$
by (*rule revision-stepE*[*OF right*]) (*use False new(3) assms(1−3)* **in** ‹*auto simp add*: *new(2)*›)
show *?thesis* **by** (*rule local-steps-commute*[*OF* s_2 *new(1)*]) (*use new assms l-fresh-left l′-fresh-right* s_2 **in** *auto*)
qed
next
case *get*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-get*) (*use get assms(1−5)* **in** *auto*)
next
case *set*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-set*) (*use set assms(1−5)* **in** *auto*)
next
case (*fork - - - - r′′*)
have *r′′-fresh-left*: $r'' \notin RID_G\ s_2$
by (*rule only-fork-introduces-rids′*[*OF left-step*]) (*simp add*: s_1*-r fork(3)*)+
have *l-fresh-right*: $l \notin LID_G\ s_2'$
by (*rule only-new-introduces-lids′*[*OF right*]) (*simp add*: *side fork(2)*)+
show *?thesis* **by** (*rule local-and-fork-commute*[*OF* s_2 *fork(1)*]) (*use fork(1−2) neq* s_2 *l-fresh-right r′′-fresh-left* s_1*-r* **in** *auto*)
next
case *join*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-join*, *use join left-step neq* **in** *auto*)
next
case $join_\varepsilon$
show *?thesis* **by** (*rule SLC-sym*, *rule* $SLC\text{-}join_\varepsilon$, *use* $join_\varepsilon$ *left-step neq* **in** *auto*)
qed
qed

7.2.7 Case fork

lemma *SLC-fork*:
assumes
s_1*-r*: $s_1\ r$ = *Some* (σ, τ, $\mathcal{E}$ [*Rfork e*]) **and**
s_2: s_2 = (s_1($r \mapsto$ (σ, τ, $\mathcal{E}$[*VE* (*Rid left-forkee*)]), *left-forkee* $\mapsto$ (σ;;τ, ε, *e*))) **and**
side: *left-forkee* $\notin RID_G\ s_1$ **and**
right: *revision-step r′* s_1 s_2' **and**
neq: $r \neq r'$ **and**
reach: *reachable* (s_1 :: ('*r*,'*l*,'*v*) *global-state*) **and**

rid-inf: *infinite* (*UNIV* :: *'r set*)
shows
$\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge$ (*revision-step r' s₂ s₃* $\vee$ $s_2 = s_3$) $\wedge$ (*revision-step r s₂' s₃'* $\vee$ $s_2' = s_3'$)
proof −
have *left-step*: *revision-step r s₁ s₂* **using** *s₁-r s₂ side* **by** (*auto simp add*: *ID-distr-global-conditional*)
show *?thesis*
proof (*use right* **in** ‹*cases rule*: *revision-stepE*›)
case *app*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-app*) (*use assms*(*1*−*5*) *app* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
next
case *ifTrue*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-ifTrue*) (*use assms*(*1*−*5*) *ifTrue* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
next
case *ifFalse*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-ifFalse*) (*use assms*(*1*−*5*) *ifFalse* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
next
case (*new - - - - l*)
have *l-fresh-left*: $l \notin LID_G\ s_2$
by (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add*: *s₁-r new*(*3*))+
have *r''-fresh-right*: *left-forkee* $\notin RID_G\ s_2'$
by (*rule only-fork-introduces-rids'*[*OF right*]) (*simp add*: *side new*(*2*))+
show *?thesis* **by** (*rule SLC-sym*, *rule local-and-fork-commute*[*OF new*(*1*) *s₂*]) (*use new*(*1*−*2*) *neq s₁-r r''-fresh-right l-fresh-left s₂* **in** *auto*)
next
case *get*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-get*) (*use assms*(*1*−*5*) *get* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
next
case *set*
show *?thesis* **by** (*rule SLC-sym*, *rule SLC-set*) (*use assms*(*1*−*5*) *set* **in** ‹*auto simp add*: *ID-distr-global-conditional*›)
next
case (*fork σ' τ' ℰ' e' right-forkee*)
have *r'-unchanged-left*: $s_2\ r' = s_1\ r'$ **using** *side fork*(*2*) *neq s₂* **by** *auto*
have *r-unchanged-right*: $s_2'\ r = s_1\ r$ **using** *fork*(*1*,*3*) *neq s₁-r* **by** *auto*
have $r \neq$ *left-forkee* **using** *s₁-r side* **by** *auto*
have $r \neq$ *right-forkee* **using** *fork*(*3*) *s₁-r* **by** *auto*
have $r' \neq$ *left-forkee* **using** *fork*(*2*) *side* **by** *auto*
have $r' \neq$ *right-forkee* **using** *fork*(*2*) *fork*(*3*) **by** *auto*
show *?thesis*
proof (*cases left-forkee = right-forkee*)
case *True*
obtain *r''* **where** *r''-fresh-left*: $r'' \notin RID_G\ s_2$
using *RID_G-finite-invariant ex-new-if-finite left-step reach reachable-imp-identifiers-finite*(*1*)

rid-inf **by** *blast*
hence $r'' \neq$ *left-forkee* **using** *assms*(*2*) **by** *auto*
have $r'' \neq r$ **using** r''*-fresh-left* s_2 **by** *auto*
have $r'' \neq r'$ **using** *fork*(*2*) r''*-fresh-left* r'*-unchanged-left* **by** *auto*
have $r'' \notin RID_G\ (s_1(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (Rid\ \text{left-forkee})])))$
by (*metis* (*mono-tags*, *lifting*) RID_G*-def True UnI1* ‹$r \neq$ *right-forkee*› *domIff fun-upd-other fun-upd-triv in-restricted-global-in-updated-global*(*1*) *fork*(*3*) r''*-fresh-left* s_2)
hence $r'' \notin RID_G\ (s_1(r := None))$ **by** *blast*
have r''*-fresh-*s_1: $r'' \notin RID_G\ s_1$
using ‹$r \neq$ *left-forkee*› s_2 r''*-fresh-left* s_1*-r* ‹$r'' \neq r$› ‹$r'' \notin RID_G\ (s_1(r := None))$›
by (*auto simp add*: *ID-distr-global-conditional*)
have r''*-fresh-right*: $r'' \notin RID_G\ s_2'$
using *True* ‹$r'' \neq$ *left-forkee*› ‹$r' \neq$ *right-forkee*› ‹$r'' \neq r'$› r''*-fresh-*s_1 *fork*(*2*) r''*-fresh-*s_1
by (*auto simp add*: *fork*(*1*) *ID-distr-global-conditional fun-upd-twist*)
let $?\alpha$ = *id*(*left-forkee* := r'', r'' := *left-forkee*)
have *bij-*α: *bij* $?\alpha$ **by** (*simp add*: *swap-bij*)
let $?s_3 = s_2(r' \mapsto (\sigma', \tau', \mathcal{E}'\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma';;\tau', \varepsilon, e'))$
have *left-convergence*: *revision-step* r' s_2 $?s_3$
using *fork*(*2*) r''*-fresh-left* r'*-unchanged-left revision-step.fork* **by** *auto*
let $?s_3' = s_2'(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau, \varepsilon, e))$
have *right-convergence*: *revision-step* r s_2' $?s_3'$
using r''*-fresh-right r-unchanged-right revision-step.fork* s_1*-r* **by** *auto*
have *equiv*: $?s_3 \approx ?s_3'$
proof (*rule eq-statesI*[*of* $?\alpha$ *id*])
show $\mathcal{R}_G$ $?\alpha$ *id* $?s_3 = ?s_3'$
proof −
have s_1: $\mathcal{R}_G$ $?\alpha$ *id* $s_1 = s_1$ **using** r''*-fresh-*s_1 *side* **by** *auto*
have σ: $\mathcal{R}_S$ $?\alpha$ *id* $\sigma = \sigma$ **using** r''*-fresh-*s_1 s_1*-r True fork*(*3*) **by** *auto*
have τ: $\mathcal{R}_S$ $?\alpha$ *id* $\tau = \tau$ **using** r''*-fresh-*s_1 s_1*-r True fork*(*3*) **by** *auto*
have $\mathcal{E}$: $\mathcal{R}_C$ $?\alpha$ *id* $\mathcal{E} = \mathcal{E}$
proof
show *left-forkee* $\notin RID_C\ \mathcal{E}$ **using** s_1*-r side* **by** *auto*
show $r'' \notin RID_C\ \mathcal{E}$ **using** *True* ‹$r \neq$ *right-forkee*› r''*-fresh-left* s_2 **by** *auto*
qed
have e: $\mathcal{R}_E$ $?\alpha$ *id* $e = e$
proof
show *left-forkee* $\notin RID_E\ e$ **using** s_1*-r side* **by** *auto*
show $r'' \notin RID_E\ e$ **using** *True* ‹$r \neq$ *right-forkee*› r''*-fresh-left* s_2 **by** *auto*
qed
have σ': $\mathcal{R}_S$ $?\alpha$ *id* $\sigma' = \sigma'$ **using** *fork*(*2*) r''*-fresh-*s_1 *side* **by** *auto*
have τ': $\mathcal{R}_S$ $?\alpha$ *id* $\tau' = \tau'$ **using** *fork*(*2*) r''*-fresh-*s_1 *side* **by** *auto*
have $\mathcal{E}'$: $\mathcal{R}_C$ $?\alpha$ *id* $\mathcal{E}' = \mathcal{E}'$
proof
show *left-forkee* $\notin RID_C\ \mathcal{E}'$ **using** *fork*(*2*) *side* **by** *auto*

```
          show r'' ∉ RID_C E' using fork(2) r''-fresh-s1 by auto
        qed
        have e': R_E ?α id e' = e'
        proof
          show left-forkee ∉ RID_E e' using fork(2) side by auto
          show r'' ∉ RID_E e' using fork(2) r''-fresh-s1 by auto
        qed
        show ?thesis using True fork(1) neq σ τ E e  σ' τ' E' e' s1 s2
          bij-α ‹r'' ≠ left-forkee› ‹r' ≠ left-forkee› ‹r ≠ left-forkee› ‹r'' ≠ r› ‹r''
≠ r'›
          by auto
      qed
    qed (simp add: bij-α)+
    show ?thesis using equiv left-convergence right-convergence by blast
  next
    case False
    have right-forkee-fresh-left: right-forkee ∉ RID_G s2
      using False ‹r ≠ left-forkee› ‹r ≠ right-forkee› fork(3) s1-r
      by (auto simp add: s2 ID-distr-global-conditional, auto)
    have left-forkee-fresh-right: left-forkee ∉ RID_G s2'
      using False ‹r' ≠ right-forkee› ‹r' ≠ left-forkee› side fork(2)
      by (auto simp add: fork(1) ID-distr-global-conditional fun-upd-twist)
    show ?thesis
    proof(rule exI[where x=s2(r' := s2' r', right-forkee := s2' right-forkee)],
        rule exI[where x=s2'(r := s2 r, left-forkee := s2 left-forkee)],
        rule SLC-commute)
      show s2(r' := s2' r', right-forkee := s2' right-forkee) = s2'(r := s2 r,
left-forkee := s2 left-forkee)
       using False ‹r ≠ right-forkee› ‹r' ≠ left-forkee› ‹r' ≠ right-forkee› fork(1)
neq s2 by auto
      show revision-step r' s2 (s2(r' := s2' r', right-forkee := s2' right-forkee))
       using fork(1−2) r'-unchanged-left revision-step.fork right-forkee-fresh-left
by auto
      show revision-step r s2' (s2'(r := s2 r, left-forkee := s2 left-forkee))
       using left-forkee-fresh-right r-unchanged-right revision-step.fork s1-r s2 by
auto
    qed
  qed
 next
  case join
  show ?thesis by (rule SLC-sym, rule SLC-join, use join left-step assms(3−5)
in auto)
 next
  case join_ε
  show ?thesis by (rule SLC-sym, rule SLC-join_ε, use join_ε left-step assms(3−5)
in auto)
 qed
qed
```

7.2.8 The theorem

theorem *strong-local-confluence*:
assumes
l: *revision-step r* s_1 s_2 **and**
r: *revision-step* r' s_1 $s_2{}'$ **and**
reach: *reachable* (s_1 :: ('*r*,'*l*,'*v*) *global-state*) **and**
lid-inf: *infinite* (*UNIV* :: '*l set*) **and**
rid-inf: *infinite* (*UNIV* :: '*r set*)
shows
$\exists\, s_3\ s_3{}'.\ s_3 \approx s_3{}' \wedge$ (*revision-step* r' s_2 $s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step r* $s_2{}'$ $s_3{}' \vee s_2{}' = s_3{}'$)
proof (*cases* $r = r'$)
case *True*
thus *?thesis* **by** (*metis l local-determinism r*)
next
case *neq*: *False*
thus *?thesis* **by** (*cases rule*: *revision-stepE*[*OF l*]) (*auto simp add*: *assms SLC-app SLC-ifTrue*
SLC-ifFalse SLC-new SLC-get SLC-set SLC-fork SLC-join SLC-join$_\varepsilon$)
qed

7.3 Diagram Tiling

7.3.1 Strong local confluence diagrams

lemma *SLC*:
assumes
$s_1 s_2$: $s_1 \rightsquigarrow s_2$ **and**
$s_1 s_2{}'$: $s_1 \rightsquigarrow s_2{}'$ **and**
reach: *reachable* (s_1 :: ('*r*,'*l*,'*v*) *global-state*) **and**
lid-inf: *infinite* (*UNIV* :: '*l set*) **and**
rid-inf: *infinite* (*UNIV* :: '*r set*)
shows
$\exists\, s_3\ s_3{}'.\ s_3 \approx s_3{}' \wedge s_2 \rightsquigarrow^{=} s_3 \wedge s_2{}' \rightsquigarrow^{=} s_3{}'$
proof −
from $s_1 s_2$ **obtain** *r* **where** *l*: *revision-step r* s_1 s_2 **by** *auto*
from $s_1 s_2{}'$ **obtain** r' **where** *r*: *revision-step* r' s_1 $s_2{}'$ **by** *auto*
obtain s_3 $s_3{}'$ **where**
s_3-*eq*-$s_3{}'$: $s_3 \approx s_3{}'$ **and**
l-join: *revision-step* r' s_2 $s_3 \vee s_2 = s_3$ **and**
r-join: *revision-step r* $s_2{}'$ $s_3{}' \vee s_2{}' = s_3{}'$
using *l r reach lid-inf rid-inf strong-local-confluence* **by** *metis*
have $s_2 s_3$: $s_2 \rightsquigarrow^{=} s_3$ **using** *l-join steps-def* **by** *auto*
have $s_2{}' s_3$: $s_2{}' \rightsquigarrow^{=} s_3{}'$ **using** *r-join steps-def* **by** *auto*
show *?thesis* **using** $s_2{}' s_3$ $s_2 s_3$ s_3-*eq*-$s_3{}'$ **by** *blast*
qed

lemma *SLC-top-relaxed*:
assumes

$s_1 s_2$: $s_1 \rightsquigarrow^= s_2$ **and**
$s_1 s_2'$: $s_1 \rightsquigarrow s_2'$ **and**
reach: *reachable* (s_1 :: ('*r*,'*l*,'*v*) *global-state*) **and**
lid-inf: *infinite* (*UNIV* :: '*l set*) **and**
rid-inf: *infinite* (*UNIV* :: '*r set*)
shows
$\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge s_2 \rightsquigarrow^= s_3 \wedge s_2' \rightsquigarrow^= s_3'$
proof −
have *1*: $s_1 \rightsquigarrow s_2 \Longrightarrow s_1 \rightsquigarrow s_2' \Longrightarrow$ *?thesis* **using** *SLC lid-inf reach rid-inf* **by** *blast*
have *2*: $s_1 = s_2 \Longrightarrow s_1 \rightsquigarrow s_2' \Longrightarrow$ *?thesis*
by (*rule exI*[**where** $x{=}s_2'$], *rule exI*[**where** $x{=}s_2'$]) (*auto simp add*: $\alpha\beta$-*refl*)
show *?thesis* **using** *assms 1 2* **by** *auto*
qed

7.3.2 Mimicking diagrams

declare *bind-eq-None-conv* [*simp*]
declare *bind-eq-Some-conv* [*simp*]

lemma *in-renamed-in-unlabelled-val*:
bij $\alpha \Longrightarrow (\alpha\ r \in RID_V\ (\mathcal{R}_V\ \alpha\ \beta\ v)) = (r \in RID_V\ v)$
bij $\beta \Longrightarrow (\beta\ l \in LID_V\ (\mathcal{R}_V\ \alpha\ \beta\ v)) = (l \in LID_V\ v)$
by (*auto simp add*: *bij-is-inj inj-image-mem-iff val.set-map*(*1*−*2*))

lemma *in-renamed-in-unlabelled-expr*:
bij $\alpha \Longrightarrow (\alpha\ r \in RID_E\ (\mathcal{R}_E\ \alpha\ \beta\ v)) = (r \in RID_E\ v)$
bij $\beta \Longrightarrow (\beta\ l \in LID_E\ (\mathcal{R}_E\ \alpha\ \beta\ v)) = (l \in LID_E\ v)$
by (*auto simp add*: *bij-is-inj inj-image-mem-iff expr.set-map*(*1*−*2*))

lemma *in-renamed-in-unlabelled-store*:
assumes
bij-α: *bij* α **and**
bij-β: *bij* β
shows
$(\alpha\ r \in RID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)) = (r \in RID_S\ \sigma)$
$(\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)) = (l \in LID_S\ \sigma)$
proof −
show $(\alpha\ r \in RID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)) = (r \in RID_S\ \sigma)$
proof (*rule iffI*)
assume $\alpha\ r \in RID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
thus $r \in RID_S\ \sigma$
proof (*rule* $RID_S E$)
fix *l v*
assume *map*: $\mathcal{R}_S\ \alpha\ \beta\ \sigma\ l = Some\ v$ **and** αr: $\alpha\ r \in RID_V\ v$
hence $\sigma\ (inv\ \beta\ l) = Some\ (\mathcal{R}_V\ (inv\ \alpha)\ (inv\ \beta)\ v)$
using *bij-*α *bij-*β **by** (*auto simp add*: *bij-is-inj*)
have $r \in RID_V\ (\mathcal{R}_V\ (inv\ \alpha)\ (inv\ \beta)\ v)$
using *bij-*α *bij-*β αr *map* **by** (*auto simp add*: *bij-is-inj in-renamed-in-unlabelled-val*(*1*))

show $r \in RID_S\ \sigma$
using ‹$\sigma\ (inv\ \beta\ l) = Some\ (\mathcal{R}_V\ (inv\ \alpha)\ (inv\ \beta)\ v)$› ‹$r \in RID_V\ (\mathcal{R}_V\ (inv\ \alpha)\ (inv\ \beta)\ v)$› **by** *blast*
qed
next
assume $r \in RID_S\ \sigma$
thus $\alpha\ r \in RID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
by (*metis* (*mono-tags*, *lifting*) $RID_S E$ $RID_S I$ *bij-*α *bij-*β *fun-upd-same fun-upd-triv*

in-renamed-in-unlabelled-val(*1*) *renaming-distr-store*)
qed
show $(\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)) = (l \in LID_S\ \sigma)$
proof (*rule iffI*)
assume $\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
thus $l \in LID_S\ \sigma$
proof (*rule* $LID_S E$)
assume $\beta\ l \in dom\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
thus $l \in LID_S\ \sigma$ **by** (*auto simp add*: $LID_S I(1)$ *bij-*β *bijection.intro bijection.inv-left*)
next
fix $v\ l'$
assume $\mathcal{R}_S\ \alpha\ \beta\ \sigma\ l' = Some\ v\ \beta\ l \in LID_V\ v$
thus $l \in LID_S\ \sigma$ **using** *bij-*β **by** (*auto simp add*: $LID_S I(2)$ *in-renamed-in-unlabelled-val*(*2*))
qed
next
assume $l \in LID_S\ \sigma$
thus $\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
proof (*rule* $LID_S E$)
assume $l \in dom\ \sigma$
hence $\exists \sigma'\ v.\ \sigma = (\sigma'(l \mapsto v))$ **by** *fastforce*
hence $\beta\ l \in dom\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$ **using** *bij-*β **by** *auto*
thus $\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$ **by** *auto*
next
fix $v\ l'$
assume $\sigma\ l' = Some\ v$ **and** *l-in-v*: $l \in LID_V\ v$
hence $\exists \sigma'.\ \sigma = (\sigma'(l' \mapsto v))$ **by** *force*
thus $\beta\ l \in LID_S\ (\mathcal{R}_S\ \alpha\ \beta\ \sigma)$
using *l-in-v bij-*β **by** (*auto simp add*: $LID_S I(2)$ *in-renamed-in-unlabelled-val*(*2*))
qed
qed
qed

lemma *in-renamed-in-unlabelled-local*:
assumes
bij-α: *bij* α **and**
bij-β: *bij* β
shows
$(\alpha\ r \in RID_L\ (\mathcal{R}_L\ \alpha\ \beta\ ls)) = (r \in RID_L\ ls)$
$(\beta\ l \in LID_L\ (\mathcal{R}_L\ \alpha\ \beta\ ls)) = (l \in LID_L\ ls)$

by (*cases ls*, *simp add*: *assms in-renamed-in-unlabelled-expr in-renamed-in-unlabelled-store*)+

lemma *in-renamed-in-unlabelled-global*:
assumes
bij-α: *bij* α **and**
bij-β: *bij* β
shows
$(\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)) = (r \in RID_G\ s)$
$(\beta\ l \in LID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)) = (l \in LID_G\ s)$
proof −
show $(\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)) = (r \in RID_G\ s)$
proof (*rule iffI*)
assume $\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)$
thus $r \in RID_G\ s$
proof (*rule* $RID_G E$)
assume $\alpha\ r \in dom\ (\mathcal{R}_G\ \alpha\ \beta\ s)$
hence $r \in dom\ s$ **by** (*metis bij-α domIff fun-upd-same fun-upd-triv renaming-distr-global*(*2*))
thus $r \in RID_G\ s$ **by** *auto*
next
fix r' *ls*
assume $\mathcal{R}sr'$: $\mathcal{R}_G\ \alpha\ \beta\ s\ r' = Some\ ls$ **and** αr: $\alpha\ r \in RID_L\ ls$
have *s-inv-αr'*: $s\ (inv\ \alpha\ r') = Some\ (\mathcal{R}_L\ (inv\ \alpha)\ (inv\ \beta)\ ls)$
proof −
have $inv\ \alpha\ r' \in dom\ s$ **using** $\mathcal{R}sr'$ **by** *auto*
then obtain ls' **where** *s-inv-αr*: $s\ (inv\ \alpha\ r') = Some\ ls'$ **by** *blast*
hence $\mathcal{R}_G\ \alpha\ \beta\ s\ r' = Some\ (\mathcal{R}_L\ \alpha\ \beta\ ls')$ **by** *simp*
hence $ls = (\mathcal{R}_L\ \alpha\ \beta\ ls')$ **using** $\mathcal{R}sr'$ **by** *auto*
thus *?thesis* **by** (*metis* $\mathcal{R}_L$*-inv s-inv-αr bij-α bij-β*)
qed
have *r-in*: $r \in RID_L\ (\mathcal{R}_L\ (inv\ \alpha)\ (inv\ \beta)\ ls)$
by (*metis bij-α bij-β bij-imp-bij-inv bijection.intro bijection.inv-left in-renamed-in-unlabelled-local*(*1*) *αr*)
show $r \in RID_G\ s$
using *r-in s-inv-αr'* **by** *blast*
qed
next
assume $r \in RID_G\ s$
thus $\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)$
proof (*rule* $RID_G E$)
assume $r \in dom\ s$
hence $\alpha\ r \in dom\ (\mathcal{R}_G\ \alpha\ \beta\ s)$
by (*metis* (*mono-tags*, *lifting*) *bij-α domD domI fun-upd-same fun-upd-triv renaming-distr-global*(*1*))
thus $\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)$ **by** *auto*
next
fix r' *ls*
assume $s\ r' = Some\ ls\ r \in RID_L\ ls$
thus $\alpha\ r \in RID_G\ (\mathcal{R}_G\ \alpha\ \beta\ s)$

by (*metis ID-distr-global*(*1*) *UnI2 bij-*α *bij-*β *fun-upd-triv in-renamed-in-unlabelled-local*(*1*) *insertI2 renaming-distr-global*(*1*))
qed
qed
show (β *l* $\in$ *LID*$_G$ ($\mathcal{R}_G$ α β *s*)) = (*l* $\in$ *LID*$_G$ *s*)
proof (*rule iffI*)
assume β *l* $\in$ *LID*$_G$ ($\mathcal{R}_G$ α β *s*)
from *this* **obtain** *r ls* **where** *Rs-ls*: $\mathcal{R}_G$ α β *s r* = *Some ls* **and** β*l-in-ls*: β *l* $\in$ *LID*$_L$ *ls* **by** *blast*
hence *l* $\in$ *LID*$_L$ ($\mathcal{R}_L$ (*inv* α) (*inv* β) *ls*)
by (*metis bij-*α *bij-*β *bij-imp-bij-inv bijection.intro bijection.inv-left in-renamed-in-unlabelled-local*(*2*))
hence *l* $\in$ *LID*$_G$ ($\mathcal{R}_G$ (*inv* α) (*inv* β) ($\mathcal{R}_G$ α β *s*))
by (*metis* (*mono-tags*, *lifting*) *LID*$_G$*I Rs-ls bij-*α *bij-imp-bij-inv fun-upd-idem-iff renaming-distr-global*(*1*))
thus *l* $\in$ *LID*$_G$ *s* **using** *bij-*α *bij-*β **by** (*metis* $\mathcal{R}_G$*-inv*)
next
assume *l* $\in$ *LID*$_G$ *s*
then obtain *r s*$'$ *ls* **where** *s* = (*s*$'$(*r* $\mapsto$ *ls*)) *l* $\in$ *LID*$_L$ *ls* **by** (*metis LID*$_G$*E fun-upd-triv*)
thus β *l* $\in$ *LID*$_G$ ($\mathcal{R}_G$ α β *s*) **by** (*simp add*: *bij-*α *bij-*β *in-renamed-in-unlabelled-local*(*2*))
qed
qed

lemma *mimicking*:
assumes
step: *revision-step r s s*$'$ **and**
bij-α: *bij* α **and**
bij-β: *bij* β
shows *revision-step* (α *r*) ($\mathcal{R}_G$ α β *s*) ($\mathcal{R}_G$ α β *s*$'$)
proof (*use step* **in** ‹*cases rule*: *revision-stepE*›)
case *app*
then show *?thesis* **by** (*auto simp add*: *bij-*α *bij-*β *bijection.intro bijection.inv-left renaming-distr-subst*)
next
case (*new - - - - l*)
have β*l-fresh*: β *l* $\notin$ *LID*$_G$ ($\mathcal{R}_G$ α β *s*)
by (*simp add*: *bij-*α *bij-*β *in-renamed-in-unlabelled-global*(*2*) *new*(*3*))
show *?thesis* **using** β*l-fresh new* **by** (*auto simp add*: *bij-*α *bij-*β *bijection.intro bijection.inv-left*)
next
case (*fork* σ τ $\mathcal{E}$ *e r*$'$)
have α*r*$'$*-fresh*: α *r*$'$ $\notin$ *RID*$_G$ ($\mathcal{R}_G$ α β *s*)
by (*simp add*: *bij-*α *bij-*β *in-renamed-in-unlabelled-global*(*1*) *fork*(*3*))
have *s-r-as-upd*: *s* = (*s*(*r* $\mapsto$ (σ, τ, $\mathcal{E}$ [*Rfork e*]))) **using** *fork*(*2*) **by** *auto*
have *src*: $\mathcal{R}_G$ α β *s* (α *r*) = *Some* ($\mathcal{R}_S$ α β σ, $\mathcal{R}_S$ α β τ, ($\mathcal{R}_C$ α β $\mathcal{E}$) [*Rfork* ($\mathcal{R}_E$ α β *e*)])
by (*subst s-r-as-upd*, *simp add*: *bij-*α)
show *?thesis* **using** α*r*$'$*-fresh src revision-step.fork fork*(*1*) *bij-*α **by** *auto*
qed (*auto simp add*: *bij-*α *bij-*β *bijection.intro bijection.inv-left*)

lemma *mimic-single-step*:
assumes
$s_1 s_1'$: $s_1 \approx s_1'$ **and**
$s_1 s_2$: $s_1 \rightsquigarrow s_2$
shows $\exists s_2'.\ (s_2 \approx s_2') \wedge s_1' \rightsquigarrow s_2'$
proof −
from $s_1 s_1'$ **obtain** $\alpha\ \beta$ **where**
bij-α: *bij* α **and**
bij-β: *bij* β **and**
R: $\mathcal{R}_G\ \alpha\ \beta\ s_1 = s_1'$ **by** *blast*
from $s_1 s_2$ **obtain** r **where** *step*: *revision-step* $r\ s_1\ s_2$ **by** *auto*
have *mirrored-step*: *revision-step* $(\alpha\ r)\ s_1'\ (\mathcal{R}_G\ \alpha\ \beta\ s_2)$
using R *bij-α bij-β step mimicking* **by** *auto*
have *eq*: $s_2 \approx (\mathcal{R}_G\ \alpha\ \beta\ s_2)$ **using** *bij-α bij-β* **by** *blast*
have $s_1' s_2'$: $s_1' \rightsquigarrow (\mathcal{R}_G\ \alpha\ \beta\ s_2)$ **using** *mirrored-step* **by** *auto*
from *eq* $s_1' s_2'$ **show** *?thesis* **by** *blast*
qed

lemma *mimic-trans*:
assumes
s_1-*eq*-s_1': $s_1 \approx s_1'$ **and**
$s_1 s_2$: $s_1 \rightsquigarrow^* s_2$
shows $\exists s_2'.\ s_2 \approx s_2' \wedge s_1' \rightsquigarrow^* s_2'$
proof −
from s_1-*eq*-s_1' **obtain** $\alpha\ \beta$ **where**
bij-α: *bij* α **and**
bij-β: *bij* β **and**
R: $\mathcal{R}_G\ \alpha\ \beta\ s_1 = s_1'$
by *blast*
from $s_1 s_2$ **obtain** n **where** $(s_1, s_2) \in [\rightsquigarrow]^{\frown\frown} n$ **using** *rtrancl-power* **by** *blast*
thus *?thesis*
proof (*induct n arbitrary*: s_2)
case *0*
thus *?case* **using** s_1-*eq*-s_1' **by** *auto*
next
case (*Suc n*)
obtain x **where** *n-steps*: $(s_1, x) \in [\rightsquigarrow]^{\frown\frown} n$ **and** *step*: $x \rightsquigarrow s_2$ **using** *Suc.prems*
by *auto*
obtain x' **where** *x-eq-x'*: $x \approx x'$ **and** $s_1' x$: $s_1' \rightsquigarrow^* x'$ **using** *Suc.hyps n-steps*
by *blast*
obtain s_2' **where** s_2-*eq*-s_2': $s_2 \approx s_2'$ **and** $x' s_2'$: $x' \rightsquigarrow s_2'$
by (*meson step mimic-single-step x-eq-x'*)
show *?case* **using** $s_1' x$ s_2-*eq*-s_2' *trancl-into-rtrancl* $x' s_2'$ **by** *auto*
qed
qed

7.3.3 Strip diagram

lemma *strip-lemma*:
assumes
$s_1 s_2$: $s_1 \rightsquigarrow^* s_2$ **and**
$s_1 s_2'$: $s_1 \rightsquigarrow^= s_2'$ **and**
reach: *reachable* $(s_1 :: ('r,'l,'v)\ global\text{-}state)$ **and**
lid-inf: *infinite* $(UNIV :: 'l\ set)$ **and**
rid-inf: *infinite* $(UNIV :: 'r\ set)$
shows $\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge s_2 \rightsquigarrow^* s_3 \wedge s_2' \rightsquigarrow^* s_3'$
proof −
from $s_1 s_2$ **obtain** n **where** $(s_1,\ s_2) \in [\rightsquigarrow]^{\frown\frown} n$ **using** *rtrancl-power* **by** *blast*
from *reach* $s_1 s_2'$ **and** *this* **show** *?thesis*
proof (*induct n arbitrary*: $s_1\ s_2\ s_2'$)
case *0*
hence $s_1 = s_2$ **by** *simp*
hence $s_2 s_2'$: $s_2 \rightsquigarrow^* s_2'$ **using** *0.prems*(*2*) **by** *blast*
show *?case* **by** (*rule exI*[**where** $x{=}s_2'$], *rule exI*[**where** $x{=}s_2'$]) (*use* $s_2 s_2'$ **in**
‹*simp add*: $\alpha\beta$-*refl*›)
next
case (*Suc n*)
obtain a **where** $s_1 a$: $s_1 \rightsquigarrow a$ **and** as_2: $(a,\ s_2) \in [\rightsquigarrow]^{\frown\frown} n$ **by** (*meson Suc.prems*(*3*)
relpow-Suc-D2)
obtain $b\ c$ **where** $b \approx c\ a \rightsquigarrow^= b\ s_2' \rightsquigarrow^= c$
by (*metis* (*mono-tags*, *lifting*) *SLC-top-relaxed Suc.prems*(*1*) *Suc.prems*(*2*)
$\alpha\beta$-*sym lid-inf rid-inf* $s_1 a$)
obtain $d\ e$ **where** $d \approx e\ s_2 \rightsquigarrow^* d\ b \rightsquigarrow^* e$
by (*meson Suc.hyps Suc.prems*(*1*) ‹$a \rightsquigarrow^= b$› as_2 *reachability-closed-under-execution-step*
$s_1 a$ *valid-stepE*)
obtain f **where** $c \rightsquigarrow^* f\ e \approx f$
by (*meson* ‹$b \approx c$› ‹$b \rightsquigarrow^* e$› *mimic-trans*)
have $d \approx f$ **using** $\alpha\beta$-*trans* ‹$d \approx e$› ‹$e \approx f$› **by** *auto*
then show *?case* **by** (*metis* (*no-types*, *lifting*) ‹$c \rightsquigarrow^* f$› ‹$s_2 \rightsquigarrow^* d$› ‹$s_2' \rightsquigarrow^= c$›
r-into-rtrancl rtrancl-reflcl rtrancl-trans)
qed
qed

7.3.4 Confluence diagram

lemma *confluence-modulo-equivalence*:
assumes
$s_1 s_2$: $s_1 \rightsquigarrow^* s_2$ **and**
$s_1 s_2'$: $s_1' \rightsquigarrow^* s_2'$ **and**
equiv: $s_1 \approx s_1'$ **and**
reach: *reachable* $(s_1 :: ('r,'l,'v)\ global\text{-}state)$ **and**
lid-inf: *infinite* $(UNIV :: 'l\ set)$ **and**
rid-inf: *infinite* $(UNIV :: 'r\ set)$
shows $\exists s_3\ s_3'.\ s_3 \approx s_3' \wedge s_2 \rightsquigarrow^* s_3 \wedge s_2' \rightsquigarrow^* s_3'$
proof −
obtain n **where** $(s_1,\ s_2) \in [\rightsquigarrow]^{\frown\frown} n$ **using** $s_1 s_2$ *rtrancl-power* **by** *blast*

from *reach equiv* $s_1 s_2{}'$ **and** *this* **show** *?thesis*
proof (*induct n arbitrary*: s_1 $s_1{}'$ s_2 $s_2{}'$)
case *base*: *0*
hence $s_1 = s_2$ **by** *simp*
obtain $s_2{}''$ **where** $s_1 \rightsquigarrow^* s_2{}''$ $s_2{}' \approx s_2{}''$
using $\alpha\beta$*-sym base.prems*(*2*,*3*) *mimic-trans* **by** *blast*
have $s_2 \rightsquigarrow^* s_2{}''$ **using** ‹$s_1 = s_2$› ‹$s_1 \rightsquigarrow^* s_2{}''$› **by** *blast*
show *?case* **by** (*rule exI*[**where** $x{=}s_2{}''$], *rule exI*[**where** $x{=}s_2{}'$]) (*auto simp add*: $\alpha\beta$*-sym* ‹$s_2 \rightsquigarrow^* s_2{}''$› ‹$s_2{}' \approx s_2{}''$›)
next
case *step*: (*Suc n*)
obtain *a* **where** $s_1 a$: $(s_1,\ a) \in [\rightsquigarrow]^{\frown} n$ **and** as_2: $a \rightsquigarrow s_2$ **using** *step.prems*(*4*) **by** *auto*
have *reachable a* **using** *reachability-closed-under-execution relpow-imp-rtrancl* $s_1 a$ *step.prems*(*1*) **by** *blast*
obtain *b c* **where** $a \rightsquigarrow^* b$ $s_2{}' \rightsquigarrow^* c$ $b \approx c$
using $s_1 a$ *step.hyps step.prems*(*1*−*3*) **by** *blast*
have $a \rightsquigarrow^* s_2$ **by** (*simp add*: as_2 *r-into-rtrancl*)
obtain s_3 *d* **where** $s_3 \approx d$ $s_2 \rightsquigarrow^* s_3$ $b \rightsquigarrow^* d$
by (*meson* $\alpha\beta$*-sym* ‹$a \rightsquigarrow^* b$› ‹*reachable a*› as_2 *lid-inf refl-rewritesI rid-inf strip-lemma*)
obtain $s_3{}'$ **where** $s_3 \approx s_3{}'$ $c \rightsquigarrow^* s_3{}'$
by (*meson* $\alpha\beta$*-trans* ‹$b \approx c$› ‹$b \rightsquigarrow^* d$› ‹$s_3 \approx d$› *mimic-trans*)
show *?case* **by** (*meson* ‹$c \rightsquigarrow^* s_3{}'$› ‹$s_2 \rightsquigarrow^* s_3$› ‹$s_2{}' \rightsquigarrow^* c$› ‹$s_3 \approx s_3{}'$› *transD trans-rtrancl*)
qed
qed

7.4 Determinacy

theorem *determinacy*:
assumes
prog-expr: *program-expr e* **and**
e-terminates-in-s: $e \downarrow s$ **and**
e-terminates-in-s': $e \downarrow s'$ **and**
lid-inf: *infinite* (*UNIV* :: *'l set*) **and**
rid-inf: *infinite* (*UNIV* :: *'r set*)
shows $s \approx s'$
proof −
obtain *r* **where** *x*: $(\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))) \rightsquigarrow^* s$
by (*metis e-terminates-in-s execution-def maximal-execution-def terminates-in-def*)
obtain r' **where** *y*: $(\varepsilon(r' \mapsto (\varepsilon,\varepsilon,e))) \rightsquigarrow^* s'$
by (*metis e-terminates-in-s' execution-def maximal-execution-def terminates-in-def*)
let $?\alpha = id(r := r',\ r' := r)$
have *bij-α*: *bij* $?\alpha$ **by** (*simp add*: *swap-bij*)
have *equiv*: $(\varepsilon(r \mapsto (\varepsilon,\varepsilon,e))) \approx (\varepsilon(r' \mapsto (\varepsilon,\varepsilon,e)))$
proof (*rule eq-statesI*[*of* $?\alpha$ *id*])
show $\mathcal{R}_G$ $?\alpha$ *id* $(\varepsilon(r \mapsto (\varepsilon,\ \varepsilon,\ e))) = \varepsilon(r' \mapsto (\varepsilon,\ \varepsilon,\ e))$
using *bij-α prog-expr* **by** *auto*

qed (*simp add*: *bij-α*)+
have *reach*: *reachable* (*ε*(*r* ↦ (*ε*,*ε*,*e*)))
by (*simp add*: *initial-state-reachable prog-expr*)
have ∃ *a b*. (*a* ≈ *b*) ∧ *s* ⇝* *a* ∧ *s'* ⇝* *b*
by (*rule confluence-modulo-equivalence*[*OF x y equiv reach lid-inf rid-inf*])
from *this* **obtain** *a b* **where** *s* ⇝* *a s'* ⇝* *b a* ≈ *b* **by** *blast*
have *s* = *a* **by** (*meson* ‹*s* ⇝* *a*› *e-terminates-in-s maximal-execution-def rtranclD terminates-in-def tranclD*)
have *s'* = *b* **by** (*meson* ‹*s'* ⇝* *b*› *converse-rtranclE e-terminates-in-s' maximal-execution-def terminates-in-def*)
show *?thesis* **using** ‹*a* ≈ *b*› ‹*s* = *a*› ‹*s'* = *b*› **by** *auto*
qed

end

end

References

[1] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *ACM Sigplan Notices*, volume 45, pages 691–707. ACM, 2010.

[2] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *European Symposium on Programming*, pages 116–135. Springer, 2011.

[3] R. Overbeek. Formalizing the semantics of concurrent revisions. Master's thesis, Vrije Universiteit Amsterdam/Universiteit van Amsterdam, 2018. https://raw.githubusercontent.com/overbk/verifying-concurrent-revisions/master/thesis.pdf.