

# Formalization of Concurrent Revisions

Roy Overbeek

February 23, 2021

## 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.<sup>1</sup>

## Contents

<b>1</b>	<b>Data</b>	<b>4</b>
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
<b>2</b>	<b>Occurrences</b>	<b>7</b>
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

---

<sup>1</sup>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.

<b>3</b>	<b>Renaming</b>	<b>14</b>
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
<b>4</b>	<b>Substitution</b>	<b>21</b>
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
<b>5</b>	<b>Operational Semantics</b>	<b>24</b>
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
<b>6</b>	<b>Executions</b>	<b>40</b>
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

<b>7</b>	<b>Determinacy</b>	<b>49</b>
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 . . . . .	52
7.2.5	Case local . . . . .	55
7.2.6	Case new . . . . .	59
7.2.7	Case fork . . . . .	61
7.2.8	The theorem . . . . .	64
7.3	Diagram Tiling . . . . .	65
7.3.1	Strong local confluence diagrams . . . . .	65
7.3.2	Mimicking diagrams . . . . .	66
7.3.3	Strip diagram . . . . .	70
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  $\varepsilon :: 'a \rightarrow 'b$  where
   $\varepsilon \equiv \lambda x. \text{None}$ 
```

```
fun combine :: ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) (-;;- 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 | F | T
```

```
datatype (RIDV: 'r, LIDV: 'l,'v) val =
  CV const
| Var 'v
| Loc 'l
| Rid 'r
| Lambda 'v ('r,'l,'v) expr
and (RIDE: 'r, LIDE: 'l,'v) expr =
  VE ('r,'l,'v) val
| Apply ('r,'l,'v) expr ('r,'l,'v) expr
| Ite ('r,'l,'v) expr ('r,'l,'v) expr ('r,'l,'v) expr
| Ref ('r,'l,'v) expr
| Read ('r,'l,'v) expr
| Assign ('r,'l,'v) expr ('r,'l,'v) expr
| Rfork ('r,'l,'v) expr
| Rjoin ('r,'l,'v) expr
```

```
datatype (RIDC: 'r, LIDC: 'l,'v) cntxt =
  Hole ( $\square$ )
| ApplyLE ('r,'l,'v) cntxt ('r,'l,'v) expr
| ApplyRE ('r,'l,'v) val ('r,'l,'v) cntxt
| IteE ('r,'l,'v) cntxt ('r,'l,'v) expr ('r,'l,'v) expr
| RefE ('r,'l,'v) cntxt
| ReadE ('r,'l,'v) cntxt
| AssignLE ('r,'l,'v) cntxt ('r,'l,'v) expr
| AssignRE 'l ('r,'l,'v) cntxt
| RjoinE ('r,'l,'v) cntxt
```

### 1.3 Plugging and decomposing

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

#### translations

$\mathcal{E}[x] \Rightarrow \mathcal{E} \triangleleft x$

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

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

#### inductive redex :: ('r, 'l, 'v) expr $\Rightarrow$ bool **where**

*app*:  $\text{redex} \ (\text{Apply} \ (\text{VE} \ (\text{Lambda} \ x \ e)) \ (\text{VE} \ v))$   
 $| \text{iteTrue}$ :  $\text{redex} \ (\text{Ite} \ (\text{VE} \ (\text{CV} \ T)) \ e1 \ e2)$   
 $| \text{iteFalse}$ :  $\text{redex} \ (\text{Ite} \ (\text{VE} \ (\text{CV} \ F)) \ e1 \ e2)$   
 $| \text{ref}$ :  $\text{redex} \ (\text{Ref} \ (\text{VE} \ v))$   
 $| \text{read}$ :  $\text{redex} \ (\text{Read} \ (\text{VE} \ (\text{Loc} \ l)))$   
 $| \text{assign}$ :  $\text{redex} \ (\text{Assign} \ (\text{VE} \ (\text{Loc} \ l)) \ (\text{VE} \ v))$   
 $| \text{rfork}$ :  $\text{redex} \ (\text{Rfork} \ e)$   
 $| \text{rjoin}$ :  $\text{redex} \ (\text{Rjoin} \ (\text{VE} \ (\text{Rid} \ r)))$

**inductive-simps** *redex-simps* [simp]:  $\text{redex} \ e$

**inductive-cases** *redexE* [elim]:  $\text{redex} \ e$

**lemma** *plugged-redex-not-val* [simp]:  $\text{redex} \ r \Longrightarrow (\mathcal{E} \triangleleft r) \neq (\text{VE} \ t)$  **by** (cases  $\mathcal{E}$ ) *auto*

#### inductive decompose :: ('r, 'l, 'v) expr $\Rightarrow$ ('r, 'l, 'v) cntxt $\Rightarrow$ ('r, 'l, 'v) expr $\Rightarrow$ bool **where**

*top-redex*:  $\text{redex} \ e \Longrightarrow \text{decompose} \ e \ \square \ e$   
 $| \text{lapply}$ :  $\llbracket \neg \text{redex} \ (\text{Apply} \ e1 \ e2); \text{decompose} \ e1 \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Apply} \ e1 \ e2) \ (\text{ApplyL}_{\mathcal{E}} \ \mathcal{E} \ e2) \ r$   
 $| \text{rapply}$ :  $\llbracket \neg \text{redex} \ (\text{Apply} \ (\text{VE} \ v) \ e); \text{decompose} \ e \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Apply} \ (\text{VE} \ v) \ e) \ (\text{ApplyR}_{\mathcal{E}} \ v \ \mathcal{E}) \ r$   
 $| \text{ite}$ :  $\llbracket \neg \text{redex} \ (\text{Ite} \ e1 \ e2 \ e3); \text{decompose} \ e1 \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Ite} \ e1 \ e2 \ e3) \ (\text{Ite}_{\mathcal{E}} \ \mathcal{E} \ e2 \ e3) \ r$   
 $| \text{ref}$ :  $\llbracket \neg \text{redex} \ (\text{Ref} \ e); \text{decompose} \ e \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Ref} \ e) \ (\text{Ref}_{\mathcal{E}} \ \mathcal{E}) \ r$   
 $| \text{read}$ :  $\llbracket \neg \text{redex} \ (\text{Read} \ e); \text{decompose} \ e \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Read} \ e) \ (\text{Read}_{\mathcal{E}} \ \mathcal{E}) \ r$   
 $| \text{lassign}$ :  $\llbracket \neg \text{redex} \ (\text{Assign} \ e1 \ e2); \text{decompose} \ e1 \ \mathcal{E} \ r \rrbracket \Longrightarrow \text{decompose} \ (\text{Assign} \ e1 \ e2) \ (\text{AssignL}_{\mathcal{E}} \ \mathcal{E} \ e2) \ r$

| *rassign*:  $\llbracket \neg \text{redex } (\text{Assign } (\text{VE } (\text{Loc } l)) e_2); \text{decompose } e_2 \ \mathcal{E} \ r \rrbracket \implies \text{decompose } (\text{Assign } (\text{VE } (\text{Loc } l)) e_2) (\text{AssignR}_{\mathcal{E}} \ l \ \mathcal{E}) \ r$   
 | *rjoin*:  $\llbracket \neg \text{redex } (\text{Rjoin } e); \text{decompose } e \ \mathcal{E} \ r \rrbracket \implies \text{decompose } (\text{Rjoin } e) (\text{Rjoin}_{\mathcal{E}} \ \mathcal{E}) \ r$

**inductive-cases** *decomposeE* [elim]: *decompose e E r*

**lemma** *plug-decomposition-equivalence*: *redex r  $\implies$  decompose e E r = (E[r] = e)*

**proof** (*rule iffI*)

**assume** *x*: *decompose e 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<sub>1</sub> e<sub>2</sub> E r*)

**have** (*ApplyL<sub>E</sub> E e<sub>2</sub>*)  $[r] = \text{Apply } (\mathcal{E}[r]) \ e_2$  **by** *simp*

**also have**  $\dots = \text{Apply } e_1 \ e_2$  **using**  $\langle \mathcal{E}[r] = e_1 \rangle$  **by** *simp*

**then show** *?case* **by** *simp*

**qed** *simp+*

**next**

**assume** *red*: *redex r* **and** *eq*:  $\mathcal{E}[r] = e$

**have** *decompose (E[r]) E r* **by** (*induct E*) (*use red in <auto intro: decompose.intros>*)

**thus** *decompose e E r* **by** (*simp add: eq*)

**qed**

**lemma** *unique-decomposition*: *decompose e E<sub>1</sub> r<sub>1</sub>  $\implies$  decompose e E<sub>2</sub> r<sub>2</sub>  $\implies$  E<sub>1</sub> = E<sub>2</sub>  $\wedge$  r<sub>1</sub> = r<sub>2</sub>*

**by** (*induct arbitrary: E<sub>2</sub> rule: decompose.induct*) *auto*

**lemma** *completion-eq* [*simp*]:

**assumes**

*red-e*: *redex r* **and**

*red-e'*: *redex r'*

**shows**  $(\mathcal{E}[r] = \mathcal{E}'[r']) = (\mathcal{E} = \mathcal{E}' \wedge r = r')$

**proof** (*rule iffI*)

**show**  $\mathcal{E}[r] = \mathcal{E}'[r'] \implies \mathcal{E} = \mathcal{E}' \wedge r = r'$

**proof** (*rule conjI*)

**assume** *eq*:  $\mathcal{E}[r] = \mathcal{E}'[r']$

**have** *decompose (E[r]) E r* **using** *plug-decomposition-equivalence red-e* **by** *blast*

**hence** *fst-decomp:decompose (E'[r']) E r* **by** (*simp add: eq*)

**have** *snd-decomp:decompose (E'[r']) E' r'* **using** *plug-decomposition-equivalence red-e'* **by** *blast*

**show** *cntxts-eq: E = 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  $\rightarrow$  ('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  $\rightarrow$  ('r,'l,'v) local-state

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

fun LID-snapshot :: ('r,'l,'v) local-state  $\Rightarrow$  ('r,'l,'v) store (- $\sigma$  200) where
  LID-snapshot ( $\sigma, \tau, e$ ) =  $\sigma$ 

fun LID-local-store :: ('r,'l,'v) local-state  $\Rightarrow$  ('r,'l,'v) store (- $\tau$  200) where
  LID-local-store ( $\sigma, \tau, e$ ) =  $\tau$ 

fun LID-expression :: ('r,'l,'v) local-state  $\Rightarrow$  ('r,'l,'v) expr (- $e$  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 RIDS :: ('r,'l,'v) store  $\Rightarrow$  'r set where
  RIDS  $\sigma \equiv \bigcup$  (RIDV ' ran  $\sigma$ )
```

```
definition RIDL :: ('r,'l,'v) local-state  $\Rightarrow$  'r set where
  RIDL  $s \equiv$  case  $s$  of ( $\sigma, \tau, e$ )  $\Rightarrow$  RIDS  $\sigma \cup$  RIDS  $\tau \cup$  RIDE  $e$ 
```

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

#### 2.1.2 Location identifiers

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

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

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

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

## 2.2 Inference rules

### 2.2.1 Introduction and elimination rules

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

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

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

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

**lemma**  $RID_G I$  [intro]:  
 $s \text{ r} = \text{Some } v \Rightarrow r \in RID_G s$   
 $s \text{ r}' = \text{Some } ls \Rightarrow r \in RID_L ls \Rightarrow r \in RID_G s$   
**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**  
 $r \in RID_G s$   
 $r \in \text{dom } s \Rightarrow P$   
 $\bigwedge r' ls. s \text{ r}' = \text{Some } ls \Rightarrow r \in RID_L ls \Rightarrow P$   
**shows**  $P$   
**using** assms **by** (auto simp add:  $RID_G$ -def ran-def)

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

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



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

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

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

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

**lemma**  $LID_G E$  [*elim*]:  $l \in LID_G s \implies (\bigwedge r ls. s\ r = Some\ ls \implies l \in LID_L ls \implies P) \implies P$   
**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**  $r\text{-}v: r \in RID_V v$  **by**

*blast*

**have**  $l' \neq l$  **using**  $\langle (\sigma(l := None))\ l' = Some\ v \rangle$  **by** *auto*

**hence**  $\sigma\ l' = Some\ v$  **using**  $\langle (\sigma(l := None))\ l' = Some\ v \rangle$  **by** *auto*

**thus**  $r \in RID_S \sigma$  **using**  $r\text{-}v$  **by** *blast*

**qed**

**next**

**show**  $LID_S (\sigma(l := None)) \subseteq LID_S \sigma$

**proof** (*rule subsetI*)  
**fix**  $l'$   
**assume**  $l' \in LID_S (\sigma(l := None))$   
**hence**  $l' \in \text{dom} (\sigma(l := None)) \vee (\exists l'' v. (\sigma(l := None)) l'' = \text{Some } v \wedge l' \in LID_V v)$  **by** *blast*  
**thus**  $l' \in LID_S \sigma$   
**proof** (*rule disjE*)  
**assume**  $l' \in \text{dom} (\sigma(l := None))$   
**thus**  $l' \in LID_S \sigma$  **by** *auto*  
**next**  
**assume**  $\exists l'' v. (\sigma(l := None)) l'' = \text{Some } v \wedge l' \in LID_V v$   
**then obtain**  $l'' v$  **where**  $(\sigma(l := None)) l'' = \text{Some } v$  **and**  $l'\text{-in-}v: l' \in LID_V v$  **by** *blast*  
**hence**  $\sigma l'' = \text{Some } v$  **by** (*cases*  $l = l'', \text{auto}$ )  
**thus**  $l' \in LID_S \sigma$  **using**  $l'\text{-in-}v$  **by** *blast*  
**qed**  
**qed**  
**qed**

**lemma** *in-restricted-store-in-unrestricted-store* [*intro*]:  
 $r \in RID_S (\sigma(l := None)) \implies r \in RID_S \sigma$   
 $l' \in LID_S (\sigma(l := None)) \implies l' \in LID_S \sigma$   
**by** (*meson contra-subsetD ID-restricted-store-subset-store*)+

**lemma** *in-restricted-store-in-updated-store* [*intro*]:  
 $r \in RID_S (\sigma(l := None)) \implies r \in RID_S (\sigma(l \mapsto v))$   
 $l' \in LID_S (\sigma(l := None)) \implies l' \in LID_S (\sigma(l \mapsto v))$   
**proof** –  
**assume**  $r \in RID_S (\sigma(l := None))$   
**hence**  $r \in RID_S (\sigma \mid' (- \{l\}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**hence**  $r \in RID_S (\sigma(l \mapsto v) \mid' (- \{l\}))$  **by** *simp*  
**hence**  $r \in RID_S (\sigma(l := \text{Some } v, l := None))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**thus**  $r \in RID_S (\sigma(l \mapsto v))$  **by** *blast*  
**next**  
**assume**  $l' \in LID_S (\sigma(l := None))$   
**hence**  $l' \in LID_S (\sigma \mid' (- \{l\}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**hence**  $l' \in LID_S (\sigma(l \mapsto v) \mid' (- \{l\}))$  **by** *simp*  
**hence**  $l' \in LID_S (\sigma(l := \text{Some } v, l := None))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**thus**  $l' \in LID_S (\sigma(l \mapsto v))$  **by** *blast*  
**qed**

**lemma** *ID-distr-store* [*simp*]:  
 $RID_S (\tau(l \mapsto v)) = RID_S (\tau(l := None)) \cup RID_V v$   
 $LID_S (\tau(l \mapsto v)) = \text{insert } l (LID_S (\tau(l := None)) \cup LID_V v)$   
**proof** –  
**show**  $RID_S (\tau(l \mapsto v)) = RID_S (\tau(l := None)) \cup RID_V v$   
**proof** (*rule set-eqI, rule iffI*)  
**fix**  $r$   
**assume**  $r \in RID_S (\tau(l \mapsto v))$

**then obtain**  $l' v'$  **where**  $(\tau(l \mapsto v)) l' = \text{Some } v' r \in \text{RID}_V v'$  **by** *blast*  
**thus**  $r \in \text{RID}_S (\tau(l := \text{None})) \cup \text{RID}_V v$  **by** *(cases l' = l) auto*  
**qed** *auto*  
**next**  
**show**  $\text{LID}_S (\tau(l \mapsto v)) = \text{insert } l (\text{LID}_S (\tau(l := \text{None})) \cup \text{LID}_V v)$   
**proof** *(rule set-eqI, rule iffI)*  
**fix**  $l'$   
**assume**  $l' \in \text{LID}_S (\tau(l \mapsto v))$   
**hence**  $l' \in \text{dom } (\tau(l \mapsto v)) \vee (\exists l'' v'. (\tau(l \mapsto v)) l'' = \text{Some } v' \wedge l' \in \text{LID}_V v')$  **by** *blast*  
**thus**  $l' \in \text{insert } l (\text{LID}_S (\tau(l := \text{None})) \cup \text{LID}_V v)$   
**proof** *(rule disjE)*  
**assume**  $l' \in \text{dom } (\tau(l \mapsto v))$   
**thus**  $l' \in \text{insert } l (\text{LID}_S (\tau(l := \text{None})) \cup \text{LID}_V v)$  **by** *(cases l' = l) auto*  
**next**  
**assume**  $\exists l'' v'. (\tau(l \mapsto v)) l'' = \text{Some } v' \wedge l' \in \text{LID}_V v'$   
**then obtain**  $l'' v'$  **where**  $(\tau(l \mapsto v)) l'' = \text{Some } v' l' \in \text{LID}_V v'$  **by** *blast*  
**thus**  $l' \in \text{insert } l (\text{LID}_S (\tau(l := \text{None})) \cup \text{LID}_V v)$  **by** *(cases l = l'') auto*  
**qed**  
**qed** *auto*  
**qed**

**lemma** *ID-distr-local [simp]:*  
 $\text{LID}_L (\sigma, \tau, e) = \text{LID}_S \sigma \cup \text{LID}_S \tau \cup \text{LID}_E e$   
 $\text{RID}_L (\sigma, \tau, e) = \text{RID}_S \sigma \cup \text{RID}_S \tau \cup \text{RID}_E e$   
**by** *(simp add: LID\_L-def RID\_L-def)+*

**lemma** *ID-restricted-global-subset-unrestricted:*  
 $\text{LID}_G (s(r := \text{None})) \subseteq \text{LID}_G s$   
 $\text{RID}_G (s(r := \text{None})) \subseteq \text{RID}_G s$   
**proof** –  
**show**  $\text{LID}_G (s(r := \text{None})) \subseteq \text{LID}_G s$   
**proof** *(rule subsetI)*  
**fix**  $l$   
**assume**  $l \in \text{LID}_G (s(r := \text{None}))$   
**then obtain**  $r'' ls$  **where**  $(s(r := \text{None})) r'' = \text{Some } ls$  **and** *l-in-ls: l ∈ LID\_L ls* **by** *blast*  
**have**  $r'' \neq r$  **using**  $\langle (s(r := \text{None})) r'' = \text{Some } ls \rangle$  **by** *auto*  
**hence**  $s r'' = \text{Some } ls$  **using**  $\langle (s(r := \text{None})) r'' = \text{Some } ls \rangle$  **by** *auto*  
**thus**  $l \in \text{LID}_G s$  **using** *l-in-ls* **by** *blast*  
**qed**  
**next**  
**show**  $\text{RID}_G (s(r := \text{None})) \subseteq \text{RID}_G s$   
**proof** *(rule subsetI)*  
**fix**  $r'$   
**assume**  $r' \in \text{RID}_G (s(r := \text{None}))$   
**hence**  $r' \in \text{dom } (s(r := \text{None})) \vee (\exists r'' ls. (s(r := \text{None})) r'' = \text{Some } ls \wedge r' \in \text{RID}_L ls)$  **by** *blast*  
**thus**  $r' \in \text{RID}_G s$

**proof** (*rule disjE*)  
**assume**  $\exists r'' \text{ ls. } (s(r := \text{None})) r'' = \text{Some ls} \wedge r' \in \text{RID}_L \text{ ls}$   
**then obtain**  $r'' \text{ ls where } (s(r := \text{None})) r'' = \text{Some ls}$  **and**  $r'\text{-in-ls: } r' \in \text{RID}_L \text{ ls}$  **by blast**  
**have**  $\text{neq: } r'' \neq r$  **using**  $\langle (s(r := \text{None})) r'' = \text{Some ls} \rangle$  **by auto**  
**hence**  $s r'' = \text{Some ls}$  **using**  $\langle (s(r := \text{None})) r'' = \text{Some ls} \rangle$  **by auto**  
**thus**  $r' \in \text{RID}_G s$  **using**  $r'\text{-in-ls}$  **by blast**  
**qed auto**  
**qed**  
**qed**

**lemma** *in-restricted-global-in-unrestricted-global* [*intro*]:  
 $r' \in \text{RID}_G (s(r := \text{None})) \implies r' \in \text{RID}_G s$   
 $l \in \text{LID}_G (s(r := \text{None})) \implies l \in \text{LID}_G s$   
**by** (*simp add: ID-restricted-global-subset-unrestricted rev-subsetD*)**+**

**lemma** *in-restricted-global-in-updated-global* [*intro*]:  
 $r' \in \text{RID}_G (s(r := \text{None})) \implies r' \in \text{RID}_G (s(r \mapsto \text{ls}))$   
 $l \in \text{LID}_G (s(r := \text{None})) \implies l \in \text{LID}_G (s(r \mapsto \text{ls}))$   
**proof** –  
**assume**  $r' \in \text{RID}_G (s(r := \text{None}))$   
**hence**  $r' \in \text{RID}_G (s \mid' (- \{r\}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**hence**  $r' \in \text{RID}_G (s(r \mapsto \text{ls}) \mid' (- \{r\}))$  **by simp**  
**hence**  $r' \in \text{RID}_G (s(r := \text{Some ls}, r := \text{None}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**thus**  $r' \in \text{RID}_G (s(r \mapsto \text{ls}))$  **by blast**  
**next**  
**assume**  $l \in \text{LID}_G (s(r := \text{None}))$   
**hence**  $l \in \text{LID}_G (s \mid' (- \{r\}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**hence**  $l \in \text{LID}_G (s(r \mapsto \text{ls}) \mid' (- \{r\}))$  **by simp**  
**hence**  $l \in \text{LID}_G (s(r := \text{Some ls}, r := \text{None}))$  **by** (*simp add: restrict-complement-singleton-eq*)  
**thus**  $l \in \text{LID}_G (s(r \mapsto \text{ls}))$  **by blast**  
**qed**

**lemma** *ID-distr-global* [*simp*]:  
 $\text{RID}_G (s(r \mapsto \text{ls})) = \text{insert } r (\text{RID}_G (s(r := \text{None})) \cup \text{RID}_L \text{ ls})$   
 $\text{LID}_G (s(r \mapsto \text{ls})) = \text{LID}_G (s(r := \text{None})) \cup \text{LID}_L \text{ ls}$   
**proof** –  
**show**  $\text{LID}_G (s(r \mapsto \text{ls})) = \text{LID}_G (s(r := \text{None})) \cup \text{LID}_L \text{ ls}$   
**proof** (*rule set-eqI*)  
**fix**  $l$   
**show**  $(l \in \text{LID}_G (s(r \mapsto \text{ls}))) = (l \in \text{LID}_G (s(r := \text{None})) \cup \text{LID}_L \text{ ls})$   
**proof** (*rule iffI*)  
**assume**  $l \in \text{LID}_G (s(r \mapsto \text{ls}))$   
**from this obtain**  $r' \text{ ls' where } (s(r \mapsto \text{ls})) r' = \text{Some ls'}$   $l \in \text{LID}_L \text{ ls'}$  **by auto**  
**thus**  $l \in \text{LID}_G (s(r := \text{None})) \cup \text{LID}_L \text{ ls}$  **by** (*cases r = r'*) **auto**  
**qed auto**  
**qed**  
**show**  $\text{RID}_G (s(r \mapsto \text{ls})) = \text{insert } r (\text{RID}_G (s(r := \text{None})) \cup \text{RID}_L \text{ ls})$   
**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  $neg: r' \neq r$  and  $r'-l: r' \notin RID_L\ ls$ 
  show  $r' \in RID_G (s(r := None))$ 
  proof (rule RID_GE[OF r'-g])
    assume  $r' \in dom (s(r \mapsto ls))$ 
    thus ?thesis by (simp add: RID_G-def neg)
  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 neg r'-l in auto)
  qed
qed auto
qed
qed

```

**lemma** *restrictions-inwards* [*simp*]:  
 $x \neq x' \implies f(x := Some\ y, x' := None) = (f(x := None, x := Some\ y))$   
**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]:  
 $r \in RID_S (\sigma;;\tau) \implies r \notin RID_S \sigma \implies r \in RID_S \tau$   
 $r \in RID_S (\sigma;;\tau) \implies r \notin RID_S \tau \implies r \in RID_S \sigma$   
 $l \in LID_S (\sigma;;\tau) \implies l \notin LID_S \sigma \implies l \in LID_S \tau$   
 $l \in LID_S (\sigma;;\tau) \implies l \notin LID_S \tau \implies l \in LID_S \sigma$   
**by** (*meson Un-iff ID-combination-subset-union subsetCE*)+

**lemma** *in-mapped-in-component-of-combination* [intro]:  
**assumes**  
  *maps-to-v*:  $(\sigma;;\tau) \ l = \text{Some } v$  **and**  
  *l'-in-v*:  $l' \in LID_V v$   
**shows**  
   $l' \notin LID_S \tau \implies l' \in LID_S \sigma$   
   $l' \notin LID_S \sigma \implies l' \in LID_S \tau$   
**by** (*metis LID\_S I(2) combine.simps l'-in-v maps-to-v*)+

**lemma** *elim-trivial-restriction* [simp]:  $l \notin LID_S \tau \implies (\tau(l := \text{None})) = \tau$   
**by** (*simp add: LID\_S-def domIff fun-upd-idem*)

**lemma** *ID-distr-global-conditional*:  
 $s \ r = \text{Some } ls \implies RID_G s = \text{insert } r (RID_G (s(r := \text{None})) \cup RID_L ls)$   
 $s \ r = \text{Some } ls \implies LID_G s = LID_G (s(r := \text{None})) \cup LID_L ls$   
**proof** –  
  **assume**  $s \ r = \text{Some } ls$   
  **hence** *s-as-upd*:  $s = (s(r \mapsto ls))$  **by** (*simp add: fun-upd-idem*)  
  **show**  $RID_G s = \text{insert } r (RID_G (s(r := \text{None})) \cup RID_L ls)$  **by** (*subst s-as-upd, simp*)  
  **show**  $LID_G s = LID_G (s(r := \text{None})) \cup LID_L ls$  **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**  $\text{rename-val} :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{val} \Rightarrow ('r, 'l, 'v) \text{val}$   
 $(\mathcal{R}_V)$  **where**

$$\mathcal{R}_V \alpha \beta v \equiv \text{map-val } \alpha \beta \text{ id } v$$

**abbreviation**  $\text{rename-expr} :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{expr} \Rightarrow ('r, 'l, 'v) \text{expr}$   
 $(\mathcal{R}_E)$  **where**

$$\mathcal{R}_E \alpha \beta e \equiv \text{map-expr } \alpha \beta \text{ id } e$$

**abbreviation**  $\text{rename-cntxt} :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{cntxt} \Rightarrow ('r, 'l, 'v) \text{cntxt}$   
 $(\mathcal{R}_C)$  **where**

$$\mathcal{R}_C \alpha \beta \mathcal{E} \equiv \text{map-cntxt } \alpha \beta \text{ id } \mathcal{E}$$

**definition**  $\text{is-store-renaming} :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{store} \Rightarrow ('r, 'l, 'v) \text{store} \Rightarrow \text{bool}$  **where**

$$\text{is-store-renaming } \alpha \beta \sigma \sigma' \equiv \forall l. \text{ case } \sigma \ l \text{ of } \text{None} \Rightarrow \sigma' (\beta \ l) = \text{None} \mid \text{Some } v \Rightarrow \sigma' (\beta \ l) = \text{Some } (\mathcal{R}_V \alpha \beta \ v)$$

**notation**  $\text{Option.bind}$  (**infixl**  $\gg=$  80)

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

**lemma**  $\mathcal{R}_S$ -implements-renaming:  $\text{bij } \beta \Longrightarrow \text{is-store-renaming } \alpha \beta \sigma (\mathcal{R}_S \alpha \beta \sigma)$

**proof** –

assume  $\text{bij } \beta$

hence  $\text{inj } \beta$  using  $\text{bij-def}$  by *auto*

thus *?thesis* by (*auto simp add: is-store-renaming-def option.case-eq-if*)

**qed**

**fun**  $\mathcal{R}_L :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{local-state} \Rightarrow ('r, 'l, 'v) \text{local-state}$  **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**  $\text{is-global-renaming} :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{global-state} \Rightarrow ('r, 'l, 'v) \text{global-state} \Rightarrow \text{bool}$  **where**

$$\text{is-global-renaming } \alpha \beta s \ s' \equiv \forall r. \text{ case } s \ r \text{ of } \text{None} \Rightarrow s' (\alpha \ r) = \text{None} \mid \text{Some } ls \Rightarrow s' (\alpha \ r) = \text{Some } (\mathcal{R}_L \alpha \beta \ ls)$$

**fun**  $\mathcal{R}_G :: ('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r, 'l, 'v) \text{global-state} \Rightarrow ('r, 'l, 'v) \text{global-state}$  **where**

$$\mathcal{R}_G \alpha \beta s \ r = s (\text{inv } \alpha \ r) \gg= (\lambda ls. \text{Some } (\mathcal{R}_L \alpha \beta \ ls))$$

**lemma**  $\mathcal{R}_G$ -implements-renaming:  $\text{bij } \alpha \Longrightarrow \text{is-global-renaming } \alpha \beta s (\mathcal{R}_G \alpha \beta s)$

**proof** –

assume  $\text{bij } \alpha$

hence  $\text{inj } \alpha$  using  $\text{bij-def}$  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- $\beta$* : *bij*  $\beta$  **and**

*none-case*:  $\bigwedge l. \sigma l = \text{None} \implies \sigma' (\beta l) = \text{None}$  **and**

*some-case*:  $\bigwedge l v. \sigma l = \text{Some } v \implies \sigma' (\beta l) = \text{Some } (\mathcal{R}_V \alpha \beta v)$

shows

$\mathcal{R}_S \alpha \beta \sigma = \sigma'$

**proof** (rule ext, subst  $\mathcal{R}_S$ .simps)

fix  $l$

show  $\sigma (\text{inv } \beta l) \gg= (\lambda v. \text{Some } (\mathcal{R}_V \alpha \beta v)) = \sigma' l$  (**is** ?lhs =  $\sigma' l$ )

**proof** (cases  $\sigma (\text{inv } \beta l) = \text{None}$ )

case *True*

have *lhs-none*: ?lhs = *None* **by** (simp add: *True*)

have  $\sigma' (\beta (\text{inv } \beta l)) = \text{None}$  **by** (simp add: *none-case True*)

hence *rhs-none*:  $\sigma' l = \text{None}$  **by** (simp add: *bij- $\beta$  bijection.intro bijection.inv-right*)

show ?thesis **by** (simp add: *lhs-none rhs-none*)

next

case *False*

from *this* obtain  $v$  where *is-some*:  $\sigma (\text{inv } \beta l) = \text{Some } v$  **by** blast

hence *lhs-some*: ?lhs = *Some* ( $\mathcal{R}_V \alpha \beta v$ ) **by** auto

have  $\sigma' (\beta (\text{inv } \beta l)) = \text{Some } (\mathcal{R}_V \alpha \beta v)$  **by** (simp add: *is-some some-case*)

hence *rhs-some*:  $\sigma' l = \text{Some } (\mathcal{R}_V \alpha \beta v)$  **by** (simp add: *bij- $\beta$  bijection.intro*

*bijection.inv-right*)

then show ?thesis **by** (simp add: *lhs-some*)

qed

qed

lemma  $\mathcal{R}_G I$  [intro]:

assumes

*bij- $\alpha$* : *bij*  $\alpha$  **and**

*none-case*:  $\bigwedge r. s r = \text{None} \implies s' (\alpha r) = \text{None}$  **and**

*some-case*:  $\bigwedge r \sigma \tau e. s r = \text{Some } (\sigma, \tau, e) \implies s' (\alpha r) = \text{Some } (\mathcal{R}_L \alpha \beta (\sigma, \tau, e))$

shows

$\mathcal{R}_G \alpha \beta s = s'$

**proof** (rule ext, subst  $\mathcal{R}_G$ .simps)

fix  $r$

show  $s (\text{inv } \alpha r) \gg= (\lambda ls. \text{Some } (\mathcal{R}_L \alpha \beta ls)) = s' r$  (**is** ?lhs =  $s' r$ )

**proof** (cases  $s (\text{inv } \alpha r) = \text{None}$ )

case *True*

have *lhs-none*: ?lhs = *None* **by** (simp add: *True*)

have  $s' (\alpha (\text{inv } \alpha r)) = \text{None}$  **by** (simp add: *none-case True*)

hence *rhs-none*:  $s' r = \text{None}$  **by** (simp add: *bij- $\alpha$  bijection.intro bijection.inv-right*)

show ?thesis **by** (simp add: *lhs-none rhs-none*)

next

case *False*

from *this* obtain  $ls$  where  $s (\text{inv } \alpha r) = \text{Some } ls$  **by** blast

from *this* obtain  $\sigma \tau e$  where *is-some*:  $s (\text{inv } \alpha r) = \text{Some } (\sigma, \tau, e)$  **by** (cases  $ls$ ) blast



**hence** *lhs-some*:  $?lhs = \text{Some } (\mathcal{R}_L \alpha \beta (\sigma, \tau, e))$  **by** *auto*  
**have**  $s' (\alpha (\text{inv } \alpha r)) = \text{Some } (\mathcal{R}_L \alpha \beta (\sigma, \tau, e))$  **by** (*simp add: is-some some-case*)  
**hence** *rhs-some*:  $s' r = \text{Some } (\mathcal{R}_L \alpha \beta (\sigma, \tau, e))$  **by** (*simp add: bij- $\alpha$  bijec- tion.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**  $\mathcal{R}_S\text{-id}$  [*simp*]:  $\mathcal{R}_S \text{ id id } \sigma = \sigma$  **by** *auto*  
**lemma**  $\mathcal{R}_L\text{-id}$  [*simp*]:  $\mathcal{R}_L \text{ id id } ls = ls$  **by** (*cases ls simp*)  
**lemma**  $\mathcal{R}_G\text{-id}$  [*simp*]:  $\mathcal{R}_G \text{ id id } s = s$  **by** *auto*

#### 3.3.2 Composition

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

#### 3.3.3 Inverse

**lemma**  $\mathcal{R}_V\text{-inv}$  [*simp*]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_V (\text{inv } \alpha) (\text{inv } \beta) v' = v) = (\mathcal{R}_V \alpha \beta v = v')$   
**by** (*auto simp add: bijec- tion.intro bijec- tion.inv-comp-right bijec- tion.inv-comp-left*)  
**lemma**  $\mathcal{R}_E\text{-inv}$  [*simp*]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_E (\text{inv } \alpha) (\text{inv } \beta) e' = e) = (\mathcal{R}_E \alpha \beta e = e')$   
**by** (*auto simp add: bijec- tion.intro bijec- tion.inv-comp-right bijec- tion.inv-comp-left*)  
**lemma**  $\mathcal{R}_C\text{-inv}$  [*simp*]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_C (\text{inv } \alpha) (\text{inv } \beta) \mathcal{E}' = \mathcal{E}) = (\mathcal{R}_C \alpha \beta \mathcal{E} = \mathcal{E}')$   
**by** (*auto simp add: bijec- tion.intro bijec- tion.inv-comp-right bijec- tion.inv-comp-left*)  
**lemma**  $\mathcal{R}_S\text{-inv}$  [*simp*]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_S (\text{inv } \alpha) (\text{inv } \beta) \sigma' = \sigma) = (\mathcal{R}_S \alpha \beta \sigma = \sigma')$   
**by** (*auto simp add: bij-imp-bij-inv bijec- tion.intro bijec- tion.inv-comp-right bijec- tion.inv-comp-left*)

**lemma**  $\mathcal{R}_L$ -inv [simp]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_L (\text{inv } \alpha) (\text{inv } \beta) \text{ls}' = \text{ls}) = (\mathcal{R}_L \alpha \beta \text{ls} = \text{ls}')$

**by** (auto simp add: bij-imp-bij-inv bijection.intro bijection.inv-comp-right bijection.inv-comp-left)

**lemma**  $\mathcal{R}_G$ -inv [simp]:  $\llbracket \text{bij } \alpha; \text{bij } \beta \rrbracket \implies (\mathcal{R}_G (\text{inv } \alpha) (\text{inv } \beta) s' = s) = (\mathcal{R}_G \alpha \beta s = s')$

**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  $\Rightarrow$   $(r, l, v)$  global-state  $\Rightarrow$  bool  $(- \approx - [100, 100])$  **where**

$s \approx s' \equiv \exists \alpha \beta. \text{bij } \alpha \wedge \text{bij } \beta \wedge \mathcal{R}_G \alpha \beta s = s'$

**lemma** eq-statesI [intro]:

$\mathcal{R}_G \alpha \beta s = s' \implies \text{bij } \alpha \implies \text{bij } \beta \implies s \approx s'$

**using** eq-states-def **by** auto

**lemma** eq-statesE [elim]:

$s \approx s' \implies (\bigwedge \alpha \beta. \mathcal{R}_G \alpha \beta s = s' \implies \text{bij } \alpha \implies \text{bij } \beta \implies P) \implies P$

**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:  $s \approx s' \implies s' \approx s'' \implies s \approx s''$

**proof** –

**assume**  $s \approx s'$

**from this obtain**  $\alpha \beta$  **where**  $s$ - $s'$ :  $\text{bij } \alpha \text{bij } \beta \mathcal{R}_G \alpha \beta s = s'$  **by** blast

**assume**  $s' \approx s''$

**from this obtain**  $\alpha' \beta'$  **where**  $s'$ - $s''$ :  $\text{bij } \alpha' \text{bij } \beta' \mathcal{R}_G \alpha' \beta' s' = s''$  **by** blast

**show**  $s \approx s''$  **by** (rule eq-statesI[of  $\alpha' \circ \alpha \beta' \circ \beta$ ]) (use  $s$ - $s'$   $s'$ - $s''$  **in**  $\langle$ auto simp add: bij-comp $\rangle$ )

**qed**

**lemma**  $\alpha\beta$ -sym:  $s \approx s' \implies s' \approx s$

**proof** –

**assume**  $s \approx s'$

**from this obtain**  $\alpha \beta$  **where**  $s$ - $s'$ :  $\text{bij } \alpha \text{bij } \beta \mathcal{R}_G \alpha \beta s = s'$  **by** blast

**show**  $s' \approx s$  **by** (rule eq-statesI[of  $\text{inv } \alpha \text{inv } \beta$ ]) (use  $s$ - $s'$  **in**  $\langle$ auto simp add: bij-imp-bij-inv $\rangle$ )

**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]:

$$\text{bij } \beta \implies \mathcal{R}_S \alpha \beta (\sigma(l \mapsto v)) = \mathcal{R}_S \alpha \beta \sigma(\beta l \mapsto \mathcal{R}_V \alpha \beta v)$$

**by** (*auto simp add: bijection.intro bijection.inv-left-eq-iff*)

### 3.4.3 Global

**lemma** *renaming-distr-global* [simp]:

$$\text{bij } \alpha \implies \mathcal{R}_G \alpha \beta (s(r \mapsto ls)) = \mathcal{R}_G \alpha \beta s(\alpha r \mapsto \mathcal{R}_L \alpha \beta ls)$$

$$\text{bij } \alpha \implies \mathcal{R}_G \alpha \beta (s(r := \text{None})) = (\mathcal{R}_G \alpha \beta s)(\alpha r := \text{None})$$

**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*:

$$\text{bij } (id(x := x', x' := x)) \text{ (is bij ?f)}$$

**proof** (*rule bijI*)

**show** *inj ?f* **by** (*simp add: inj-on-def*)

**show** *surj ?f*

**proof**

**show**  $UNIV \subseteq \text{range } (id(x := x', x' := x))$

**proof** (*rule subsetI*)

**fix**  $y$

**assume**  $y \in (UNIV :: 'a \text{ set})$

**show**  $y \in \text{range } (id(x := x', x' := x))$  **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**

$v :: ('r, 'l, 'v) \text{ val}$  **and**

$e :: ('r, 'l, 'v) \text{ expr}$

**shows**

$$l \notin LID_V v \implies \mathcal{R}_V \alpha (\beta(l := l')) v = \mathcal{R}_V \alpha \beta v$$

$$l \notin LID_E e \implies \mathcal{R}_E \alpha (\beta(l := l')) e = \mathcal{R}_E \alpha \beta e$$

$$\begin{aligned}
r \notin RID_V v &\implies \mathcal{R}_V (\alpha(r := r')) \beta v = \mathcal{R}_V \alpha \beta v \\
r \notin RID_E e &\implies \mathcal{R}_E (\alpha(r := r')) \beta e = \mathcal{R}_E \alpha \beta e
\end{aligned}$$

**proof** –

$$\begin{aligned}
\text{have } (\forall \alpha \beta r r'. r \notin RID_V v \implies \mathcal{R}_V (\alpha(r := r')) \beta v = \mathcal{R}_V \alpha \beta v) \wedge \\
(\forall \alpha \beta r r'. r \notin RID_E e \implies \mathcal{R}_E (\alpha(r := r')) \beta e = \mathcal{R}_E \alpha \beta e)
\end{aligned}$$

**by** (induct rule: val-expr.induct) simp+

**thus**

$$\begin{aligned}
r \notin RID_V v &\implies \mathcal{R}_V (\alpha(r := r')) \beta v = \mathcal{R}_V \alpha \beta v \\
r \notin RID_E e &\implies \mathcal{R}_E (\alpha(r := r')) \beta e = \mathcal{R}_E \alpha \beta e
\end{aligned}$$

**by** simp+

$$\begin{aligned}
\text{have } (\forall \alpha \beta l l'. l \notin LID_V v \implies \mathcal{R}_V \alpha (\beta(l := l')) v = \mathcal{R}_V \alpha \beta v) \wedge \\
(\forall \alpha \beta l l'. l \notin LID_E e \implies \mathcal{R}_E \alpha (\beta(l := l')) e = \mathcal{R}_E \alpha \beta e)
\end{aligned}$$

**by** (induct rule: val-expr.induct) simp+

**thus**

$$\begin{aligned}
l \notin LID_V v &\implies \mathcal{R}_V \alpha (\beta(l := l')) v = \mathcal{R}_V \alpha \beta v \text{ and} \\
l \notin LID_E e &\implies \mathcal{R}_E \alpha (\beta(l := l')) e = \mathcal{R}_E \alpha \beta e
\end{aligned}$$

**by** simp+

**qed**

**lemma** *eliminate-renaming-cntxt* [simp]:

$$\begin{aligned}
r \notin RID_C \mathcal{E} &\implies \mathcal{R}_C (\alpha(r := r')) \beta \mathcal{E} = \mathcal{R}_C \alpha \beta \mathcal{E} \\
l \notin LID_C \mathcal{E} &\implies \mathcal{R}_C \alpha (\beta(l := l')) \mathcal{E} = \mathcal{R}_C \alpha \beta \mathcal{E}
\end{aligned}$$

**by** (induct  $\mathcal{E}$  rule: cntxt.induct) auto

**lemma** *eliminate-swap-val* [simp, intro]:

$$\begin{aligned}
r \notin RID_V v &\implies r' \notin RID_V v \implies \mathcal{R}_V (id(r := r', r' := r)) id v = v \\
l \notin LID_V v &\implies l' \notin LID_V v \implies \mathcal{R}_V id (id(l := l', l' := l)) v = v
\end{aligned}$$

**by** simp+

**lemma** *eliminate-swap-expr* [simp, intro]:

$$\begin{aligned}
r \notin RID_E e &\implies r' \notin RID_E e \implies \mathcal{R}_E (id(r := r', r' := r)) id e = e \\
l \notin LID_E e &\implies l' \notin LID_E e \implies \mathcal{R}_E id (id(l := l', l' := l)) e = e
\end{aligned}$$

**by** simp+

**lemma** *eliminate-swap-cntxt* [simp, intro]:

$$\begin{aligned}
r \notin RID_C \mathcal{E} &\implies r' \notin RID_C \mathcal{E} \implies \mathcal{R}_C (id(r := r', r' := r)) id \mathcal{E} = \mathcal{E} \\
l \notin LID_C \mathcal{E} &\implies l' \notin LID_C \mathcal{E} \implies \mathcal{R}_C id (id(l := l', l' := l)) \mathcal{E} = \mathcal{E}
\end{aligned}$$

**by** simp+

**lemma** *eliminate-swap-store-rid* [simp, intro]:

$$\begin{aligned}
r \notin RID_S \sigma &\implies r' \notin RID_S \sigma \implies \mathcal{R}_S (id(r := r', r' := r)) id \sigma = \sigma \\
\text{by (rule } \mathcal{R}_S I) & \text{ (auto simp add: swap-bij RID}_S\text{-def domIff ranI)}
\end{aligned}$$

**lemma** *eliminate-swap-store-lid* [simp, intro]:

$$\begin{aligned}
l \notin LID_S \sigma &\implies l' \notin LID_S \sigma \implies \mathcal{R}_S id (id(l := l', l' := l)) \sigma = \sigma \\
\text{by (rule } \mathcal{R}_S I) & \text{ (auto simp add: swap-bij LID}_S\text{-def domIff ranI)}
\end{aligned}$$

**lemma** *eliminate-swap-global-rid* [simp, intro]:

$$r \notin RID_G s \implies r' \notin RID_G s \implies \mathcal{R}_G (id(r := r', r' := r)) id s = s$$

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

**lemma** *eliminate-swap-global-lid* [*simp*, *intro*]:

$l \notin LID_G\ s \implies l' \notin LID_G\ s \implies \mathcal{R}_G\ id\ (id(l := l', l' := l))\ s = s$

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*]:  $r \in RID_E\ (subst\ (VE\ v)\ x\ e) \implies r \notin RID_E\ e \implies r \in RID_V\ v$

using *subst-introduces-no-rids* by *fastforce*

**lemma** *lid-substE* [*dest*]:  $l \in LID_E\ (subst\ (VE\ v)\ x\ e) \implies l \notin LID_E\ e \implies l \in LID_V\ v$

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-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**  $set3\text{-val}$  ( $\mathcal{V}_V$ )

**notation**  $set3\text{-expr}$  ( $\mathcal{V}_E$ )

**abbreviation**  $rename\text{-vars}\text{-val} :: ('v \Rightarrow 'v) \Rightarrow ('r, 'l, 'v) \text{ val} \Rightarrow ('r, 'l, 'v) \text{ val}$  ( $\mathcal{RV}_V$ )

**where**

$\mathcal{RV}_V \zeta \equiv map\text{-val } id \ id \ \zeta$

**abbreviation**  $rename\text{-vars}\text{-expr} :: ('v \Rightarrow 'v) \Rightarrow ('r, 'l, 'v) \text{ expr} \Rightarrow ('r, 'l, 'v) \text{ expr}$  ( $\mathcal{RV}_E$ ) **where**

$\mathcal{RV}_E \zeta \equiv map\text{-expr } id \ id \ \zeta$

**lemma**  $var\text{-renaming}\text{-preserves}\text{-size}$ :

**fixes**

$v :: ('r, 'l, 'v) \text{ val}$  **and**

$e :: ('r, 'l, 'v) \text{ expr}$  **and**

$\alpha :: 'r \Rightarrow 'r'$  **and**

$\beta :: 'l \Rightarrow 'l'$  **and**

$\zeta :: 'v \Rightarrow 'v'$

**shows**

$size (map\text{-val } \alpha \ \beta \ \zeta \ v) = size \ v$

$size (map\text{-expr } \alpha \ \beta \ \zeta \ e) = size \ e$

**proof** –

**have**  $(\forall (\alpha :: 'r \Rightarrow 'r') (\beta :: 'l \Rightarrow 'l') (\zeta :: 'v \Rightarrow 'v')). size (map\text{-val } \alpha \ \beta \ \zeta \ v) = size \ v) \wedge$

$(\forall (\alpha :: 'r \Rightarrow 'r') (\beta :: 'l \Rightarrow 'l') (\zeta :: 'v \Rightarrow 'v')). size (map\text{-expr } \alpha \ \beta \ \zeta \ e) = size \ e)$

**by** (*induct rule: val-expr.induct*) *auto*

**thus**

$size (map\text{-val } \alpha \ \beta \ \zeta \ v) = size \ v$

$size (map\text{-expr } \alpha \ \beta \ \zeta \ e) = size \ e$

**by** *auto*

**qed**

### 4.3.2 Definition

**function**

$nat\text{-subst}_V :: ('r, 'l, nat) \text{ expr} \Rightarrow nat \Rightarrow ('r, 'l, nat) \text{ val} \Rightarrow ('r, 'l, nat) \text{ expr}$  **and**

$nat\text{-subst}_E :: ('r, 'l, nat) \text{ expr} \Rightarrow nat \Rightarrow ('r, 'l, nat) \text{ expr} \Rightarrow ('r, 'l, nat) \text{ expr}$

**where**

$nat\text{-subst}_V \ e \ x \ (CV \ const) = VE \ (CV \ const)$

|  $nat\text{-subst}_V \ e \ x \ (Var \ x') = (if \ x = x' \ then \ e \ else \ VE \ (Var \ x'))$

|  $nat\text{-subst}_V \ e \ x \ (Loc \ l) = VE \ (Loc \ l)$

|  $nat\text{-subst}_V \ e \ x \ (Rid \ r) = VE \ (Rid \ r)$

|  $nat\text{-subst}_V \ e \ x \ (Lambda \ y \ e') = VE \ ($

$if \ x = y \ then$

$Lambda \ y \ e'$

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

### 4.3.3 Proof obligations

**lemma** *nat-subst<sub>E</sub>-distr*:

**fixes**  $e :: ('r, 'l, \text{nat}) \text{ expr}$

**shows**  $\mathcal{R}_E \alpha \beta (\text{nat-subst}_E e x e') = \text{nat-subst}_E (\mathcal{R}_E \alpha \beta e) x (\mathcal{R}_E \alpha \beta e')$

**proof** –

**fix**  $v' :: ('r, 'l, \text{nat}) \text{ val}$

**have**

$(\forall \alpha \beta x e \zeta. \mathcal{R}_E \alpha \beta (\text{nat-subst}_V e x (\mathcal{RV}_V \zeta v')) = \text{nat-subst}_V (\mathcal{R}_E \alpha \beta e) x$   
 $(\mathcal{R}_V \alpha \beta (\mathcal{RV}_V \zeta v')) \wedge$

$(\forall \alpha \beta x e \zeta. \mathcal{R}_E \alpha \beta (\text{nat-subst}_E e x (\mathcal{RV}_E \zeta e')) = \text{nat-subst}_E (\mathcal{R}_E \alpha \beta e) x$   
 $(\mathcal{R}_E \alpha \beta (\mathcal{RV}_E \zeta e')))$

by (*induct rule: val-expr.induct*) (*auto simp add: expr.set-map(3) fun.map-ident*)

**hence**  $\mathcal{R}_E \alpha \beta (\text{nat-subst}_E e x (\mathcal{RV}_E \text{id } e')) = \text{nat-subst}_E (\mathcal{R}_E \alpha \beta e) x (\mathcal{R}_E \alpha \beta$   
 $\beta (\mathcal{RV}_E \text{id } e'))$  **by** *blast*

**thus** *?thesis by simp*

**qed**

**lemma** *nat-subst<sub>E</sub>-introduces-no-rids*:

**fixes**  $e' :: ('r, 'l, \text{nat}) \text{ expr}$

**shows**  $\text{RID}_E (\text{nat-subst}_E e x e') \subseteq \text{RID}_E e \cup \text{RID}_E e'$

**proof** –

**fix**  $v' :: ('r, 'l, \text{nat}) \text{ val}$

**have**

$(\forall x e. \forall \zeta. \text{RID}_E (\text{nat-subst}_V e x (\mathcal{RV}_V \zeta v')) \subseteq \text{RID}_E e \cup \text{RID}_V (\mathcal{RV}_V \zeta v'))$   
 $\wedge$

$(\forall x e. \forall \zeta. \text{RID}_E (\text{nat-subst}_E e x (\mathcal{RV}_E \zeta e')) \subseteq \text{RID}_E e \cup \text{RID}_E (\mathcal{RV}_E \zeta e'))$

by (*induct rule: val-expr.induct*) (*auto 0 4 simp add: expr.set-map(1)*)

**hence**  $\text{RID}_E (\text{nat-subst}_E e x (\mathcal{RV}_E \text{id } e')) \subseteq \text{RID}_E e \cup \text{RID}_E (\mathcal{RV}_E \text{id } e')$  **by**  
*blast*

**thus** *?thesis by simp*

**qed**

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

**lemma** *nat-subst<sub>E</sub>-models-substitution: substitution nat-subst<sub>E</sub>*  
**by** (*simp add: nat-subst<sub>E</sub>-distr nat-subst<sub>E</sub>-introduces-no-lids nat-subst<sub>E</sub>-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* ::  $'r \Rightarrow ('r, 'l, 'v) \text{ global-state} \Rightarrow ('r, 'l, 'v) \text{ global-state} \Rightarrow \text{bool}$   
**where**

*app*:  $s r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Apply } (VE (\text{Lambda } x e)) (VE v)]) \Longrightarrow \text{revision-step } r s (s(r \mapsto (\sigma, \tau, \mathcal{E}[\text{subst } (VE v) x e])))$

*ifTrue*:  $s r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ite } (VE (CV T)) e1 e2]) \Longrightarrow \text{revision-step } r s (s(r \mapsto (\sigma, \tau, \mathcal{E}[e1])))$

*ifFalse*:  $s r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ite } (VE (CV F)) e1 e2]) \Longrightarrow \text{revision-step } r s (s(r \mapsto (\sigma, \tau, \mathcal{E}[e2])))$

*new*:  $s r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ref } (VE v)]) \Longrightarrow l \notin LID_G s \Longrightarrow \text{revision-step } r s (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE (Loc l)])))$



| *get*:  $s \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Read } (VE \ (Loc \ l))]) \implies l \in \text{dom } (\sigma;;\tau) \implies \text{revision-step } r \ s \ (s(r \mapsto (\sigma, \tau, \mathcal{E}[\text{VE } (the \ ((\sigma;;\tau) \ l))])))$   
 | *set*:  $s \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Assign } (VE \ (Loc \ l) \ (VE \ v))]) \implies l \in \text{dom } (\sigma;;\tau) \implies \text{revision-step } r \ s \ (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[\text{VE } (CV \ Unit))]))$   
  
 | *fork*:  $s \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e]) \implies r' \notin RID_G \ s \implies \text{revision-step } r \ s \ (s(r \mapsto (\sigma, \tau, \mathcal{E}[\text{VE } (Rid \ r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e)))$   
 | *join*:  $s \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE \ (Rid \ r'))]) \implies s \ r' = \text{Some } (\sigma', \tau', VE \ v) \implies \text{revision-step } r \ s \ (s(r := \text{Some } (\sigma, (\tau;;\tau'), \mathcal{E}[\text{VE } (CV \ Unit)]), r' := \text{None}))$   
 | *join<sub>ε</sub>*:  $s \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE \ (Rid \ r'))]) \implies s \ r' = \text{None} \implies \text{revision-step } r \ s \ \varepsilon$

**inductive-cases** *revision-stepE* [*elim*, *consumes 1*, *case-names app ifTrue ifFalse new get set fork join join<sub>ε</sub>*]:  
*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*:  $\bigwedge \sigma \ \tau \ \mathcal{E} \ v. \ s \ r \neq \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ref } (VE \ v)])$

**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<sub>S</sub>I(2)*)

**lemma** *only-fork-introduces-rids* [*intro*, *dest*]:

**assumes**

*step*: *revision-step r s s'* **and**

*not-fork*:  $\bigwedge \sigma \ \tau \ \mathcal{E} \ e. \ s \ r \neq \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e])$

**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*:  $\bigwedge \sigma \tau \mathcal{E} e. s r \neq \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e])$

**shows**  $r' \notin \text{RID}_G s \implies r' \notin \text{RID}_G s'$

**using** *assms* **by** *blast*

**lemma** *only-new-introduces-lids'* [*dest*]:

**assumes**

*step*: *revision-step*  $r$   $s$   $s'$  **and**

*not-new*:  $\bigwedge \sigma \tau \mathcal{E} v. s r \neq \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ref } (VE v)])$

**shows**  $l \notin \text{LID}_G s \implies l \notin \text{LID}_G s'$

**using** *assms* **by** *blast*

## 5.3 Domain subsumption

### 5.3.1 Definitions

**definition** *domains-subsume* ::  $('r, 'l, 'v)$  *local-state*  $\Rightarrow$  *bool* ( $\mathcal{S}$ ) **where**

$\mathcal{S} \text{ } ls = (\text{LID}_L \text{ } ls \subseteq \text{doms } ls)$

**definition** *domains-subsume-globally* ::  $('r, 'l, 'v)$  *global-state*  $\Rightarrow$  *bool* ( $\mathcal{S}_G$ ) **where**

$\mathcal{S}_G s = (\forall r \text{ } ls. s r = \text{Some } ls \longrightarrow \mathcal{S} \text{ } ls)$

**lemma** *domains-subsume-globallyI* [*intro*]:

$(\bigwedge r \sigma \tau e. s r = \text{Some } (\sigma, \tau, e) \implies \mathcal{S} (\sigma, \tau, e)) \implies \text{domains-subsume-globally } s$

**using** *domains-subsume-globally-def* **by** *auto*

**definition** *subsumes-accessible* ::  $'r \Rightarrow 'r \Rightarrow ('r, 'l, 'v)$  *global-state*  $\Rightarrow$  *bool* ( $\mathcal{A}$ ) **where**

$\mathcal{A} r_1 r_2 s = (r_2 \in \text{RID}_L (\text{the } (s r_1)) \longrightarrow (\text{LID}_S ((\text{the } (s r_2))_\sigma) \subseteq \text{doms } (\text{the } (s r_1))))$

**lemma** *subsumes-accessibleI* [*intro*]:

$(r_2 \in \text{RID}_L (\text{the } (s r_1)) \implies \text{LID}_S ((\text{the } (s r_2))_\sigma) \subseteq \text{doms } (\text{the } (s r_1))) \implies \mathcal{A} r_1 r_2 s$

**using** *subsumes-accessible-def* **by** *auto*

**definition** *subsumes-accessible-globally* ::  $('r, 'l, 'v)$  *global-state*  $\Rightarrow$  *bool* ( $\mathcal{A}_G$ ) **where**

$\mathcal{A}_G s = (\forall r_1 r_2. r_1 \in \text{dom } s \longrightarrow r_2 \in \text{dom } s \longrightarrow \mathcal{A} r_1 r_2 s)$

**lemma** *subsumes-accessible-globallyI* [*intro*]:

$(\bigwedge r_1 \sigma_1 \tau_1 e_1 r_2 \sigma_2 \tau_2 e_2. s r_1 = \text{Some } (\sigma_1, \tau_1, e_1) \implies s r_2 = \text{Some } (\sigma_2, \tau_2, e_2) \implies \mathcal{A} r_1 r_2 s) \implies \mathcal{A}_G s$

**using** *subsumes-accessible-globally-def* **by** *auto*

### 5.3.2 The theorem

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

**assumes**

```

     $\mathcal{S}_G$ -s:  $\mathcal{S}_G$  s and
    r-in-dom:  $r \in \text{dom } s$ 
  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 ...  $\subseteq \text{doms}$  (the (s r))
        by (metis  $\mathcal{S}_G$ -s domains-subsume-def domains-subsume-globally-def local.app(2) option.sel)
        also have ... =  $\text{doms}$  (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 ...  $\subseteq \text{doms}$  (the (s r))
        by (metis  $\mathcal{S}_G$ -s domains-subsume-def domains-subsume-globally-def ifTrue(2) option.sel)
        also have ... =  $\text{doms}$  (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:  $\langle r = r' \rangle$  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:  $\langle r = r' \rangle$  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:  $\langle r = r' \rangle$  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:  $\langle r = r' \rangle$  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**  $\dots \subseteq doms (the (s r))$   
**by** (*metis*  $\mathcal{S}_G$ -*s domains-subsume-def domains-subsume-globally-def fork(2)*  
*option.sel*)  
**also have**  $\dots = doms (the (s' r))$  **using** *fork* **by** *auto*  
**finally have**  $\mathcal{S} (the (s' r))$  **by** (*simp add: domains-subsume-def*)  
**thus** *?thesis* **by** (*simp add:  $\langle r = r' \rangle s'-r$* )  
**next**  
**assume**  $r'' = r'$   
**have**  $LID_L (the (s' r'')) \subseteq LID_L (the (s r))$  **using** *fork(1-2)* **by** *auto*  
**also have**  $\dots \subseteq doms (the (s r))$   
**by** (*metis*  $\mathcal{S}_G$ -*s domains-subsume-def domains-subsume-globally-def fork(2)*  
*option.sel*)  
**also have**  $\dots = dom \sigma' \cup dom \tau'$  **using** *fork* **by** *simp*  
**also have**  $\dots = dom (\sigma';;\tau')$  **by** (*simp add: dom-combination-dom-union*)  
**also have**  $\dots = doms (the (s' r''))$  **using** *fork* **by** *simp*  
**finally have**  $\mathcal{S} (the (s' r''))$  **by** (*simp add: domains-subsume-def*)  
**thus** *?thesis* **by** (*simp add:  $\langle r'' = r' \rangle s'-r$* )  
**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**  $\dots \subseteq doms (the (s r)) \cup LID_S \tau''$   
**by** (*metis Un-mono*  $\mathcal{S}_G$ -*s domains-subsume-def domains-subsume-globally-def*  
*join(2) option.sel subset-refl*)  
**also have**  $\dots \subseteq doms (the (s r)) \cup LID_L (the (s r''))$  **using** *join(3)* **by** *auto*  
**also have**  $\dots \subseteq doms (the (s r)) \cup doms (the (s r''))$   
**by** (*metis (no-types, lifting) Un-absorb*  $\mathcal{S}_G$ -*s domains-subsume-def do-*  
*mains-subsume-globally-def join(3) option.sel sup.orderI sup-mono*)  
**also have**  $\dots = dom \sigma' \cup dom \tau' \cup dom \sigma'' \cup dom \tau''$  **using** *join* **by** *auto*  
**also have**  $\dots \subseteq dom \sigma' \cup dom \tau' \cup LID_S \sigma'' \cup dom \tau''$  **by** *auto*  
**also have**  $\dots \subseteq dom \sigma' \cup dom \tau' \cup dom \sigma' \cup dom \tau' \cup dom \tau''$   
**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**  $\dots = dom \sigma' \cup dom \tau' \cup dom \tau''$  **by** *auto*  
**also have**  $\dots = dom \sigma' \cup dom (\tau';;\tau'')$  **by** (*auto simp add: dom-combination-dom-union*)  
**also have**  $\dots = doms (the (s' r))$  **using** *join* **by** *auto*  
**finally have**  $\mathcal{S} (the (s' r))$  **by** (*simp add: domains-subsume-def*)  
**thus** *?thesis* **using**  $\langle r' = r \rangle s'-r$  **by** *auto*  
**next**  
**case** *join $_{\varepsilon}$*

```

    then show ?thesis using s'-r by blast
  qed
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
     $s'-r_1: s' r_1 = \text{Some } (\sigma_1, \tau_1, e_1)$  and
     $s'-r_2: s' r_2 = \text{Some } (\sigma_2, \tau_2, e_2)$ 
  show  $\mathcal{A} r_1 r_2 s'$ 
  proof (cases  $r_1 = r_2$ )
    case True
      then show ?thesis using  $\mathcal{S}_G\text{-imp-}\mathcal{A}\text{-refl } \langle \mathcal{S}_G s' \rangle s'-r_1$  by blast
    next
      case  $r_1\text{-neq-}r_2: \text{False}$ 
        have  $r_1\text{-nor-}r_2\text{-updated-implies-thesis}: s' r_1 = s r_1 \implies s' r_2 = s r_2 \implies$ 
        ?thesis
        proof -
          assume  $r_1\text{-unchanged}: s' r_1 = s r_1$  and  $r_2\text{-unchanged}: s' r_2 = s r_2$ 
          have  $\mathcal{A} r_1 r_2 s$ 
            by (metis  $\mathcal{A}_G\text{-s domIff option.discI } r_1\text{-unchanged } r_2\text{-unchanged } s'-r_1 s'-r_2$ 
            subsumes-accessible-globally-def)
          show ?thesis using  $\langle \mathcal{A} r_1 r_2 s \rangle r_1\text{-unchanged } r_2\text{-unchanged subsumes-accessible-def}$ 
        by auto
        qed
      case  $r_1\text{-or-}r_2\text{-updated-implies-thesis}: s' r_1 \neq s r_1 \vee s' r_2 \neq s r_2 \implies$  ?thesis
      proof -
        assume  $r_1\text{-or-}r_2\text{-updated}: s' r_1 \neq s r_1 \vee s' r_2 \neq s r_2$ 
        show ?thesis
        proof (use step in  $\langle \text{cases rule: revision-stepE} \rangle$ )
          case app
            have  $r_1 = r \vee r_2 = r$  by (metis fun-upd-other app(I)  $r_1\text{-or-}r_2\text{-updated}$ )
            then show ?thesis
            proof (rule disjE)
              assume  $r_1\text{-eq-}r: r_1 = r$ 
              show  $\mathcal{A} r_1 r_2 s'$ 
              proof (rule subsumes-accessibleI)
                assume  $r_2\text{-in-}s'-r_1: r_2 \in \text{RID}_L (\text{the } (s' r_1))$ 
                have  $\text{LID}_S ((\text{the } (s' r_2))_\sigma) \subseteq \text{LID}_S ((\text{the } (s r_2))_\sigma)$  using app by auto
                also have  $\dots \subseteq \text{doms } (\text{the } (s r_1))$ 
                proof -
                  have  $r_2\text{-in-}s-r_1: r_2 \in \text{RID}_L (\text{the } (s r_1))$  using app  $r_2\text{-in-}s'-r_1 r_1\text{-eq-}r$ 
                by auto
                have  $\mathcal{A} r_1 r_2 s$ 
                  by (metis  $\mathcal{A}_G\text{-s domI fun-upd-other app } r_1\text{-eq-}r s'-r_2$  subsumes-accessible-globally-def)
                show ?thesis using  $\langle \mathcal{A} r_1 r_2 s \rangle r_2\text{-in-}s-r_1$  subsumes-accessible-def by blast
              qed
            qed
          qed
        qed
      case  $r_2\text{-in-}s-r_1: r_2 \in \text{RID}_L (\text{the } (s r_1))$  using app  $r_2\text{-in-}s'-r_1 r_1\text{-eq-}r$ 
    by auto
  qed
  have  $\mathcal{A} r_1 r_2 s$ 
    by (metis  $\mathcal{A}_G\text{-s domI fun-upd-other app } r_1\text{-eq-}r s'-r_2$  subsumes-accessible-globally-def)
  show ?thesis using  $\langle \mathcal{A} r_1 r_2 s \rangle r_2\text{-in-}s-r_1$  subsumes-accessible-def by blast

```

```

    qed
    also have ...  $\subseteq$  doms (the (s' r1)) using app by auto
    finally show LIDS (the (s' r2)σ)  $\subseteq$  doms (the (s' r1)) by simp
  qed
next
  assume r2-eq-r: r2 = r
  show  $\mathcal{A}$  r1 r2 s'
  proof (rule subsumes-accessibleI)
    assume r2-in-s'-r1: r2  $\in$  RIDL (the (s' r1))
    have LIDS (the (s' r2)σ) = LIDS (the (s r2)σ) using app by auto
    also have ...  $\subseteq$  doms (the (s r1))
    proof -
      have r2-in-s-r1: r2  $\in$  RIDL (the (s r1)) using app(1) r1-neq-r2 r2-eq-r
    r2-in-s'-r1 by auto
    have  $\mathcal{A}$  r1 r2 s
    by (metis (no-types, lifting)  $\mathcal{A}_G$ -s domIff fun-upd-other app option.discI
    r2-eq-r s'-r1 subsumes-accessible-globally-def)
    show ?thesis using ( $\mathcal{A}$  r1 r2 s) r2-in-s-r1 subsumes-accessible-def by
  blast

  qed
  also have ...  $\subseteq$  doms (the (s' r1)) by (simp add: app)
  finally show LIDS (the (s' r2)σ)  $\subseteq$  doms (the (s' r1)) by simp
  qed
qed
next
  case ifTrue
  have r1 = r  $\vee$  r2 = r by (metis fun-upd-other ifTrue(1) r1-or-r2-updated)
  then show ?thesis
  proof (rule disjE)
    assume r1-eq-r: r1 = r
    show  $\mathcal{A}$  r1 r2 s'
    proof (rule subsumes-accessibleI)
      assume r2-in-s'-r1: r2  $\in$  RIDL (the (s' r1))
      have LIDS ((the (s' r2))σ)  $\subseteq$  LIDS ((the (s r2))σ) using ifTrue by
    auto

    also have ...  $\subseteq$  doms (the (s r1))
    proof -
      have r2-in-s-r1: r2  $\in$  RIDL (the (s r1)) using ifTrue r2-in-s'-r1
    r1-eq-r by auto
    have  $\mathcal{A}$  r1 r2 s
    by (metis  $\mathcal{A}_G$ -s domI fun-upd-other ifTrue r1-eq-r s'-r2 sub-
    sumes-accessible-globally-def)
    show ?thesis using ( $\mathcal{A}$  r1 r2 s) r2-in-s-r1 subsumes-accessible-def by
  blast

  qed
  also have ...  $\subseteq$  doms (the (s' r1)) using ifTrue by auto
  finally show LIDS (the (s' r2)σ)  $\subseteq$  doms (the (s' r1)) by simp
  qed
next

```

```

assume  $r_2\text{-eq-}r$ :  $r_2 = r$ 
show  $\mathcal{A} r_1 r_2 s'$ 
proof (rule subsumes-accessibleI)
  assume  $r_2\text{-in-}s'\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$ 
  have  $\text{LID}_S (\text{the } (s' r_2)_\sigma) = \text{LID}_S (\text{the } (s r_2)_\sigma)$  using ifTrue by auto
  also have  $\dots \subseteq \text{doms } (\text{the } (s r_1))$ 
  proof –
    have  $r_2\text{-in-}s\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s r_1))$  using ifTrue(1)  $r_1\text{-neq-}r_2$ 
 $r_2\text{-eq-}r$   $r_2\text{-in-}s'\text{-}r_1$  by auto
    have  $\mathcal{A} r_1 r_2 s$ 
      by (metis (no-types, lifting)  $\mathcal{A}_G\text{-}s$  domIff fun-upd-other ifTrue
option.discI  $r_2\text{-eq-}r$   $s'\text{-}r_1$  subsumes-accessible-globally-def)
    show ?thesis using ( $\mathcal{A} r_1 r_2 s$ )  $r_2\text{-in-}s\text{-}r_1$  subsumes-accessible-def by
blast

  qed
  also have  $\dots \subseteq \text{doms } (\text{the } (s' r_1))$  by (simp add: ifTrue)
  finally show  $\text{LID}_S (\text{the } (s' r_2)_\sigma) \subseteq \text{doms } (\text{the } (s' r_1))$  by simp
qed
qed
next
case ifFalse
have  $r_1 = r \vee r_2 = r$  by (metis fun-upd-other ifFalse(1)  $r_1\text{-or-}r_2\text{-updated}$ )
then show ?thesis
proof (rule disjE)
  assume  $r_1\text{-eq-}r$ :  $r_1 = r$ 
  show  $\mathcal{A} r_1 r_2 s'$ 
  proof (rule subsumes-accessibleI)
    assume  $r_2\text{-in-}s'\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$ 
    have  $\text{LID}_S ((\text{the } (s' r_2))_\sigma) \subseteq \text{LID}_S ((\text{the } (s r_2))_\sigma)$  using ifFalse by
auto

    also have  $\dots \subseteq \text{doms } (\text{the } (s r_1))$ 
    proof –
      have  $r_2\text{-in-}s\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s r_1))$  using ifFalse  $r_2\text{-in-}s'\text{-}r_1$ 
 $r_1\text{-eq-}r$  by auto
      have  $\mathcal{A} r_1 r_2 s$ 
        by (metis  $\mathcal{A}_G\text{-}s$  domI fun-upd-other ifFalse  $r_1\text{-eq-}r$   $s'\text{-}r_2$  sub-
sumes-accessible-globally-def)
      show ?thesis using ( $\mathcal{A} r_1 r_2 s$ )  $r_2\text{-in-}s\text{-}r_1$  subsumes-accessible-def by
blast

    qed
    also have  $\dots \subseteq \text{doms } (\text{the } (s' r_1))$  using ifFalse by auto
    finally show  $\text{LID}_S (\text{the } (s' r_2)_\sigma) \subseteq \text{doms } (\text{the } (s' r_1))$  by simp
qed
next
assume  $r_2\text{-eq-}r$ :  $r_2 = r$ 
show  $\mathcal{A} r_1 r_2 s'$ 
proof (rule subsumes-accessibleI)
  assume  $r_2\text{-in-}s'\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$ 
  have  $\text{LID}_S (\text{the } (s' r_2)_\sigma) = \text{LID}_S (\text{the } (s r_2)_\sigma)$  using ifFalse by auto

```



```

    also have ...  $\subseteq$  doms (the (s r1))
  proof -
    have r2-in-s-r1: r2  $\in$  RIDL (the (s r1)) using ifFalse(1) r1-neq-r2
  r2-eq-r r2-in-s'-r1 by auto
    have  $\mathcal{A}$  r1 r2 s
      by (metis (no-types, lifting)  $\mathcal{A}_G$ -s domIff fun-upd-other ifFalse
option.discI r2-eq-r s'-r1 subsumes-accessible-globally-def)
    show ?thesis using ( $\mathcal{A}$  r1 r2 s) r2-in-s-r1 subsumes-accessible-def by
blast

  qed
  also have ...  $\subseteq$  doms (the (s' r1)) by (simp add: ifFalse)
  finally show LIDS (the (s' r2) $\sigma$ )  $\subseteq$  doms (the (s' r1)) by simp
  qed
  qed
next
case new
have r1 = r  $\vee$  r2 = r by (metis fun-upd-other new(1) r1-or-r2-updated)
then show ?thesis
proof (rule disjE)
  assume r1-eq-r: r1 = r
  show  $\mathcal{A}$  r1 r2 s'
  proof (rule subsumes-accessibleI)
    assume r2-in-s'-r1: r2  $\in$  RIDL (the (s' r1))
    have LIDS ((the (s' r2)) $\sigma$ )  $\subseteq$  LIDS ((the (s r2)) $\sigma$ ) using new by auto
    also have ...  $\subseteq$  doms (the (s r1))
    proof -
      have r2-in-s-r1: r2  $\in$  RIDL (the (s r1)) using new r2-in-s'-r1 r1-eq-r
    by auto
      have  $\mathcal{A}$  r1 r2 s
        by (metis  $\mathcal{A}_G$ -s domI fun-upd-other new(1-2) r1-eq-r s'-r2
subsumes-accessible-globally-def)
      show ?thesis using ( $\mathcal{A}$  r1 r2 s) r2-in-s-r1 subsumes-accessible-def by
blast

    qed
    also have ...  $\subseteq$  doms (the (s' r1)) using new by auto
    finally show LIDS (the (s' r2) $\sigma$ )  $\subseteq$  doms (the (s' r1)) by simp
    qed
  next
  assume r2-eq-r: r2 = r
  show  $\mathcal{A}$  r1 r2 s'
  proof (rule subsumes-accessibleI)
    assume r2-in-s'-r1: r2  $\in$  RIDL (the (s' r1))
    have LIDS (the (s' r2) $\sigma$ ) = LIDS (the (s r2) $\sigma$ ) using new by auto
    also have ...  $\subseteq$  doms (the (s r1))
    proof -
      have r2-in-s-r1: r2  $\in$  RIDL (the (s r1)) using new(1) r1-neq-r2
  r2-eq-r r2-in-s'-r1 by auto
      have  $\mathcal{A}$  r1 r2 s
        by (metis (no-types, lifting)  $\mathcal{A}_G$ -s domIff fun-upd-other new(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 ...  $\subseteq$  doms (the (s' r1)) by (auto simp add: new)
  finally show LID_S (the (s' r2) $_{\sigma}$ )  $\subseteq$  doms (the (s' r1)) by simp
  qed
  qed
next
case get
have r1 = r  $\vee$  r2 = r by (metis fun-upd-other get(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  $\in$  RID_L (the (s' r1))
    have LID_S ((the (s' r2)) $_{\sigma}$ )  $\subseteq$  LID_S ((the (s r2)) $_{\sigma}$ ) using get by auto
    also have ...  $\subseteq$  doms (the (s r1))
    proof -
      have r2-in-s-r1: r2  $\in$  RID_L (the (s r1)) using get r2-in-s'-r1 r1-eq-r
  apply auto
    by (meson RID_S I)
    have A r1 r2 s
      by (metis A_G-s domI fun-upd-other get(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 ...  $\subseteq$  doms (the (s' r1)) using get by auto
  finally show LID_S (the (s' r2) $_{\sigma}$ )  $\subseteq$  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  $\in$  RID_L (the (s' r1))
  have LID_S (the (s' r2) $_{\sigma}$ ) = LID_S (the (s r2) $_{\sigma}$ ) using get by auto
  also have ...  $\subseteq$  doms (the (s r1))
  proof -
    have r2-in-s-r1: r2  $\in$  RID_L (the (s r1)) using get(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 get(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 ...  $\subseteq$  doms (the (s' r1)) by (simp add: get)

```

```

      finally show  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s' r_1))$  by simp
    qed
  qed
next
case set
have  $r_1 = r \vee r_2 = r$  by (metis fun-upd-other set(1) r1-or-r2-updated)
then show ?thesis
proof (rule disjE)
  assume r1-eq-r:  $r_1 = r$ 
  show  $\mathcal{A} r_1 r_2 s'$ 
  proof (rule subsumes-accessibleI)
    assume r2-in-s'-r1:  $r_2 \in RID_L (the (s' r_1))$ 
    have  $LID_S ((the (s' r_2))_\sigma) \subseteq LID_S ((the (s r_2))_\sigma)$  using set by auto
    also have  $\dots \subseteq doms (the (s r_1))$ 
    proof -
      have r2-in-s-r1:  $r_2 \in RID_L (the (s r_1))$  using set r2-in-s'-r1 r1-eq-r
    by auto
    have  $\mathcal{A} r_1 r_2 s$ 
      by (metis  $\mathcal{A}_G$ -s domI fun-upd-other set(1-2) r1-eq-r s'-r2
subsumes-accessible-globally-def)
    show ?thesis using  $(\mathcal{A} r_1 r_2 s)$  r2-in-s-r1 subsumes-accessible-def by
blast

    qed
  also have  $\dots \subseteq doms (the (s' r_1))$  using set 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:  $r_2 = r$ 
show  $\mathcal{A} r_1 r_2 s'$ 
proof (rule subsumes-accessibleI)
  assume r2-in-s'-r1:  $r_2 \in RID_L (the (s' r_1))$ 
  have  $LID_S (the (s' r_2)_\sigma) = LID_S (the (s r_2)_\sigma)$  using set by auto
  also have  $\dots \subseteq doms (the (s r_1))$ 
  proof -
    have r2-in-s-r1:  $r_2 \in RID_L (the (s r_1))$  using set(1) r1-neq-r2 r2-eq-r
  r2-in-s'-r1 by auto
  have  $\mathcal{A} r_1 r_2 s$ 
    by (metis (no-types, lifting)  $\mathcal{A}_G$ -s domIff fun-upd-other set(1-2)
option.discI r2-eq-r s'-r1 subsumes-accessible-globally-def)
  show ?thesis using  $(\mathcal{A} r_1 r_2 s)$  r2-in-s-r1 subsumes-accessible-def by
blast

    qed
  also have  $\dots \subseteq doms (the (s' r_1))$  by (auto simp add: set)
  finally show  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s' r_1))$  by simp
  qed
qed
next
case (fork  $\sigma \tau \mathcal{E} e r'$ )
have s'-r:  $s' r = Some (\sigma, \tau, \mathcal{E} [VE (Rid r')])$  using fork by auto

```

**have**  $s'-r'$ :  $s' r' = \text{Some } (\sigma;;\tau, \varepsilon, e)$   
**by** (*simp add: local.fork(1)*)  
**have** *case1*:  $r_1 = r \implies r_2 \neq r \implies r_2 \neq r' \implies ?thesis$   
**proof** (*rule subsumes-accessibleI*)  
**assume**  $r_1 = r \ r_2 \neq r \ r_2 \neq r'$   
**assume**  $r_2\text{-in-}s'\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$   
**have**  $\text{LID}_S ((\text{the } (s' r_2))_\sigma) \subseteq \text{LID}_S ((\text{the } (s r_2))_\sigma)$  **using** *fork(1-2)* **by**  
(*simp add:  $\langle r_2 \neq r' \rangle$* )  
**also have**  $\dots \subseteq \text{doms } (\text{the } (s r_1))$   
**proof** –  
**have**  $r_2\text{-in-}s\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s r_1))$  **using** *fork*  $\langle r_1 = r \rangle \langle r_2 \neq r' \rangle$   
 $r_2\text{-in-}s'\text{-}r_1$  *s'-r* **by** *auto*  
**have**  $\mathcal{A} \ r_1 \ r_2 \ s$   
**by** (*metis (no-types, lifting)  $\mathcal{A}_{G-s} \langle r_1 = r \rangle \langle r_2 \neq r' \rangle \text{domIff fun-upd-other}$* )  
*fork(1-2)* *option.discI s'-r\_2 subsumes-accessible-globally-def*)  
**show** *?thesis* **using**  $\langle \mathcal{A} \ r_1 \ r_2 \ s \rangle$   $r_2\text{-in-}s\text{-}r_1$  *subsumes-accessible-def* **by**  
*blast*  
**qed**  
**also have**  $\dots \subseteq \text{doms } (\text{the } (s' r_1))$  **by** (*simp add:  $\langle r_1 = r \rangle$  fork(2) s'-r*)  
**finally show**  $\text{LID}_S (\text{the } (s' r_2)_\sigma) \subseteq \text{doms } (\text{the } (s' r_1))$  **by** *simp*  
**qed**  
**have** *case2*:  $r_1 \neq r \implies r_1 \neq r' \implies r_2 = r \implies ?thesis$   
**proof** (*rule subsumes-accessibleI*)  
**assume**  $r_1 \neq r \ r_1 \neq r' \ r_2 = r$   
**assume**  $r_2\text{-in-}s'\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$   
**have**  $\text{LID}_S ((\text{the } (s' r_2))_\sigma) \subseteq \text{LID}_S ((\text{the } (s r_2))_\sigma)$   
**using**  $\langle r_1 \neq r' \rangle \langle r_1 \neq r \rangle$  *fork*  $r_2\text{-in-}s'\text{-}r_1$  *s'-r\_1* **by** *auto*  
**also have**  $\dots \subseteq \text{doms } (\text{the } (s r_1))$   
**proof** –  
**have**  $r_2\text{-in-}s\text{-}r_1$ :  $r_2 \in \text{RID}_L (\text{the } (s r_1))$  **using**  $\langle r_1 \neq r' \rangle \langle r_1 \neq r \rangle$  *fork(1)*  
 $r_2\text{-in-}s'\text{-}r_1$  **by** *auto*  
**have**  $\mathcal{A} \ r_1 \ r_2 \ s$   
**by** (*metis (no-types, lifting)  $\mathcal{A}_{G-s} \langle r_1 \neq r' \rangle \langle r_2 = r \rangle \text{domIff fun-upd-other}$* )  
*fork(1-2)* *option.discI s'-r\_1 subsumes-accessible-globally-def*)  
**show** *?thesis* **using**  $\langle \mathcal{A} \ r_1 \ r_2 \ s \rangle$   $r_2\text{-in-}s\text{-}r_1$  *subsumes-accessible-def* **by**  
*auto*  
**qed**  
**also have**  $\dots \subseteq \text{doms } (\text{the } (s' r_1))$  **by** (*simp add:  $\langle r_1 \neq r' \rangle \langle r_1 \neq r \rangle$* )  
*fork(1)*  
**finally show**  $\text{LID}_S (\text{the } (s' r_2)_\sigma) \subseteq \text{doms } (\text{the } (s' r_1))$  **by** *simp*  
**qed**  
**have** *case3*:  $r_1 = r' \implies r_2 \neq r \implies r_2 \neq r' \implies ?thesis$   
**proof** (*rule subsumes-accessibleI*)  
**fix**  $l$   
**assume**  $r_1 = r' \ r_2 \neq r \ r_2 \neq r'$   
**assume**  $r_2 \in \text{RID}_L (\text{the } (s' r_1))$   
**hence**  $r_2 \in \text{RID}_L (\text{the } (s r))$  **using**  $\text{RID}_L I(3)$   $\langle r_1 = r' \rangle$  *fork(2)* *s'-r'* **by**  
*auto*  
**have**  $s \ r_2 = s' \ r_2$  **by** (*simp add:  $\langle r_2 \neq r' \rangle \langle r_2 \neq r \rangle$  fork(1)*)

hence  $\mathcal{A} r r_2 s$  using  $\mathcal{A}_G\text{-}s$  fork(2)  $s'\text{-}r_2$  subsumes-accessible-globally-def  
 by auto  
 hence  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s r))$   
 by (simp add:  $\langle r_2 \in RID_L (the (s r)) \rangle \langle s r_2 = s' r_2 \rangle$  subsumes-accessible-def)  
 also have  $\dots = dom \sigma \cup dom \tau$  by (simp add: fork(2))  
 also have  $\dots = dom (\sigma;;\tau)$  by (simp add: dom-combination-dom-union)  
 also have  $\dots = doms (the (s' r'))$  by (simp add:  $s'\text{-}r'$ )  
 finally show  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s' r_1))$  using  $\langle r_1 = r' \rangle$   
 by blast  
 qed  
 have case4:  $r_1 \neq r \implies r_1 \neq r' \implies r_2 = r' \implies ?thesis$   
 proof –  
 assume  $r_1 \neq r r_1 \neq r' r_2 = r'$   
 have  $r_2 \notin RID_L (the (s r_1))$  using  $\langle r_1 \neq r' \rangle \langle r_1 \neq r \rangle \langle r_2 = r' \rangle$  fork(1,3)  
 $s'\text{-}r_1$  by auto  
 hence  $r_2 \notin RID_L (the (s' r_1))$  by (simp add:  $\langle r_1 \neq r' \rangle \langle r_1 \neq r \rangle$  fork(1))  
 thus ?thesis by blast  
 qed  
 have case5:  $r_1 = r \implies r_2 = r' \implies ?thesis$   
 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:  $\langle r_2 = r' \rangle s'\text{-}r'$ )  
 also have  $\dots \subseteq LID_S \sigma \cup LID_S \tau$  by auto  
 also have  $\dots \subseteq LID_L (the (s' r_1))$  by (simp add:  $\langle r_1 = r \rangle s'\text{-}r$ )  
 also have  $\dots \subseteq doms (the (s' r_1))$   
 by (metis  $\langle \mathcal{S}_G s' \rangle \langle r_1 = r \rangle$  domains-subsume-def domains-subsume-globally-def  
 option.sel  $s'\text{-}r$ )  
 finally show  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s' r_1))$  by simp  
 qed  
 have case6:  $r_1 = r' \implies r_2 = r \implies ?thesis$   
 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:  $s'\text{-}r_2$   
 subsetI)  
 also have  $\dots \subseteq doms (the (s' r_2))$   
 using  $\langle \mathcal{S}_G s' \rangle$  domains-subsume-def domains-subsume-globally-def  $s'\text{-}r_2$   
 by auto  
 also have  $\dots = dom \sigma \cup dom \tau$  by (simp add:  $\langle r_2 = r \rangle s'\text{-}r$ )  
 also have  $\dots = dom (\sigma;;\tau)$  by (simp add: dom-combination-dom-union)  
 finally show  $LID_S (the (s' r_2)_\sigma) \subseteq doms (the (s' r_1))$   
 using  $\langle r_1 = r' \rangle s'\text{-}r'$  by auto  
 qed  
 show ?thesis using case1 case2 case3 case4 case5 case6 fork(1)  $r_1\text{-}neq\text{-}r_2$   
 $r_1\text{-}nor\text{-}r_2\text{-}updated\text{-}implies\text{-}thesis$  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))  
 $r_1\text{-}or\text{-}r_2\text{-}updated$   $s'\text{-}r_1$   $s'\text{-}r_2$   
 then show ?thesis

```

proof (rule disjE)
  assume  $r_1 = r$ 
  show  $\mathcal{A} \ r_1 \ r_2 \ s'$ 
  proof (rule subsumes-accessibleI)
    assume  $r_2\text{-in-}s'\text{-}r_1: r_2 \in RID_L \ (the \ (s' \ r_1))$ 
    show  $LID_S \ (the \ (s' \ r_2)_\sigma) \subseteq doms \ (the \ (s' \ r_1))$ 
    proof (cases  $r_2 \in RID_S \ \tau'$ )
      case  $r_2\text{-in-}\tau': True$ 
      have  $LID_S \ (the \ (s' \ r_2)_\sigma) = LID_S \ (the \ (s \ r_2)_\sigma)$ 
      by (metis  $\langle r_1 = r \rangle$  fun-upd-def join(1) option.distinct(1)  $r_1\text{-neq-}r_2$ 
 $s'\text{-}r_2$ )
      also have  $\dots \subseteq doms \ (the \ (s \ r'))$ 
      proof –
      have  $r_2\text{-in-}s\text{-}r': r_2 \in RID_L \ (the \ (s \ r'))$  by (simp add: join(3)
 $r_2\text{-in-}\tau'$ )
      have  $\mathcal{A} \ r' \ r_2 \ s$ 
      by (metis  $\mathcal{A}_{G\text{-}s} \ \langle r_1 = r \rangle$  domI fun-upd-def join(1) join(3)  $r_1\text{-neq-}r_2$ 
 $s'\text{-}r_2$  subsumes-accessible-globally-def)
      show ?thesis using  $\langle \mathcal{A} \ r' \ r_2 \ s \rangle$   $r_2\text{-in-}s\text{-}r'$  subsumes-accessible-def
by blast
      qed
      also have  $\dots = dom \ \sigma' \cup dom \ \tau'$  by (simp add: join(3))
      also have  $\dots \subseteq LID_S \ \sigma' \cup dom \ \tau'$  by auto
      also have  $\dots \subseteq dom \ \sigma \cup dom \ \tau \cup dom \ \tau'$ 
      proof –
      have  $r' \in RID_L \ (the \ (s \ r))$  by (simp add: join(2))
      have  $\mathcal{A} \ r \ r' \ s$  using  $\mathcal{A}_{G\text{-}s}$  join(2–3) subsumes-accessible-globally-def
by auto
      show ?thesis using  $\langle \mathcal{A} \ r \ r' \ s \rangle$  join(2–3) subsumes-accessible-def by auto
      qed
      also have  $\dots = dom \ \sigma \cup dom \ (\tau;;\tau')$  by (auto simp add:
dom-combination-dom-union)
      also have  $\dots = doms \ (the \ (s' \ r_1))$  using join by (auto simp add:  $\langle r_1$ 
 $= r \rangle$ )
      finally show ?thesis by simp
next
      case  $r_2\text{-nin-}\tau': False$ 
      have  $LID_S \ (the \ (s' \ r_2)_\sigma) = LID_S \ (the \ (s \ r_2)_\sigma)$ 
      by (metis  $\langle r_1 = r \rangle$  fun-upd-def join(1) option.distinct(1)  $r_1\text{-neq-}r_2$ 
 $s'\text{-}r_2$ )
      also have  $\dots \subseteq doms \ (the \ (s \ r_1))$ 
      proof –
      have  $r_2\text{-in-}s\text{-}r_1: r_2 \in RID_L \ (the \ (s \ r))$ 
      proof –
      have  $RID_L \ (the \ (s' \ r_1)) = RID_S \ \sigma \cup RID_S \ (\tau;;\tau') \cup RID_C \ \mathcal{E}$ 
      by (metis (no-types, lifting) ID-distr-completion(1) ID-distr-local(2)
 $\langle r_1 = r \rangle$  expr.simps(153) fun-upd-apply local.join(1) option.discI option.sel  $s'\text{-}r_1$ 
sup-bot.right-neutral val.simps(66))

```

**hence**  $r_2 \in RID_S \sigma \cup RID_S \tau \cup RID_C \mathcal{E}$  **using**  $r_2\text{-in-}s'\text{-}r_1$   
 $r_2\text{-nin-}\tau'$  **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\text{-}s \langle r\_1 = r \rangle join(1-2)*)  
 $domIff\ fun\text{-}upd\text{-}def\ option.\text{discI}\ s'\text{-}r_2\ subsumes\text{-}accessible\text{-}globally\text{-}def$   
**show** *?thesis* **using**  $\langle \mathcal{A} r_1 r_2 s \rangle r_2\text{-in-}s\text{-}r_1\ subsumes\text{-}accessible\text{-}def$   
 $\langle r_1 = r \rangle$  **by** *blast*  
**qed**  
**also have**  $\dots = dom\ \sigma \cup dom\ \tau$  **by** (*simp add: \langle r\_1 = r \rangle join(2)*)  
**also have**  $\dots \subseteq dom\ \sigma \cup dom\ (\tau;;\tau')$  **by** (*auto simp add:*  
 $dom\text{-}combination\text{-}dom\text{-}union$ )  
**also have**  $\dots = doms\ (the\ (s'\ r_1))$  **using** *join \langle r\_1 = r \rangle* **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**  $r_2\text{-in-}s'\text{-}r_1: r_2 \in RID_L\ (the\ (s'\ r_1))$   
**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*  
 $join(1-2)\ option.\text{discI}\ option.\text{sel}\ s'\text{-}r_2$ )  
**also have**  $\dots \subseteq doms\ (the\ (s\ r_1))$   
**proof** –  
**have**  $r_2\text{-in-}s\text{-}r_1: r_2 \in RID_L\ (the\ (s\ r_1))$   
**by** (*metis \langle r\_2 = r \rangle fun-upd-apply local.join(1) option.\text{discI}\ r\_1\text{-}neq\text{-}r\_2*  
 $r_2\text{-in-}s'\text{-}r_1\ s'\text{-}r_1$ )  
**have**  $\mathcal{A} r_1 r_2 s$   
**by** (*metis (no-types, lifting) \mathcal{A}\_G\text{-}s \langle r\_2 = r \rangle domIff\ fun-upd-apply*  
 $join(1-2)\ option.\text{discI}\ s'\text{-}r_1\ subsumes\text{-}accessible\text{-}globally\text{-}def$ )  
**show** *?thesis* **using**  $\langle \mathcal{A} r_1 r_2 s \rangle r_2\text{-in-}s\text{-}r_1\ subsumes\text{-}accessible\text{-}def$  **by**  
*blast*  
**qed**  
**also have**  $\dots \subseteq doms\ (the\ (s'\ r_1))$   
**by** (*metis \langle r\_2 = r \rangle eq-refl\ fun-upd-def local.join(1) option.\text{distinct}(1)*  
 $r_1\text{-}neq\text{-}r_2\ s'\text{-}r_1$ )  
**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**  $s'\text{-}r_1$  **by** *blast*  
**qed**  
**qed**  
**show**  $\mathcal{A} r_1 r_2 s'$  **using**  $r_1\text{-}nor\text{-}r_2\text{-}updated\text{-}implies\text{-}thesis\ r_1\text{-}or\text{-}r_2\text{-}updated\text{-}implies\text{-}thesis$   
**by** *blast*  
**qed**

qed  
qed

## 5.4 Relaxed definition of the operational semantics

**inductive** *revision-step-relaxed* :: '*r* ⇒ ('*r*, '*l*, '*v*) *global-state* ⇒ ('*r*, '*l*, '*v*) *global-state* ⇒ *bool* **where**

*app*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Apply } (VE\ (\text{Lambda } x\ e))\ (VE\ v)]) \implies \text{revision-step-relaxed } r\ s$   
 $(s(r \mapsto (\sigma, \tau, \mathcal{E}[\text{subst } (VE\ v)\ x\ e])))$   
*ifTrue*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ite } (VE\ (CV\ T))\ e1\ e2]) \implies \text{revision-step-relaxed } r\ s$   
 $(s(r \mapsto (\sigma, \tau, \mathcal{E}[e1])))$   
*ifFalse*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ite } (VE\ (CV\ F))\ e1\ e2]) \implies \text{revision-step-relaxed } r\ s$   
 $(s(r \mapsto (\sigma, \tau, \mathcal{E}[e2])))$

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

*fork*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e]) \implies r' \notin \text{RID}_G\ s \implies \text{revision-step-relaxed } r\ s$   
 $(s(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (\text{Rid } r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e)))$   
*join*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE\ (\text{Rid } r'))]) \implies s\ r' = \text{Some } (\sigma', \tau', VE\ v)$   
 $\implies \text{revision-step-relaxed } r\ s\ (s(r := \text{Some } (\sigma, (\tau;;\tau'), \mathcal{E}[VE\ (CV\ \text{Unit})]), r' := \text{None}))$   
*join<sub>ε</sub>*:  $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE\ (\text{Rid } r'))]) \implies s\ r' = \text{None} \implies \text{revision-step-relaxed } r\ s\ \varepsilon$

**inductive-cases** *revision-step-relaxedE* [*elim*, *consumes 1*, *case-names app ifTrue ifFalse new get set fork join join<sub>ε</sub>*]:

*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* ([ $\rightsquigarrow$ ]) **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  $\rightsquigarrow$  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  $\rightsquigarrow^*$  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  $\rightsquigarrow^=$  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<sub>E</sub>* e = {}  $\wedge$  *RID<sub>E</sub>* 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**  $\downarrow$  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*)  $\in$   $[\rightsquigarrow]^{\rightsquigarrow n}$

**by** (*metis execution-def reachable-def rtrancl-power*)

**thus** *P s*

**proof** (*induct n arbitrary: s*)

**case** 0

**have** *initial = s* **using** *0.prem(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'*)  $\in$   $[\rightsquigarrow]^{\rightsquigarrow n}$  **and** *step*: s'  $\rightsquigarrow$  s **using** *Suc.prem(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  $s = \varepsilon(r \mapsto (\varepsilon, \varepsilon, e))$  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  $s\text{-}r'$ :  $s\ r' = \text{Some}(\sigma', \tau', e')$ 
    have  $r' = r$  using  $s\ s\text{-}r'$  prog-expr-e by (meson domI domIff fun-upd-other)
    hence  $LID_L(\sigma', \tau', e') = LID_L(\varepsilon, \varepsilon, e)$  using  $s\ s\text{-}r'$  by auto
    also have  $\dots = \{\}$  using prog-expr-e by auto
    also have  $\dots = \text{dom } \sigma' \cup \text{dom } \tau'$  using  $\langle r' = r \rangle\ s\ s\text{-}r'$  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  $s\text{-}r1$ :  $s\ r_1 = \text{Some}(\sigma_1, \tau_1, e_1)$  and  $s\text{-}r2$ :  $s\ r_2 = \text{Some}(\sigma_2, \tau_2, e_2)$ 
    have  $r_2 = r$  using  $s\ s\text{-}r2$  prog-expr-e by (meson domI domIff fun-upd-other)
    hence  $\sigma_2 = \varepsilon$  using  $s\ s\text{-}r2$  by auto
    hence  $LID_S\ \sigma_2 = \{\}$  by auto
    thus  $\mathcal{A}\ r_1\ r_2\ s$  using  $s\text{-}r2$  by auto
  qed
qed
qed (use step-preserves-SG-and-AG in auto)

corollary reachable-imp-SG: reachable  $s \implies \mathcal{S}_G\ s$ 
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  $s \implies \text{revision-step } r\ s\ s' = \text{revision-step-relaxed } r\ s\ s'$ 
proof –
  assume reach: reachable  $s$ 
  have doms-sub-local: SG s by (rule reachable-imp-SG[OF reach])
  show revision-step  $r\ s\ s' = \text{revision-step-relaxed } r\ s\ s'$ 
  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  $l \notin \bigcup \{ \text{doms } ls \mid ls. ls \in \text{ran } s \}$ 
proof
  assume  $l \in \bigcup \{ \text{doms } ls \mid ls. ls \in \text{ran } s \}$ 
  then obtain  $ls$  where  $\text{in-ran}: ls \in \text{ran } s$  and  $\text{in-doms}: l \in \text{doms } ls$  by
blast
  from  $\text{in-doms}$  have  $l \in LID_L \text{ } ls$  by (cases  $ls$ ) auto
  have  $l \in LID_G \text{ } s$ 
  proof -
    have  $ls \in \{ls. \exists r. s \text{ } r = \text{Some } ls\}$  by (metis (full-types)  $\text{in-ran}$   $\text{ran-def}$ )
    then show  $?thesis$  using  $\langle l \in LID_L \text{ } ls \rangle$  by blast
  qed
  thus  $\text{False}$  using  $\text{new}$  by auto
qed
qed (simp add:  $\text{new.hyps}(2)$ )
thus  $?thesis$  using  $\text{new.hyps}(1)$  by blast
qed (use  $\text{revision-step-relaxed.intros}$  in simp)+
next
assume  $\text{step}: \text{revision-step-relaxed } r \text{ } s \text{ } s'$ 
show  $\text{revision-step } r \text{ } s \text{ } s'$ 
proof (use  $\text{step}$  in  $\langle \text{induct rule}: \text{revision-step-relaxedE} \rangle$ )
  case (new  $\sigma \tau \mathcal{E} v l$ )
  have  $\text{revision-step } r \text{ } s \text{ } (s(r \mapsto (\sigma, \tau(l \mapsto v)), \mathcal{E} [VE (Loc \text{ } l)]))$ 
  proof (rule  $\text{revision-step.new}$ )
    show  $s \text{ } r = \text{Some } (\sigma, \tau, \mathcal{E} [Ref (VE \text{ } v)])$  by (simp add:  $\text{new.hyps}(2)$ )
    show  $l \notin LID_G \text{ } s$ 
  proof
    assume  $l \in LID_G \text{ } s$ 
    then obtain  $r' \sigma' \tau' e'$  where  $s\text{-}r': s \text{ } r' = \text{Some } (\sigma', \tau', e')$  and  $l\text{-in-local}$ :
 $l \in LID_L (\sigma', \tau', e')$  by auto
    hence  $l \in \text{dom } \sigma' \cup \text{dom } \tau'$ 
    by (metis (no-types, lifting)  $\text{domains-subsume-def}$   $\text{domains-subsume-globally-def}$ 
 $\text{doms.simps}$   $\text{doms-sub-local}$   $\text{rev-subsetD}$ )
    thus  $\text{False}$  by (meson  $s\text{-}r'$   $\text{new.hyps}(3)$   $\text{ranI}$ )
  qed
  qed
  then show  $?case$  using  $\text{new.hyps}(1)$  by blast
next
case (get  $\sigma \tau \mathcal{E} l$ )
have  $\text{revision-step } r \text{ } s \text{ } (s(r \mapsto (\sigma, \tau, \mathcal{E} [VE (the ((\sigma;;\tau) \text{ } l)]))))$ 
proof
  show  $s \text{ } r = \text{Some } (\sigma, \tau, \mathcal{E} [Read (VE (Loc \text{ } l))])$  by (simp add:  $\text{get.hyps}(2)$ )
  show  $l \in \text{dom } (\sigma;;\tau)$ 
  proof -
    have  $l \in LID_L (\sigma, \tau, \mathcal{E} [Read (VE (Loc \text{ } l))])$  by simp
    hence  $l \in \text{dom } \sigma \cup \text{dom } \tau$ 
    using  $\text{domains-subsume-def}$   $\text{domains-subsume-globally-def}$   $\text{doms-sub-local}$ 
 $\text{get.hyps}(2)$  by fastforce
    thus  $l \in \text{dom } (\sigma;;\tau)$  by (simp add:  $\text{dom-combination-dom-union}$ )
  qed
  qed

```

```

qed
then show ?case using get.hyps(1) by auto
next
  case (set  $\sigma$   $\tau$   $\mathcal{E}$   $l$   $v$ )
  have revision-step  $r$   $s$  ( $s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E} [VE (CV Unit)]))$ )
  proof
    show  $s$   $r = Some$  ( $\sigma, \tau, \mathcal{E} [Assign (VE (Loc l)) (VE v)]$ ) by (simp add:
set.hyps(2))
    show  $l \in dom$  ( $\sigma;;\tau$ )
    proof –
      have  $l \in LID_L$  ( $\sigma, \tau, \mathcal{E} [Assign (VE (Loc l)) (VE v)]$ ) by simp
      hence  $l \in dom$   $\sigma \cup dom$   $\tau$ 
      using domains-subsume-def domains-subsume-globally-def doms-sub-local
set.hyps(2) by fastforce
      thus  $l \in dom$  ( $\sigma;;\tau$ ) 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 ( $RID_V$   $v$ )  $\wedge$  finite ( $LID_V$   $v$ ))  $\wedge$  finite ( $RID_E$   $e$ )  $\wedge$  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 ( $RID_S$   $\tau$ )  $\implies$  finite ( $RID_S$  ( $\tau(l := None)$ ))
finite ( $LID_S$   $\tau$ )  $\implies$  finite ( $LID_S$  ( $\tau(l := None)$ ))
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]:  
 $finite (RID_G s) \implies finite (RID_G (s(r := None)))$   
 $finite (LID_G s) \implies finite (LID_G (s(r := None)))$   
**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**  $LID_G (s(r := None)) \subseteq LID_G s$  **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]:  
 $s r' = Some\ ls \implies finite (RID_G (s(r := None))) \implies r \neq r' \implies finite (RID_L ls)$   
 $s r' = Some\ ls \implies finite (LID_G (s(r := None))) \implies r \neq r' \implies finite (LID_L ls)$   
**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]:  
 $finite (RID_S \varepsilon)$   
 $finite (LID_S \varepsilon)$   
 $finite (RID_G \varepsilon)$   
 $finite (LID_G \varepsilon)$   
**by** (simp add: RID\_S-def LID\_S-def RID\_G-def LID\_G-def)+

**lemma** *finite-combination* [intro]:  
 $finite (RID_S \sigma) \implies finite (RID_S \tau) \implies finite (RID_S (\sigma;;\tau))$   
 $finite (LID_S \sigma) \implies finite (LID_S \tau) \implies finite (LID_S (\sigma;;\tau))$   
**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  $\sigma\ \tau\ \mathcal{E}\ r'\ \sigma'\ \tau'\ v$ )  
**hence**  $r \neq r'$  **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<sub>G</sub> s)*  
**shows**  
*finite (LID<sub>G</sub> s')*  
**proof** (use *step in* (cases rule: *revision-stepE*))  
**case** (join  $\sigma \tau \mathcal{E} r' \sigma' \tau' v$ )  
**hence**  $r \neq r'$  **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<sub>G</sub> s)*  
*finite (LID<sub>G</sub> s)*  
**proof** –  
**from** *reach* **obtain**  $e r$  **where** *exec: execution e* ( $\varepsilon(r \mapsto (\varepsilon, \varepsilon, e))$ )  $s$  **using** *reachable-def execution-def* **by** *auto*  
**hence** *prog-exp: program-expr e* **by** (*meson execution-def*)  
**obtain**  $n$  **where** *n-reachable:* ( $\varepsilon(r \mapsto (\varepsilon, \varepsilon, e)), s$ )  $\in [\rightsquigarrow] \rightsquigarrow^n$  **using** *exec* **by** (*meson execution-def rtrancl-imp-relpow*)  
**hence** *finite (RID<sub>G</sub> s)  $\wedge$  finite (LID<sub>G</sub> s)*  
**proof** (*induct n arbitrary: s*)  
**case** 0  
**hence**  $s: s = \varepsilon(r \mapsto (\varepsilon, \varepsilon, e))$  **by** *auto*  
**hence** *rid-dom: dom s = {r}* **by** *auto*  
**hence** *rid-ran:  $\bigcup (RID_L \text{ ` } \text{ran } s) = \{ \}$*  **using**  $s$  **by** (*auto simp add: prog-exp*)  
**have** *rids: RID<sub>G</sub> s = {r}* **by** (*unfold RID<sub>G</sub>-def, use rid-dom rid-ran in auto*)  
**have** *lid-ran:  $\bigcup (LID_L \text{ ` } \text{ran } s) = \{ \}$*  **using**  $s$  **by** (*auto simp add: prog-exp*)  
**hence** *lids: LID<sub>G</sub> s = { }* **by** (*unfold LID<sub>G</sub>-def, simp*)  
**thus** *?case* **using** *rids lids* **by** *simp*  
**next**  
**case** (*Suc n*)  
**then obtain**  $s'$  **where**  
*n-steps:* ( $\varepsilon(r \mapsto (\varepsilon, \varepsilon, e)), s'$ )  $\in [\rightsquigarrow] \rightsquigarrow^n$  **and**  
*step: s'  $\rightsquigarrow$  s*  
**by** (*meson relpow-Suc-E*)  
**have** *fin-rid: finite (RID<sub>G</sub> s')* **using** *Suc.hyps n-steps* **by** *blast*  
**have** *fin-lid: finite (LID<sub>G</sub> s')* **using** *Suc.hyps n-steps* **by** *blast*  
**thus** *?case* **by** (*meson RID<sub>G</sub>-finite-invariant RID<sub>L</sub>-finite-invariant fin-rid local.step valid-stepE*)  
**qed**  
**thus** *finite (RID<sub>G</sub> s) finite (LID<sub>G</sub> s)* **by** *auto*  
**qed**

**lemma** *reachable-imp-identifiers-available*:

**assumes**

*reachable* ( $s :: ('r, 'l, 'v) \text{ global-state}$ )

**shows**

*infinite* ( $UNIV :: 'r \text{ set}$ )  $\implies \exists r. r \notin RID_G s$

*infinite* ( $UNIV :: 'l \text{ set}$ )  $\implies \exists l. l \notin LID_G s$

**by** (*simp add: assms ex-new-if-finite reachable-imp-identifiers-finite*)<sup>+</sup>

### 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**  $s\text{-}s'$ :  $s \rightsquigarrow s'$  **using** *step* **by** *blast*

**have** *init*  $\rightsquigarrow^* s'$  **using** *init-s*  $s\text{-}s'$  **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*  $s \implies s \rightsquigarrow^* s' \implies \text{reachable } s'$

**proof** –

**assume** *reach*: *reachable*  $s$  **and**  $s \rightsquigarrow^* s'$

**then obtain**  $n$  **where**  $(s, s') \in [\rightsquigarrow] \rightsquigarrow^n$  **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**  $(s, s'') \in [\rightsquigarrow] \rightsquigarrow^n$   $s'' \rightsquigarrow s'$  **using** *Suc.prem*s **by** *auto*

**have** *reachable*  $s''$  **by** (*simp add: Suc.hyps*  $\langle (s, s'') \in [\rightsquigarrow] \rightsquigarrow^n \rangle$ )

**then show** *?case* **using**  $\langle s'' \rightsquigarrow s' \rangle$  *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 ( $\sigma, \tau, \mathcal{E}$  [Apply (VE (Lambda x e)) (VE v)])
  shows (revision-step r s s') = (s' = (s(r  $\mapsto$  ( $\sigma, \tau, \mathcal{E}$  [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 ( $\sigma, \tau, \mathcal{E}$  [Ite (VE (CV T)) e1 e2])
  shows (revision-step r s s') = (s' = (s(r  $\mapsto$  ( $\sigma, \tau, \mathcal{E}$  [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 ( $\sigma, \tau, \mathcal{E}$  [Ite (VE (CV F)) e1 e2])
  shows (revision-step r s s') = (s' = (s(r  $\mapsto$  ( $\sigma, \tau, \mathcal{E}$  [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**  
 $s\text{-}r: s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Ref } (VE\ v)])$   
**shows**  $(\text{revision-step } r\ s\ s') = (\exists l. l \notin LID_G\ s \wedge s' = (s(r \mapsto (\sigma, \tau(l \mapsto v)), \mathcal{E} [VE\ (Loc\ l)]))))$  **(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**  
 $s\text{-}r: s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Read } (VE\ (Loc\ l))])$   
**shows**  $(\text{revision-step } r\ s\ s') = (l \in \text{dom } (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [VE\ (the\ ((\sigma;;\tau)\ l)]))))))$  **(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**  
 $s\text{-}r: s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Assign } (VE\ (Loc\ l))\ (VE\ v)])$   
**shows**  $(\text{revision-step } r\ s\ s') = (l \in \text{dom } (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma, \tau(l \mapsto v)), \mathcal{E} [VE\ (CV\ Unit)]))))$  **(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**  
 $s\text{-}r: s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Rfork } e])$   
**shows**  $(\text{revision-step } r\ s\ s') = (\exists r'. r' \notin RID_G\ (s(r \mapsto (\sigma, \tau, \mathcal{E} [\text{Rfork } e]))) \wedge s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [VE\ (Rid\ r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e))))$  **(is ?l = ?r)**  
**proof** (rule *iffI*)  
**assume** step: ?l  
**show** ?r  
**proof** (use *step* **in** *cases rule: revision-stepE*)  
**case** (fork  $\sigma\ \tau\ \mathcal{E}\ e\ r'$ )  
**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**  
 $s\text{-}r: s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Rjoin } (VE\ (Rid\ r'))])$  **and**  
 $s\text{-}r': s\ r' = \text{Some } (\sigma', \tau', VE\ v)$   
**shows**  $(\text{revision-step } r\ s\ s') = (s' = (s(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E} [VE\ (CV\ Unit)]), r' := \text{None})))$  **(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<sub>ε</sub>-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<sub>ε</sub> s-r s-r')

## 7.2 Strong local confluence

### 7.2.1 Local determinism

**lemma** local-determinism:

**assumes**  
left: revision-step r s<sub>1</sub> s<sub>2</sub> **and**  
right: revision-step r s<sub>1</sub> s<sub>2</sub>'  
**shows** s<sub>2</sub> ≈ s<sub>2</sub>'  
**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<sub>G</sub> s<sub>1</sub> **and**  
s<sub>2</sub>': s<sub>2</sub>' = s<sub>1</sub>(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<sub>G</sub> id ?β s<sub>2</sub> = s<sub>2</sub>'  
**by** (use new side s<sub>2</sub>' 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<sub>G</sub> s<sub>1</sub> **and**  
s<sub>2</sub>': s<sub>2</sub>' = s<sub>1</sub>(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<sub>G</sub> ?α id s<sub>2</sub> = s<sub>2</sub>'  
**by** (use fork side s<sub>2</sub>' 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*:

$\exists s_3' s_3. s_3' \approx s_3 \wedge (\text{revision-step } r' s_2 s_3' \vee s_2 = s_3') \wedge (\text{revision-step } r s_2' s_3 \vee s_2' = s_3) \implies$   
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r s_2' s_3 \vee s_2' = s_3) \wedge (\text{revision-step } r' s_2 s_3' \vee s_2 = s_3')$   
**by** (*metis*  $\alpha\beta$ -*sym*)

**lemma** *SLC-commute*:

$\llbracket s_3 = s_3'; \text{revision-step } r' s_2 s_3; \text{revision-step } r s_2' s_3' \rrbracket \implies$   
 $s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$   
**using**  $\alpha\beta$ -*refl* **by** *auto*

## 7.2.3 Case join-epsilon

**lemma** *SLC-join $_\epsilon$* :

**assumes**

*s<sub>1</sub>-r*:  $s_1 r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (\text{VE } (\text{Rid } r''))])$  **and**  
*s<sub>2</sub>*:  $s_2 = \epsilon$  **and**  
*side*:  $s_1 r'' = \text{None}$  **and**  
*right*: *revision-step*  $r' s_1 s_2'$  **and**  
*neq*:  $r \neq r'$

**shows**

$\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**proof** –

**have** *right-collapsed-case*:  $s_2' = \epsilon \implies ?thesis$

**by** (*rule* *exI*[**where**  $x=\epsilon$ ], *rule* *exI*[**where**  $x=\epsilon$ ], *use* *s<sub>2</sub>* **in** *auto*)

**have** *left-step-still-available-case*:  $s_2' \neq \epsilon \implies s_2' r = s_1 r \implies s_2' r'' = \text{None} \implies ?thesis$

**by** (*rule* *exI*[**where**  $x=\epsilon$ ], *rule* *exI*[**where**  $x=\epsilon$ ]) (*use* *assms* **in** *auto*)

**show** *?thesis*

**proof** (*use* *right* **in**  $\langle \text{cases rule: revision-stepE} \rangle$ )

**case** (*join* - - - *right-joinee*)

**have** *r-unchanged-left*:  $s_2' r = s_1 r$  **using** *join* *assms* **by** *auto*

**have** *r'-unchanged-right*:  $s_2' r'' = \text{None}$  **using** *join* *assms* **by** *auto*

**have** *right-joinee*  $\neq r'$  **using** *join*(2-3) **by** *auto*

**hence** *s<sub>2</sub>'-nonempty*:  $s_2' \neq \epsilon$  **using** *assms* *join* **by** (*auto* *simp* *add*: *fun-upd-twist*)

**show** *?thesis* **by** (*rule* *left-step-still-available-case*[*OF* *s<sub>2</sub>'-nonempty* *r-unchanged-left* *r'-unchanged-right*])

**next**

**case** *join $_\epsilon$*

**show** *?thesis* **by** (*rule* *right-collapsed-case*, *use* *join $_\epsilon$* (2-3) *right* **in** *auto*)

**qed** ((*rule* *left-step-still-available-case*, *use* *side* *neq* *s<sub>1</sub>-r* *right* **in** *auto*)[1])+

**qed**

## 7.2.4 Case join

**lemma** *join-and-local-commute*:

**assumes**  
 $s_2 = s_1(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (\text{CV Unit})]), r'' := \text{None})$   
 $s_2' = s_1(r' \mapsto ls)$   
 $r \neq r'$   
 $r' \neq r''$   
 $\text{revision-step } r' s_2 (s_1(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (\text{CV Unit})]), r'' := \text{None}, r' := \text{Some } ls))$   
 $s_2' r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Rjoin } (\text{VE } (\text{Rid } r''))])$   
 $s_2' r'' = \text{Some } (\sigma', \tau', \text{VE } v)$

**shows**  
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**apply** (rule  $\text{exI}[\text{where } x=s_1(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (\text{CV Unit})]), r'' := \text{None}, r' := \text{Some } ls)]$ )

**apply** (rule  $\text{exI}[\text{where } x=s_1(r' := \text{Some } ls, r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (\text{CV Unit})]), r'' := \text{None})]$ )

**by** (rule  $\text{SLC-commute}$ , use  $\text{assms in auto}$ )

**lemma SLC-join:**

**assumes**  
 $s_{1-r}: s_1 r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (\text{VE } (\text{Rid } r''))])$  **and**  
 $s_2: s_2 = (s_1(r := \text{Some } (\sigma, (\tau;;\tau'), \mathcal{E}[\text{VE } (\text{CV Unit})]), r'' := \text{None}))$  **and**  
 $\text{side}: s_1 r'' = \text{Some } (\sigma', \tau', \text{VE } v)$  **and**  
 $\text{right}: \text{revision-step } r' s_1 s_2'$  **and**  
 $\text{neg}: r \neq r'$

**shows**  
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**proof** –

**have**  $\text{left-step}: \text{revision-step } r s_1 s_2$  **using**  $s_{1-r} s_2$  **side by auto**

**have**  $r'\text{-not-joined}: r' \neq r''$  **using**  $\text{right side by auto}$

**show**  $?thesis$

**proof** (use  $\text{right in } \langle \text{cases rule: revision-stepE} \rangle$ )

**case** (new - - - l)

**have**  $l\text{-fresh-left}: l \notin \text{LID}_G s_2$  **by** (rule  $\text{only-new-introduces-lids}'[\text{OF left-step}]$ )  
 (use  $\text{new right } s_{1-r}$  **in auto**)

**show**  $?thesis$  **by** (rule  $\text{join-and-local-commute}$ , use  $\text{assms } r'\text{-not-joined new } l\text{-fresh-left}$  **in auto**)

**next**

**case** (fork - - -  $r'''$ )

**have**  $r'\text{-unchanged-left}: s_2 r' = s_1 r'$  **using**  $\text{fork assms by auto}$

**have**  $r'''\text{-fresh-left}: r''' \notin \text{RID}_G s_2$  **using**  $\text{left-step fork}(3)$   $\text{only-fork-introduces-rids}'$   
 $s_{1-r}$  **by auto**

**have**  $r\text{-unchanged-right}: s_2' r = s_1 r$  **using**  $\text{fork assms by auto}$

**have**  $r''\text{-unchanged-right}: s_2' r'' = s_1 r''$  **using**  $\text{fork assms by auto}$

**let**  $?s_3 = s_2(r' := s_2' r', r''' := s_2' r''')$

**let**  $?s_3' = s_2'(r := s_2 r, r'' := \text{None})$

**show**  $?thesis$

**proof** (rule  $\text{exI}[\text{where } x=?s_3], \text{rule } \text{exI}[\text{where } x=?s_3], \text{rule } \text{SLC-commute}$ )

**show**  $?s_3 = ?s_3'$  **using** *fork(1) fork(3) neq r'-not-joined s<sub>1</sub>-r s<sub>2</sub>* **by** (*auto simp add: ID-distr-global-conditional*)  
**show** *revision-step r' s<sub>2</sub> ?s<sub>3</sub>* **using** *fork(1-2) r'-unchanged-left r'''-fresh-left* **by** (*auto simp add: ID-distr-global-conditional*)  
**show** *revision-step r s<sub>2</sub>' ?s<sub>3</sub>'* **using** *r''-unchanged-right r-unchanged-right s<sub>1</sub>-r s<sub>2</sub> side* **by** *auto*  
**qed**  
**next**  
**case** (*join - - - r'''*)  
**have** *r'-unchanged-left: s<sub>2</sub> r' = s<sub>1</sub> r'* **using** *join(2) neq r'-not-joined s<sub>2</sub>* **by** *auto*  
**have** *r-unchanged-right: s<sub>2</sub>' r = s<sub>1</sub> r* **using** *join(1,3) neq s<sub>1</sub>-r* **by** *auto*  
**show** *?thesis*  
**proof** (*cases r'' = r'''*)  
**case** *True*  
**have** *r'''-none-left: s<sub>2</sub> r''' = None* **by** (*simp add: True s<sub>2</sub>*)  
**have** *r''-none-right: s<sub>2</sub>' 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**  $\varepsilon = \varepsilon$  **by** (*rule refl*)  
**show** *revision-step r' s<sub>2</sub> ε* **using** *r'-unchanged-left r'''-none-left join(2)* **by** *auto*  
**show** *revision-step r s<sub>2</sub>' ε* **using** *r-unchanged-right r''-none-right s<sub>1</sub>-r* **by** *auto*  
**qed**  
**next**  
**case** *False*  
**have** *r'''-unchanged-left: s<sub>2</sub> r''' = s<sub>1</sub> r'''* **using** *False join(1,3) s<sub>2</sub> r-unchanged-right* **by** *auto*  
**have** *r''-unchanged-right': s<sub>2</sub>' r'' = s<sub>1</sub> 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<sub>3</sub>], rule exI[where x=?s<sub>3</sub>'], rule SLC-commute*)  
**show**  $?s_3 = ?s_3'$  **using** *join(1) neq r'-not-joined r-unchanged-right s<sub>1</sub>-r s<sub>2</sub> s<sub>1</sub>-r* **by** *fastforce*  
**show** *revision-step r' s<sub>2</sub> ?s<sub>3</sub>* **by** (*simp add: join r'''-unchanged-left r'-unchanged-left*)  
**show** *revision-step r s<sub>2</sub>' ?s<sub>3</sub>'* **using** *r''-unchanged-right' r-unchanged-right s<sub>1</sub>-r side s<sub>2</sub>* **by** *auto*  
**qed**  
**qed**  
**next**  
**case** *join<sub>ε</sub>*  
**show** *?thesis* **by** (*rule SLC-sym, rule SLC-join<sub>ε</sub>, use left-step neq right join<sub>ε</sub> 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**

$\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**by** (*metis (no-types, lifting) assms fun-upd-twist fun-upd-upd local-determinism*)

**lemma** *local-and-fork-commute*:

**assumes**

$s_2 = s_1(r \mapsto x)$   
 $s_2' = s_1(r' \mapsto (\sigma, \tau, \mathcal{E} [VE (Rid r'')]), r'' \mapsto (\sigma;;\tau, \varepsilon, e))$   
 $s_2 r' = \text{Some } (\sigma, \tau, \mathcal{E}[Rfork e])$   
 $r'' \notin RID_G s_2$   
*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**

$\exists s_3 s_3'. (s_3 \approx s_3') \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**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**

$s_1$ - $r$ :  $s_1 r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Apply } (VE (Lambda x e)) (VE v)])$  **and**

$s_2$ :  $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[\text{subst } (VE v) x e]))$  **and**

*right*: *revision-step*  $r'$   $s_1$   $s_2'$  **and**

*neq*:  $r \neq r'$

**shows**

$\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$

**proof** –  
**have** *left-step: revision-step*  $r\ s_1\ s_2$  **using**  $s_1\text{-}r\ s_2$  **by** *auto*  
**show** *?thesis*  
**proof** (*use right in*  $\langle$ *cases rule: revision-stepE* $\rangle$ )  
  **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:  $s_1\text{-}r\ new(\beta)$* ) +  
  **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:  $s_1\text{-}r\ fork(\beta)$* ) +  
  **show** *?thesis* **by** (*rule local-and-fork-commute*[*OF  $s_2\ fork(1)$* ]) (*use fork neq  $s_2$*   
 *$r''$ -fresh-left  $s_1\text{-}r$  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 $_\epsilon$*   
  **show** *?thesis* **by** (*rule SLC-sym, rule SLC-join $_\epsilon$ , use join $_\epsilon$  left-step right neq in*  
*auto*)  
  **qed** ((*rule local-steps-commute*[*OF  $s_2$* ], *use assms in auto*)[*1*]) +  
**qed**

**lemma** *SLC-ifTrue:*

**assumes**  
   $s_1\text{-}r: s_1\ r = \text{Some}(\sigma, \tau, \mathcal{E}[\text{Ite}(\text{VE}(\text{CV}\ T))\ e1\ e2])$  **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**  
 $\exists s_3\ s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r\ s_2'\ s_3' \vee s_2' = s_3')$   
**proof** –  
**have** *left-step: revision-step*  $r\ s_1\ s_2$  **using**  $s_1\text{-}r\ s_2$  **by** *auto*  
**show** *?thesis*  
**proof** (*use right in*  $\langle$ *cases rule: revision-stepE* $\rangle$ )  
  **case** (*new - - - l*)  
  **have** *l-fresh-left:  $l \notin LID_G\ s_2$*   
  **by** (*rule only-new-introduces-lids'*[*OF left-step*]) (*simp add:  $s_1\text{-}r\ new(\beta)$* ) +  
  **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:  $s_1\text{-}r\ fork(\beta)$* ) +  
  **show** *?thesis* **by** (*rule local-and-fork-commute*[*OF  $s_2\ fork(1)$* ]) (*use fork neq  $s_2$*   
 *$r''$ -fresh-left  $s_1\text{-}r$  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<sub>ε</sub>*  
**show** ?thesis **by** (rule *SLC-sym*, rule *SLC-join<sub>ε</sub>*, use *join<sub>ε</sub> left-step right neq* **in** *auto*)  
**qed** ((rule *local-steps-commute*[*OF s<sub>2</sub>*], use *assms* **in** *auto*)[1])+  
**qed**

**lemma** *SLC-ifFalse*:

**assumes**  
*s<sub>1-r</sub>*:  $s_1 \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ite } (VE \ (CV \ F)) \ e1 \ e2])$  **and**  
*s<sub>2</sub>*:  $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[e2]))$  **and**  
*right*: *revision-step*  $r' \ s_1 \ s_2'$  **and**  
*neq*:  $r \neq r'$   
**shows**  
 $\exists s_3 \ s_3'. \ s_3 \approx s_3' \wedge (\text{revision-step } r' \ s_2 \ s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r \ s_2' \ s_3' \vee s_2' = s_3')$   
**proof** –  
**have** *left-step*: *revision-step*  $r \ s_1 \ s_2$  **using** *s<sub>1-r</sub>* *s<sub>2</sub>* **by** *auto*  
**show** ?thesis  
**proof** (use *right* **in** (cases rule: *revision-stepE*))  
**next**  
**case** (*new* - - - - *l*)  
**have** *l-fresh-left*:  $l \notin \text{LID}_G \ s_2$   
**by** (rule *only-new-introduces-lids'*[*OF left-step*]) (simp add: *s<sub>1-r</sub>* *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*:  $r'' \notin \text{RID}_G \ s_2$   
**by** (rule *only-fork-introduces-rids'*[*OF left-step*]) (simp add: *s<sub>1-r</sub>* *fork*(3))+  
**show** ?thesis **by** (rule *local-and-fork-commute*[*OF s<sub>2</sub> fork*(1)]) (use *fork neq* *s<sub>2</sub>* *r''-fresh-left* *s<sub>1-r</sub>* **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<sub>ε</sub>*  
**show** ?thesis **by** (rule *SLC-sym*, rule *SLC-join<sub>ε</sub>*, use *join<sub>ε</sub> left-step right neq* **in** *auto*)  
**qed** ((rule *local-steps-commute*[*OF s<sub>2</sub>*], use *assms* **in** *auto*)[1])+  
**qed**

**lemma** *SLC-set*:

**assumes**  
*s<sub>1-r</sub>*:  $s_1 \ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Assign } (VE \ (Loc \ l)) \ (VE \ v)])$  **and**  
*s<sub>2</sub>*:  $s_2 = s_1(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE \ (CV \ Unit)]))$  **and**

*side:  $l \in \text{dom}(\sigma;;\tau)$  and*  
*right: revision-step  $r' s_1 s_2'$  and*  
*neg:  $r \neq r'$*   
**shows**  
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$   
**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** *(new - - - l)*  
**have** *l-fresh-left:  $l \notin \text{LID}_G s_2$*   
**by** *(rule only-new-introduces-lids'[OF left-step]) (simp add: s1-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:  $r'' \notin \text{RID}_G s_2$*   
**by** *(rule only-fork-introduces-rids'[OF left-step]) (simp add: s1-r fork(3))+*  
**show** *?thesis by (rule local-and-fork-commute[OF s2 fork(1)]) (use fork neg s2 r''-fresh-left s1-r side in auto)*  
**next**  
**case** *join*  
**show** *?thesis by (rule SLC-sym, rule SLC-join, use join left-step neg in auto)*  
**next**  
**case** *join $_\epsilon$*   
**show** *?thesis by (rule SLC-sym, rule SLC-join $_\epsilon$ , use join $_\epsilon$  left-step neg in auto)*  
**qed** *((rule local-steps-commute[OF s2], use assms in auto)[1])+*  
**qed**

**lemma** *SLC-get:*

**assumes**  
 $s_1$ - $r$ :  $s_1 r = \text{Some}(\sigma, \tau, \mathcal{E}[\text{Read}(VE(\text{Loc } l))])$  **and**  
 $s_2$ :  $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[VE(\text{the}((\sigma;;\tau) l))])$  **and**  
*side:  $l \in \text{dom}(\sigma;;\tau)$  and*  
*right: revision-step  $r' s_1 s_2'$  and*  
*neg:  $r \neq r'$*   
**shows**  
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$   
**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** *(new - - - l)*  
**have** *l-fresh-left:  $l \notin \text{LID}_G s_2$*   
**by** *(rule only-new-introduces-lids'[OF left-step]) (simp add: s1-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( $\beta$ ))+
  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 $_\epsilon$ 
  show ?thesis by (rule SLC-sym, rule SLC-join $_\epsilon$ , use join $_\epsilon$  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:  $s_1\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ref } (VE\ v)])$  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
   $\exists s_3\ s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r\ s_2'\ s_3' \vee s_2' = s_3')$ 
proof –
  have left-step: revision-step  $r\ s_1\ s_2$  using  $s_1$ -r 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<sub>L</sub>-finite-invariant reachable-imp-identifiers-finite(2)*)  
 hence  $l-l''$ :  $l \neq l''$  by (*auto simp add: s<sub>2</sub>*)  
 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  $?\beta = id(l := l'', l'' := l)$   
 have  $bij$ - $\beta$ :  $bij ?\beta$  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<sub>2</sub> ?s<sub>3</sub>*  
   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<sub>2</sub>' ?s<sub>3</sub>'*  
   using *l''-fresh-right' new(1) neq s<sub>1</sub>-r* by *auto*  
 have *equiv: ?s<sub>3</sub>  $\approx$  ?s<sub>3</sub>'*  
 proof (*rule eq-statesI[of id ?\beta]*)  
   show  $\mathcal{R}_G id ?\beta ?s_3 = ?s_3'$   
   proof –  
     have  $s_1$ :  $\mathcal{R}_G id ?\beta s_1 = s_1$  using *l''-fresh-s<sub>1</sub> side* by *auto*  
     have  $\sigma$ :  $\mathcal{R}_S id ?\beta \sigma = \sigma$  using *l''-fresh-s<sub>1</sub> s<sub>1</sub>-r side* by *auto*  
     have  $\mathcal{E}$ :  $\mathcal{R}_C id ?\beta \mathcal{E} = \mathcal{E}$   
     proof  
       show  $l \notin LID_C \mathcal{E}$  using *s<sub>1</sub>-r side* by *auto*  
       show  $l'' \notin LID_C \mathcal{E}$  using *l''-fresh-left s<sub>2</sub>* by *auto*  
     qed  
     have  $\tau lv$ :  $\mathcal{R}_S id (id(l := l'', l'' := l)) (\tau(l \mapsto v)) = (\tau(l'' \mapsto v))$   
     proof –  
       have  $\tau$ :  $\mathcal{R}_S id ?\beta \tau = \tau$  using *l''-fresh-s<sub>1</sub> s<sub>1</sub>-r side* by *auto*  
       have  $v$ :  $\mathcal{R}_V id ?\beta v = v$   
       proof  
         show  $l \notin LID_V v$  using *s<sub>1</sub>-r side* by *auto*  
         show  $l'' \notin LID_V v$  using *l''-fresh-s<sub>1</sub> s<sub>1</sub>-r* by *auto*  
       qed  
       show *?thesis* by (*simp add:  $\tau v$  bij- $\beta$* )  
     qed  
     have  $\sigma'$ :  $\mathcal{R}_S id ?\beta \sigma' = \sigma'$  using *l''-fresh-s<sub>1</sub> new(2-3)* by (*auto simp add: True ID-distr-global-conditional*)  
     have  $\mathcal{E}'$ :  $\mathcal{R}_C id ?\beta \mathcal{E}' = \mathcal{E}'$  using *l''-fresh-s<sub>1</sub> new(2-3)* by (*auto simp add: True ID-distr-global-conditional*)  
     have  $\tau l''v$ :  $\mathcal{R}_S id (id(l := l'', l'' := l)) (\tau'(l'' \mapsto v')) = (\tau'(l \mapsto v'))$   
     proof –  
       have  $\tau'$ :  $\mathcal{R}_S id ?\beta \tau' = \tau'$  using *new(2-3) l''-fresh-s<sub>1</sub>* by (*auto simp add: True ID-distr-global-conditional*)  
       have  $v'$ :  $\mathcal{R}_V id ?\beta v' = v'$  using *new(2-3) l''-fresh-s<sub>1</sub>* by (*auto simp add: True ID-distr-global-conditional*)  
       show *?thesis* by (*simp add:  $\tau' v'$  bij- $\beta$* )  
     qed  
     show *?thesis* using *True neq s<sub>1</sub>  $\sigma$   $\mathcal{E}$   $\tau lv$   $\sigma'$   $\mathcal{E}'$   $\tau l''v$   $s_2$   $l-l''$  *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 ∉ LIDG s2'
  by (rule revision-stepE[OF left-step]) (use False side new(1-2) in ⟨auto
simp add: s1-r⟩)
have l'-fresh-right: l' ∉ LIDG s2
  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 s2 new(1)]) (use new assms
l-fresh-left l'-fresh-right s2 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'' ∉ RIDG s2
  by (rule only-fork-introduces-rids'[OF left-step]) (simp add: s1-r fork(3))+
have l-fresh-right: l ∉ LIDG s2'
  by (rule only-new-introduces-lids'[OF right]) (simp add: side fork(2))+
show ?thesis by (rule local-and-fork-commute[OF s2 fork(1)]) (use fork(1-2)
neq s2 l-fresh-right r''-fresh-left s1-r in auto)
next
case join
show ?thesis by (rule SLC-sym, rule SLC-join, use join left-step neq in auto)
next
case joinε
show ?thesis by (rule SLC-sym, rule SLC-joinε, use joinε left-step neq in auto)
qed
qed

```

### 7.2.7 Case fork

lemma *SLC-fork*:

assumes

$s_1$ - $r$ :  $s_1 \ r = \text{Some } (\sigma, \tau, \mathcal{E} \ [Rfork \ e])$  **and**

$s_2$ :  $s_2 = (s_1(r \mapsto (\sigma, \tau, \mathcal{E} \ [VE \ (Rid \ left\text{-}forkee)])), left\text{-}forkee \mapsto (\sigma;;\tau, \varepsilon, e))$  **and**

$side$ :  $left\text{-}forkee \notin RID_G \ s_1$  **and**

$right$ :  $revision\text{-}step \ r' \ s_1 \ s_2'$  **and**

$neq$ :  $r \neq r'$  **and**

$reach$ :  $reachable \ (s_1 :: ('r, 'l, 'v) \ global\text{-}state)$  **and**

$rid\text{-}inf$ :  $infinite \ (UNIV :: 'r \ set)$

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**  $(\text{auto simp add: ID-distr-global-conditional})$   
**show** *?thesis*  
**proof**  $(\text{use right in } \langle \text{cases rule: revision-stepE} \rangle)$   
**case** *app*  
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule SLC-app})$   $(\text{use assms(1–5) app in } \langle \text{auto simp add: ID-distr-global-conditional} \rangle)$   
**next**  
**case** *ifTrue*  
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule SLC-ifTrue})$   $(\text{use assms(1–5) ifTrue in } \langle \text{auto simp add: ID-distr-global-conditional} \rangle)$   
**next**  
**case** *ifFalse*  
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule SLC-ifFalse})$   $(\text{use assms(1–5) ifFalse in } \langle \text{auto simp add: ID-distr-global-conditional} \rangle)$   
**next**  
**case**  $(\text{new } \dots \text{ l})$   
**have** *l-fresh-left*:  $l \notin \text{LID}_G\ s_2$   
**by**  $(\text{rule only-new-introduces-lids'}[\text{OF left-step}])$   $(\text{simp add: } s_1-r\ \text{new}(3))$   
**have** *r''-fresh-right*:  $\text{left-forkee} \notin \text{RID}_G\ s_2'$   
**by**  $(\text{rule only-fork-introduces-rids'}[\text{OF right}])$   $(\text{simp add: side new}(2))$   
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule local-and-fork-commute}[\text{OF new}(1)\ s_2])$   
 $(\text{use new}(1–2)\ \text{neq } s_1-r\ r''\text{-fresh-right } l\text{-fresh-left } s_2 \text{ in auto})$   
**next**  
**case** *get*  
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule SLC-get})$   $(\text{use assms(1–5) get in } \langle \text{auto simp add: ID-distr-global-conditional} \rangle)$   
**next**  
**case** *set*  
**show** *?thesis* **by**  $(\text{rule SLC-sym, rule SLC-set})$   $(\text{use assms(1–5) set in } \langle \text{auto simp add: ID-distr-global-conditional} \rangle)$   
**next**  
**case**  $(\text{fork } \sigma'\ \tau'\ \mathcal{E}'\ e'\ \text{right-forkee})$   
**have** *r'-unchanged-left*:  $s_2\ r' = s_1\ r'$  **using** *side*  $\text{fork}(2)\ \text{neq } s_2$  **by** *auto*  
**have** *r-unchanged-right*:  $s_2'\ r = s_1\ r$  **using**  $\text{fork}(1,3)\ \text{neq } s_1-r$  **by** *auto*  
**have**  $r \neq \text{left-forkee}$  **using**  $s_1-r$  *side* **by** *auto*  
**have**  $r \neq \text{right-forkee}$  **using**  $\text{fork}(3)\ s_1-r$  **by** *auto*  
**have**  $r' \neq \text{left-forkee}$  **using**  $\text{fork}(2)$  *side* **by** *auto*  
**have**  $r' \neq \text{right-forkee}$  **using**  $\text{fork}(2)\ \text{fork}(3)$  **by** *auto*  
**show** *?thesis*  
**proof**  $(\text{cases left-forkee} = \text{right-forkee})$   
**case** *True*  
**obtain**  $r''$  **where** *r''-fresh-left*:  $r'' \notin \text{RID}_G\ s_2$   
**using** *RID<sub>G</sub>-finite-invariant ex-new-if-finite left-step reach reachable-imp-identifiers-finite(1)*  
*rid-inf* **by** *blast*  
**hence**  $r'' \neq \text{left-forkee}$  **using**  $\text{assms}(2)$  **by** *auto*  
**have**  $r'' \neq r$  **using** *r''-fresh-left*  $s_2$  **by** *auto*

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

```

    show  $r'' \notin RID_E e'$  using fork(2)  $r''$ -fresh- $s_1$  by auto
  qed
  show ?thesis using True fork(1) neq  $\sigma \tau \mathcal{E} e \sigma' \tau' \mathcal{E}' e' s_1 s_2$ 
    bij- $\alpha$   $\langle r'' \neq \text{left-forkee} \rangle \langle r' \neq \text{left-forkee} \rangle \langle r \neq \text{left-forkee} \rangle \langle r'' \neq r \rangle \langle r'' \neq$ 
 $r' \rangle$ 
    by auto
  qed
  qed (simp add: bij- $\alpha$ )+
  show ?thesis using equiv left-convergence right-convergence by blast
next
case False
have right-forkee-fresh-left:  $\text{right-forkee} \notin RID_G s_2$ 
  using False  $\langle r \neq \text{left-forkee} \rangle \langle r \neq \text{right-forkee} \rangle$  fork(3)  $s_1-r$ 
  by (auto simp add:  $s_2$  ID-distr-global-conditional, auto)
have left-forkee-fresh-right:  $\text{left-forkee} \notin RID_G s_2'$ 
  using False  $\langle r' \neq \text{right-forkee} \rangle \langle r' \neq \text{left-forkee} \rangle$  side fork(2)
  by (auto simp add: fork(1) ID-distr-global-conditional fun-upd-twist)
show ?thesis
proof(rule exI[where  $x=s_2(r' := s_2' r', \text{right-forkee} := s_2' \text{right-forkee})$ ],
  rule exI[where  $x=s_2'(r := s_2 r, \text{left-forkee} := s_2 \text{left-forkee})$ ],
  rule SLC-commute)
  show  $s_2(r' := s_2' r', \text{right-forkee} := s_2' \text{right-forkee}) = s_2'(r := s_2 r,$ 
 $\text{left-forkee} := s_2 \text{left-forkee})$ 
    using False  $\langle r \neq \text{right-forkee} \rangle \langle r' \neq \text{left-forkee} \rangle \langle r' \neq \text{right-forkee} \rangle$  fork(1)
  neq  $s_2$  by auto
  show revision-step  $r' s_2 (s_2(r' := s_2' r', \text{right-forkee} := s_2' \text{right-forkee}))$ 
    using fork(1-2)  $r'$ -unchanged-left revision-step.fork right-forkee-fresh-left
  by auto
  show revision-step  $r s_2'(s_2'(r := s_2 r, \text{left-forkee} := s_2 \text{left-forkee}))$ 
    using left-forkee-fresh-right  $r$ -unchanged-right revision-step.fork  $s_1-r s_2$  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 $_\epsilon$ 
  show ?thesis by (rule SLC-sym, rule SLC-join $_\epsilon$ , use join $_\epsilon$  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**  
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$   
**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 $_{\epsilon}$ )  
**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**  
 $s_3\text{-eq-}s_3': s_3 \approx s_3'$  **and**  
 $l\text{-join}$ : revision-step  $r' s_2 s_3 \vee s_2 = s_3$  **and**  
 $r\text{-join}$ : 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**  $l\text{-join steps-def}$  **by** auto  
**have**  $s_2' s_3: s_2' \rightsquigarrow^= s_3'$  **using**  $r\text{-join steps-def}$  **by** auto  
**show** ?thesis **using**  $s_2' s_3 s_2 s_3 s_3\text{-eq-}s_3'$  **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:  $s_1 \rightsquigarrow s_2 \implies s_1 \rightsquigarrow s_2' \implies ?thesis$  **using** *SLC lid-inf reach rid-inf* **by**  
*blast*  
**have** 2:  $s_1 = s_2 \implies s_1 \rightsquigarrow s_2' \implies ?thesis$   
**by** (*rule exI[where x=s<sub>2</sub>]*, *rule exI[where x=s<sub>2</sub>']*) (*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*  $\alpha \implies (\alpha r \in RID_V (\mathcal{R}_V \alpha \beta v)) = (r \in RID_V v)$   
*bij*  $\beta \implies (\beta l \in LID_V (\mathcal{R}_V \alpha \beta v)) = (l \in LID_V v)$   
**by** (*auto simp add: bij-is-inj inj-image-mem-iff val.set-map(1-2)*)

**lemma** *in-renamed-in-unlabelled-expr*:

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

**lemma** *in-renamed-in-unlabelled-store*:

**assumes**  
*bij- $\alpha$* : *bij*  $\alpha$  **and**  
*bij- $\beta$* : *bij*  $\beta$

**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<sub>S</sub>E*)

**fix**  $l v$

**assume** *map*:  $\mathcal{R}_S \alpha \beta \sigma l = \text{Some } v$  **and**  *$\alpha r$* :  $\alpha r \in RID_V v$

**hence**  $\sigma (inv \beta l) = \text{Some } (\mathcal{R}_V (inv \alpha) (inv \beta) v)$

**using** *bij- $\alpha$*  *bij- $\beta$*  **by** (*auto simp add: bij-is-inj*)

**have**  $r \in RID_V (\mathcal{R}_V (inv \alpha) (inv \beta) v)$

**using** *bij- $\alpha$*  *bij- $\beta$*   *$\alpha r$*  *map* **by** (*auto simp add: bij-is-inj in-renamed-in-unlabelled-val(1)*)

**show**  $r \in RID_S \sigma$

**using**  $\langle \sigma (inv \beta l) = \text{Some } (\mathcal{R}_V (inv \alpha) (inv \beta) v) \rangle \langle r \in RID_V (\mathcal{R}_V (inv \alpha) (inv \beta) v) \rangle$  **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) RIDSE RIDSI bij- $\alpha$  bij- $\beta$  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 LIDSE)
    assume  $\beta l \in dom (\mathcal{R}_S \alpha \beta \sigma)$ 
    thus  $l \in LID_S \sigma$  by (auto simp add: LIDSI(1) bij- $\beta$  bijection.intro bijec-
tion.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- $\beta$  by (auto simp add: LIDSI(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 LIDSE)
    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- $\beta$  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- $\beta$  by (auto simp add: LIDSI(2) in-renamed-in-unlabelled-val(2))
qed
qed
qed

```

**lemma** *in-renamed-in-unlabelled-local:*

**assumes**

*bij- $\alpha$ : bij  $\alpha$*  **and**

*bij- $\beta$ : bij  $\beta$*

**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- $\alpha$ : bij  $\alpha$*  **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 RIDGE)**  
**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**  $\alpha r: \alpha r \in RID_L ls$   
**have**  $s\text{-inv-}\alpha r': s (inv\ \alpha\ r') = Some (\mathcal{R}_L (inv\ \alpha) (inv\ \beta)\ ls)$   
**proof –**  
**have**  $inv\ \alpha\ r' \in dom\ s$  **using**  $\mathcal{R}sr'$  **by auto**  
**then obtain**  $ls'$  **where**  $s\text{-inv-}\alpha r: s (inv\ \alpha\ r') = Some\ ls'$  **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*  $\mathcal{R}_L\text{-inv}\ s\text{-inv-}\alpha r\ bij\text{-}\alpha\ bij\text{-}\beta$ )  
**qed**  
**have**  $r\text{-in}: r \in RID_L (\mathcal{R}_L (inv\ \alpha) (inv\ \beta)\ ls)$   
**by** (*metis*  $bij\text{-}\alpha\ bij\text{-}\beta\ bij\text{-}imp\text{-}bij\text{-}inv\ bijection.intro\ bijection.inv\text{-}left\ in\text{-}renamed\text{-}in\text{-}unlabelled\text{-}local(1)$   
 $\alpha r$ )  
**show**  $r \in RID_G s$   
**using**  $r\text{-in}\ s\text{-inv-}\alpha r'$  **by blast**  
**qed**  
**next**  
**assume**  $r \in RID_G s$   
**thus**  $\alpha r \in RID_G (\mathcal{R}_G \alpha \beta s)$   
**proof (rule RIDGE)**  
**assume**  $r \in dom\ s$   
**hence**  $\alpha r \in dom (\mathcal{R}_G \alpha \beta s)$   
**by** (*metis* (*mono-tags, lifting*)  $bij\text{-}\alpha\ domD\ domI\ fun\text{-}upd\text{-}same\ fun\text{-}upd\text{-}triv\ renaming\text{-}distr\text{-}global(1)$ )  
**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*  $ID\text{-}distr\text{-}global(1)\ UnI2\ bij\text{-}\alpha\ bij\text{-}\beta\ fun\text{-}upd\text{-}triv\ in\text{-}renamed\text{-}in\text{-}unlabelled\text{-}local(1)$   
 $insertI2\ renaming\text{-}distr\text{-}global(1)$ )  
**qed**  
**qed**  
**show**  $(\beta l \in LID_G (\mathcal{R}_G \alpha \beta s)) = (l \in LID_G s)$

**proof** (*rule iffI*)  
**assume**  $\beta l \in LID_G (\mathcal{R}_G \alpha \beta s)$   
**from this obtain**  $r\ ls$  **where**  $Rs\text{-}ls: \mathcal{R}_G \alpha \beta s\ r = \text{Some } ls$  **and**  $\beta l\text{-in-}ls: \beta l \in LID_L\ ls$  **by** *blast*  
**hence**  $l \in LID_L (\mathcal{R}_L (inv\ \alpha) (inv\ \beta)\ ls)$   
**by** (*metis bij- $\alpha$  bij- $\beta$  bij-imp-bij-inv bijection.intro bijection.inv-left in-renamed-in-unlabelled-local(2)*)  
**hence**  $l \in LID_G (\mathcal{R}_G (inv\ \alpha) (inv\ \beta) (\mathcal{R}_G \alpha \beta s))$   
**by** (*metis (mono-tags, lifting) LID\_G I Rs-ls bij- $\alpha$  bij-imp-bij-inv fun-upd-idem-iff renaming-distr-global(1)*)  
**thus**  $l \in LID_G s$  **using**  $bij\text{-}\alpha\ bij\text{-}\beta$  **by** (*metis  $\mathcal{R}_G\text{-inv}$* )  
**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**  $\beta l \in LID_G (\mathcal{R}_G \alpha \beta s)$  **by** (*simp add: bij- $\alpha$  bij- $\beta$  in-renamed-in-unlabelled-local(2)*)  
**qed**  
**qed**

**lemma** *mimicking:*

**assumes**  
*step: revision-step*  $r\ s\ s'$  **and**  
*bij- $\alpha$ :*  $bij\ \alpha$  **and**  
*bij- $\beta$ :*  $bij\ \beta$   
**shows** *revision-step*  $(\alpha\ r) (\mathcal{R}_G \alpha \beta s) (\mathcal{R}_G \alpha \beta s')$   
**proof** (*use step in (cases rule: revision-stepE)*)  
**case** *app*  
**then show** *?thesis* **by** (*auto simp add: bij- $\alpha$  bij- $\beta$  bijection.intro bijection.inv-left renaming-distr-subst*)  
**next**  
**case** (*new - - - l*)  
**have**  $\beta l\text{-fresh}: \beta l \notin LID_G (\mathcal{R}_G \alpha \beta s)$   
**by** (*simp add: bij- $\alpha$  bij- $\beta$  in-renamed-in-unlabelled-global(2) new(3)*)  
**show** *?thesis* **using**  $\beta l\text{-fresh}\ new$  **by** (*auto simp add: bij- $\alpha$  bij- $\beta$  bijection.intro bijection.inv-left*)  
**next**  
**case** (*fork  $\sigma\ \tau\ \mathcal{E}\ e\ r'$* )  
**have**  $\alpha r'\text{-fresh}: \alpha r' \notin RID_G (\mathcal{R}_G \alpha \beta s)$   
**by** (*simp add: bij- $\alpha$  bij- $\beta$  in-renamed-in-unlabelled-global(1) fork(3)*)  
**have**  $s\text{-r-as-upd}: s = (s(r \mapsto (\sigma, \tau, \mathcal{E} [Rfork\ e])))$  **using** *fork(2)* **by** *auto*  
**have**  $src: \mathcal{R}_G \alpha \beta s (\alpha\ r) = \text{Some} (\mathcal{R}_S \alpha \beta \sigma, \mathcal{R}_S \alpha \beta \tau, (\mathcal{R}_C \alpha \beta \mathcal{E}) [Rfork (\mathcal{R}_E \alpha \beta e)])$   
**by** (*subst s-r-as-upd, simp add: bij- $\alpha$* )  
**show** *?thesis* **using**  $\alpha r'\text{-fresh}\ src$  *revision-step.fork fork(1) bij- $\alpha$*  **by** *auto*  
**qed** (*auto simp add: bij- $\alpha$  bij- $\beta$  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- $\alpha$* : *bij*  $\alpha$  **and**  
*bij- $\beta$* : *bij*  $\beta$  **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- $\alpha$*  *bij- $\beta$*  *step mimicking* **by** *auto*  
**have** *eq*:  $s_2 \approx (\mathcal{R}_G \alpha \beta s_2)$  **using** *bij- $\alpha$*  *bij- $\beta$*  **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**  
 $s_1\text{-eq-}s_1'$ :  $s_1 \approx s_1'$  **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**  $s_1\text{-eq-}s_1'$  **obtain**  $\alpha \beta$  **where**  
*bij- $\alpha$* : *bij*  $\alpha$  **and**  
*bij- $\beta$* : *bij*  $\beta$  **and**  
*R*:  $\mathcal{R}_G \alpha \beta s_1 = s_1'$   
**by** *blast*  
**from**  $s_1 s_2$  **obtain**  $n$  **where**  $(s_1, s_2) \in [\rightsquigarrow] \rightsquigarrow^n$  **using** *rtrancl-power* **by** *blast*  
**thus** *?thesis*  
**proof** (*induct*  $n$  *arbitrary*:  $s_2$ )  
**case**  $0$   
**thus** *?case* **using**  $s_1\text{-eq-}s_1'$  **by** *auto*  
**next**  
**case** (*Suc*  $n$ )  
**obtain**  $x$  **where**  $n\text{-steps}$ :  $(s_1, x) \in [\rightsquigarrow] \rightsquigarrow^n$  **and** *step*:  $x \rightsquigarrow s_2$  **using** *Suc.prem*  
**by** *auto*  
**obtain**  $x'$  **where**  $x\text{-eq-}x'$ :  $x \approx x'$  **and**  $s_1' x$ :  $s_1' \rightsquigarrow^* x'$  **using** *Suc.hyps*  $n\text{-steps}$   
**by** *blast*  
**obtain**  $s_2'$  **where**  $s_2\text{-eq-}s_2'$ :  $s_2 \approx s_2'$  **and**  $x' s_2'$ :  $x' \rightsquigarrow s_2'$   
**by** (*meson* *step mimic-single-step*  $x\text{-eq-}x'$ )  
**show** *?case* **using**  $s_1' x$   $s_2\text{-eq-}s_2'$  *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-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**  $(s_1, s_2) \in [\rightsquigarrow]^\sim n$  **using** *rtrancl-power* **by** *blast*  
**from** *reach*  $s_1 s_2'$  **and this show** *?thesis*  
**proof** (*induct n arbitrary: s1 s2 s2'*)  
**case** 0  
**hence**  $s_1 = s_2$  **by** *simp*  
**hence**  $s_2 s_2'$ :  $s_2 \rightsquigarrow^* s_2'$  **using** *0.premis(2)* **by** *blast*  
**show** *?case* **by** (*rule exI[where x=s2]*, *rule exI[where x=s2']*) (*use s2 s2' in*  
*(simp add: alpha-beta-refl)*)  
**next**  
**case** (*Suc n*)  
**obtain**  $a$  **where**  $s_1 a$ :  $s_1 \rightsquigarrow a$  **and**  $as_2$ :  $(a, s_2) \in [\rightsquigarrow]^\sim n$  **by** (*meson Suc.premis(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.premis(1) Suc.premis(2)*  
*alpha-beta-sym lid-inf rid-inf s1 a*)  
**obtain**  $d e$  **where**  $d \approx e$   $s_2 \rightsquigarrow^* d$   $b \rightsquigarrow^* e$   
**by** (*meson Suc.hyps Suc.premis(1) (a rightsquigarrow^= b) as2 reachability-closed-under-execution-step*  
*s1 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) (s2 rightsquigarrow^\* d) (s2' 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-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**  $(s_1, s_2) \in [\rightsquigarrow]^\sim n$  **using**  $s_1 s_2$  *rtrancl-power* **by** *blast*

**from** *reach equiv s1 s2'* **and this show** *?thesis*

**proof** (*induct n arbitrary: s1 s1' s2 s2'*)

**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.premis(2,3) mimic-trans* **by** *blast*

**have**  $s_2 \rightsquigarrow^* s_2''$  **using**  $\langle s_1 = s_2 \rangle \langle s_1 \rightsquigarrow^* s_2'' \rangle$  **by** *blast*  
**show**  $?case$  **by** (*rule* *exI*[**where**  $x=s_2''$ ], *rule* *exI*[**where**  $x=s_2'$ ]) (*auto simp*  
*add:  $\alpha\beta$ -sym  $\langle s_2 \rightsquigarrow^* s_2'' \rangle \langle s_2' \approx s_2'' \rangle$* )  
**next**  
**case** *step: (Suc n)*  
**obtain**  $a$  **where**  $s_1 a: (s_1, a) \in [\rightsquigarrow]^\sim n$  **and**  $as_2: a \rightsquigarrow s_2$  **using** *step.prem*s(4)  
**by** *auto*  
**have** *reachable a* **using** *reachability-closed-under-execution relpow-imp-rtrancl*  
 $s_1 a$  *step.prem*s(1) **by** *blast*  
**obtain**  $b c$  **where**  $a \rightsquigarrow^* b$   $s_2' \rightsquigarrow^* c$   $b \approx c$   
**using**  $s_1 a$  *step.hyps step.prem*s(1-3) **by** *blast*  
**have**  $a \rightsquigarrow^* s_2$  **by** (*simp add: as<sub>2</sub> 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  $\langle a \rightsquigarrow^* b \rangle \langle reachable a \rangle as_2 lid-inf refl-rewritesI rid-inf$*   
*strip-lemma*)  
**obtain**  $s_3' c$  **where**  $s_3 \approx s_3'$   $c \rightsquigarrow^* s_3'$   
**by** (*meson  $\alpha\beta$ -trans  $\langle b \approx c \rangle \langle b \rightsquigarrow^* d \rangle \langle s_3 \approx d \rangle mimic-trans$* )  
**show**  $?case$  **by** (*meson  $\langle c \rightsquigarrow^* s_3' \rangle \langle s_2 \rightsquigarrow^* s_3 \rangle \langle s_2' \rightsquigarrow^* c \rangle \langle s_3 \approx s_3' \rangle transD$*   
*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- $\alpha$ : 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- $\alpha$  prog-expr* **by** *auto*

**qed** (*simp add: bij- $\alpha$* )<sup>+</sup>

**have** *reach: reachable  $(\varepsilon(r \mapsto (\varepsilon, \varepsilon, e)))$*

**by** (*simp add: initial-state-reachable prog-expr*)

**have**  $\exists a b. (a \approx b) \wedge s \rightsquigarrow^* a \wedge s' \rightsquigarrow^* b$

**by** (*rule confluence-modulo-equivalence[OF x y equiv reach lid-inf rid-inf]*)

**from this obtain**  $a b$  **where**  $s \rightsquigarrow^* a$   $s' \rightsquigarrow^* b$   $a \approx b$  **by** *blast*



```

have  $s = a$  by (meson  $\langle s \rightsquigarrow^* a \rangle$  e-terminates-in-s maximal-execution-def rtranclD
terminates-in-def tranclD)
have  $s' = b$  by (meson  $\langle s' \rightsquigarrow^* b \rangle$  converse-rtranclE e-terminates-in-s' maxi-
mal-execution-def terminates-in-def)
show ?thesis using  $\langle a \approx b \rangle \langle s = a \rangle \langle s' = b \rangle$  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>.