

Formalization of Concurrent Revisions

Roy Overbeek

May 26, 2024

Abstract

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

Contents

1	Data	4
1.1	Function notations	4
1.2	Values, expressions and execution contexts	4
1.3	Plugging and decomposing	5
1.4	Stores and states	6
2	Occurrences	6
2.1	Definitions	7
2.1.1	Revision identifiers	7
2.1.2	Location identifiers	7
2.2	Inference rules	7
2.2.1	Introduction and elimination rules	7
2.2.2	Distributive laws	8
2.2.3	Miscellaneous laws	10

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

3	Renaming	10
3.1	Definitions	10
3.2	Introduction rules	11
3.3	Renaming-equivalence	12
3.3.1	Identity	12
3.3.2	Composition	12
3.3.3	Inverse	12
3.3.4	Equivalence	13
3.4	Distributive laws	13
3.4.1	Expression	13
3.4.2	Store	13
3.4.3	Global	13
3.5	Miscellaneous laws	14
3.6	Swaps	14
4	Substitution	15
4.1	Definition	15
4.2	Trivial model	16
4.3	Example model	16
4.3.1	Preliminaries	16
4.3.2	Definition	16
4.3.3	Proof obligations	17
5	Operational Semantics	17
5.1	Definition	18
5.2	Introduction lemmas for identifiers	18
5.3	Domain subsumption	19
5.3.1	Definitions	19
5.3.2	The theorem	19
5.4	Relaxed definition of the operational semantics	20
6	Executions	20
6.1	Generalizing the original transition	21
6.1.1	Definition	21
6.1.2	Closures	21
6.2	Properties	21
6.3	Invariants	22
6.3.1	Inductive invariance	22
6.3.2	Subsumption is invariant	22
6.3.3	Finitude is invariant	22
6.3.4	Reachability is invariant	24

7	Determinacy	24
7.1	Rule determinism	25
7.2	Strong local confluence	26
7.2.1	Local determinism	26
7.2.2	General principles	26
7.2.3	Case join-epsilon	26
7.2.4	Case join	27
7.2.5	Case local	27
7.2.6	Case new	29
7.2.7	Case fork	29
7.2.8	The theorem	30
7.3	Diagram Tiling	30
7.3.1	Strong local confluence diagrams	30
7.3.2	Mimicking diagrams	30
7.3.3	Strip diagram	32
7.3.4	Confluence diagram	32
7.4	Determinacy	32

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  $\tau'$ 
  <proof>
```

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) *cntxt* \Rightarrow ('r,'l,'v) *expr* \Rightarrow ('r,'l,'v) *expr* (**infix** \triangleleft 60) **where**

$\square \triangleleft e = e$
| *ApplyL* $_{\mathcal{E}}$ $\mathcal{E} e1 \triangleleft e = \text{Apply } (\mathcal{E} \triangleleft e) e1$
| *ApplyR* $_{\mathcal{E}}$ *val* $\mathcal{E} \triangleleft e = \text{Apply } (VE \text{ val}) (\mathcal{E} \triangleleft e)$
| *Ite* $_{\mathcal{E}}$ $\mathcal{E} e1 e2 \triangleleft e = \text{Ite } (\mathcal{E} \triangleleft e) e1 e2$
| *Ref* $_{\mathcal{E}}$ $\mathcal{E} \triangleleft e = \text{Ref } (\mathcal{E} \triangleleft e)$
| *Read* $_{\mathcal{E}}$ $\mathcal{E} \triangleleft e = \text{Read } (\mathcal{E} \triangleleft e)$
| *AssignL* $_{\mathcal{E}}$ $\mathcal{E} e1 \triangleleft e = \text{Assign } (\mathcal{E} \triangleleft e) e1$
| *AssignR* $_{\mathcal{E}}$ *l* $\mathcal{E} \triangleleft e = \text{Assign } (VE (\text{Loc } l)) (\mathcal{E} \triangleleft e)$
| *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*]: $(\mathcal{E}[e1] = \mathcal{E}[e2]) = (e1 = e2)$ *<proof>*

lemma *VE-empty-cntxt* [*simp*]: $(VE v = \mathcal{E}[e]) = (\mathcal{E} = \square \wedge VE v = e)$ *<proof>*

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

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

inductive-simps *redex-simps* [*simp*]: *redex* *e*

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

lemma *plugged-redex-not-val* [*simp*]: *redex* *r* $\Longrightarrow (\mathcal{E} \triangleleft r) \neq (VE t)$ *<proof>*

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

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

| *rjoin*: $\llbracket \neg \text{redex } (R\text{join } e); \text{decompose } e \ \mathcal{E} \ r \rrbracket \implies \text{decompose } (R\text{join } e) \ (R\text{join}_{\mathcal{E}} \ \mathcal{E}) \ r$

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

lemma *plug-decomposition-equivalence*: $\text{redex } r \implies \text{decompose } e \ \mathcal{E} \ r = (\mathcal{E}[r] = e)$
 ⟨*proof*⟩

lemma *unique-decomposition*: $\text{decompose } e \ \mathcal{E}_1 \ r_1 \implies \text{decompose } e \ \mathcal{E}_2 \ r_2 \implies \mathcal{E}_1 = \mathcal{E}_2 \wedge r_1 = r_2$
 ⟨*proof*⟩

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*⟩

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 $(\sigma, \tau, e) = \text{dom } \sigma \cup \text{dom } \tau$

fun *LID-snapshot* :: (r, l, v) *local-state* \Rightarrow (r, l, v) *store* (*- σ* 200) **where**

LID-snapshot $(\sigma, \tau, e) = \sigma$

fun *LID-local-store* :: (r, l, v) *local-state* \Rightarrow (r, l, v) *store* (*- τ* 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 $RID_S :: ('r, 'l, 'v) \text{ store} \Rightarrow 'r \text{ set}$ **where**
 $RID_S \sigma \equiv \bigcup (RID_V \text{ `ran } \sigma)$

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

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

2.1.2 Location identifiers

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

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

2.2 Inference rules

2.2.1 Introduction and elimination rules

lemma RID_{SI} [intro]: $\sigma \text{ l} = \text{Some } v \Longrightarrow r \in RID_V v \Longrightarrow r \in RID_S \sigma$
(proof)

lemma RID_{SE} [elim]: $r \in RID_S \sigma \Longrightarrow (\bigwedge l v. \sigma \text{ l} = \text{Some } v \Longrightarrow r \in RID_V v \Longrightarrow P) \Longrightarrow P$
(proof)

lemma RID_{LI} [intro, consumes 1]:

assumes $s = (\sigma, \tau, e)$

shows

$r \in RID_S \sigma \Longrightarrow r \in RID_L s$

$r \in RID_S \tau \Longrightarrow r \in RID_L s$

$r \in RID_E e \Longrightarrow r \in RID_L s$

(proof)

lemma RID_{LE} [elim]:

$\llbracket r \in RID_L s; (\bigwedge \sigma \tau e. s = (\sigma, \tau, e) \Longrightarrow (r \in RID_S \sigma \Longrightarrow P) \Longrightarrow (r \in RID_S \tau \Longrightarrow P) \Longrightarrow (r \in RID_E e \Longrightarrow P) \Longrightarrow P) \rrbracket \Longrightarrow P$
(proof)

lemma RID_{GI} [intro]:

$s \text{ r} = \text{Some } v \Longrightarrow r \in RID_G s$

$s \text{ r}' = \text{Some } ls \Longrightarrow r \in RID_L ls \Longrightarrow r \in RID_G s$

$\langle proof \rangle$

lemma $RID_G E$ [elim]:

assumes

$$r \in RID_G s$$

$$r \in dom s \implies P$$

$$\bigwedge r' ls. s r' = Some ls \implies r \in RID_L ls \implies P$$

shows P

$\langle proof \rangle$

lemma $LID_S I$ [intro]:

$$\sigma l = Some v \implies l \in LID_S \sigma$$

$$\sigma l' = Some v \implies l \in LID_V v \implies l \in LID_S \sigma$$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $LID_L I$ [intro]:

assumes $s = (\sigma, \tau, e)$

shows

$$r \in LID_S \sigma \implies r \in LID_L s$$

$$r \in LID_S \tau \implies r \in LID_L s$$

$$r \in LID_E e \implies r \in LID_L s$$

$\langle proof \rangle$

lemma $LID_L E$ [elim]:

$$\llbracket l \in LID_L s; (\bigwedge \sigma \tau e. s = (\sigma, \tau, e) \implies (l \in LID_S \sigma \implies P) \implies (l \in LID_S \tau \implies P) \implies (l \in LID_E e \implies P) \implies P) \rrbracket \implies P$$

$\langle proof \rangle$

lemma $LID_G I$ [intro]: $s r = Some ls \implies l \in LID_L ls \implies l \in LID_G s$

$\langle proof \rangle$

lemma $LID_G E$ [elim]: $l \in LID_G s \implies (\bigwedge r ls. s r = Some ls \implies l \in LID_L ls \implies P) \implies P$

$\langle proof \rangle$

2.2.2 Distributive laws

lemma ID -distr-completion [simp]:

$$RID_E (\mathcal{E}[e]) = RID_C \mathcal{E} \cup RID_E e$$

$$LID_E (\mathcal{E}[e]) = LID_C \mathcal{E} \cup LID_E e$$

$\langle proof \rangle$

lemma *ID-restricted-store-subset-store*:

$$\begin{aligned} RID_S (\sigma(l := None)) &\subseteq RID_S \sigma \\ LID_S (\sigma(l := None)) &\subseteq LID_S \sigma \end{aligned}$$

<proof>

lemma *in-restricted-store-in-unrestricted-store* [intro]:

$$\begin{aligned} r \in RID_S (\sigma(l := None)) &\implies r \in RID_S \sigma \\ l' \in LID_S (\sigma(l := None)) &\implies l' \in LID_S \sigma \end{aligned}$$

<proof>

lemma *in-restricted-store-in-updated-store* [intro]:

$$\begin{aligned} r \in RID_S (\sigma(l := None)) &\implies r \in RID_S (\sigma(l \mapsto v)) \\ l' \in LID_S (\sigma(l := None)) &\implies l' \in LID_S (\sigma(l \mapsto v)) \end{aligned}$$

<proof>

lemma *ID-distr-store* [simp]:

$$\begin{aligned} RID_S (\tau(l \mapsto v)) &= RID_S (\tau(l := None)) \cup RID_V v \\ LID_S (\tau(l \mapsto v)) &= insert\ l\ (LID_S (\tau(l := None)) \cup LID_V v) \end{aligned}$$

<proof>

lemma *ID-distr-local* [simp]:

$$\begin{aligned} LID_L (\sigma, \tau, e) &= LID_S \sigma \cup LID_S \tau \cup LID_E e \\ RID_L (\sigma, \tau, e) &= RID_S \sigma \cup RID_S \tau \cup RID_E e \end{aligned}$$

<proof>

lemma *ID-restricted-global-subset-unrestricted*:

$$\begin{aligned} LID_G (s(r := None)) &\subseteq LID_G s \\ RID_G (s(r := None)) &\subseteq RID_G s \end{aligned}$$

<proof>

lemma *in-restricted-global-in-unrestricted-global* [intro]:

$$\begin{aligned} r' \in RID_G (s(r := None)) &\implies r' \in RID_G s \\ l \in LID_G (s(r := None)) &\implies l \in LID_G s \end{aligned}$$

<proof>

lemma *in-restricted-global-in-updated-global* [intro]:

$$\begin{aligned} r' \in RID_G (s(r := None)) &\implies r' \in RID_G (s(r \mapsto ls)) \\ l \in LID_G (s(r := None)) &\implies l \in LID_G (s(r \mapsto ls)) \end{aligned}$$

<proof>

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

$$\begin{aligned} RID_G (s(r \mapsto ls)) &= insert\ r\ (RID_G (s(r := None)) \cup RID_L ls) \\ LID_G (s(r \mapsto ls)) &= LID_G (s(r := None)) \cup LID_L ls \end{aligned}$$

<proof>

lemma *restrictions-inwards* [simp]:

$$x \neq x' \implies f(x := Some\ y, x' := None) = (f(x' := None, x := Some\ y))$$

<proof>

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>

lemma *in-combination-in-component* [intro]:

$$r \in RID_S (\sigma;;\tau) \implies r \notin RID_S \sigma \implies r \in RID_S \tau$$

$$r \in RID_S (\sigma;;\tau) \implies r \notin RID_S \tau \implies r \in RID_S \sigma$$

$$l \in LID_S (\sigma;;\tau) \implies l \notin LID_S \sigma \implies l \in LID_S \tau$$

$$l \in LID_S (\sigma;;\tau) \implies l \notin LID_S \tau \implies l \in LID_S \sigma$$

<proof>

lemma *in-mapped-in-component-of-combination* [intro]:

assumes

$$\text{maps-to-}v: (\sigma;;\tau) \ l = \text{Some } v \text{ and}$$

$$l' \text{-in-}v: l' \in LID_V v$$

shows

$$l' \notin LID_S \tau \implies l' \in LID_S \sigma$$

$$l' \notin LID_S \sigma \implies l' \in LID_S \tau$$

<proof>

lemma *elim-trivial-restriction* [simp]: $l \notin LID_S \tau \implies (\tau(l := \text{None})) = \tau$

<proof>

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>

end

3 Renaming

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

theory *Renaming*

imports *Occurrences*

begin

3.1 Definitions

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

(\mathcal{R}_V) **where**

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

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

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

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

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

fun \mathcal{R}_S :: ('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) *store* ⇒ ('r,'l,'v) *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>

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

fun \mathcal{R}_G :: ('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) *global-state* ⇒ ('r,'l,'v) *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>

3.2 Introduction rules

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

assumes

bij- β : $\text{bij } \beta$ **and**

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

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

shows

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

<proof>

lemma $\mathcal{R}_G I$ [intro]:
assumes
bij- α : *bij* α **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>

3.3 Renaming-equivalence

3.3.1 Identity

declare *val.map-id* [simp]
declare *expr.map-id* [simp]
declare *cntxt.map-id* [simp]
lemma \mathcal{R}_S -id [simp]: $\mathcal{R}_S\ id\ id\ \sigma = \sigma$ <proof>
lemma \mathcal{R}_L -id [simp]: $\mathcal{R}_L\ id\ id\ ls = ls$ <proof>
lemma \mathcal{R}_G -id [simp]: $\mathcal{R}_G\ id\ id\ s = s$ <proof>

3.3.2 Composition

declare *val.map-comp* [simp]
declare *expr.map-comp* [simp]
declare *cntxt.map-comp* [simp]
lemma \mathcal{R}_S -comp [simp]: $\llbracket\ \text{bij}\ \beta;\ \text{bij}\ \beta'\ \rrbracket \implies \mathcal{R}_S\ \alpha'\ \beta'(\mathcal{R}_S\ \alpha\ \beta\ s) = \mathcal{R}_S(\alpha' \circ \alpha)(\beta' \circ \beta)\ s$
<proof>
lemma \mathcal{R}_L -comp [simp]: $\llbracket\ \text{bij}\ \beta;\ \text{bij}\ \beta'\ \rrbracket \implies \mathcal{R}_L\ \alpha'\ \beta'(\mathcal{R}_L\ \alpha\ \beta\ ls) = \mathcal{R}_L(\alpha' \circ \alpha)(\beta' \circ \beta)\ ls$
<proof>
lemma \mathcal{R}_G -comp [simp]: $\llbracket\ \text{bij}\ \alpha;\ \text{bij}\ \alpha';\ \text{bij}\ \beta;\ \text{bij}\ \beta'\ \rrbracket \implies \mathcal{R}_G\ \alpha'\ \beta'(\mathcal{R}_G\ \alpha\ \beta\ s) = \mathcal{R}_G(\alpha' \circ \alpha)(\beta' \circ \beta)\ s$
<proof>

3.3.3 Inverse

lemma \mathcal{R}_V -inv [simp]: $\llbracket\ \text{bij}\ \alpha;\ \text{bij}\ \beta\ \rrbracket \implies (\mathcal{R}_V\ (\text{inv}\ \alpha)\ (\text{inv}\ \beta)\ v' = v) = (\mathcal{R}_V\ \alpha\ \beta\ v = v')$
<proof>
lemma \mathcal{R}_E -inv [simp]: $\llbracket\ \text{bij}\ \alpha;\ \text{bij}\ \beta\ \rrbracket \implies (\mathcal{R}_E\ (\text{inv}\ \alpha)\ (\text{inv}\ \beta)\ e' = e) = (\mathcal{R}_E\ \alpha\ \beta\ e = e')$
<proof>
lemma \mathcal{R}_C -inv [simp]: $\llbracket\ \text{bij}\ \alpha;\ \text{bij}\ \beta\ \rrbracket \implies (\mathcal{R}_C\ (\text{inv}\ \alpha)\ (\text{inv}\ \beta)\ \mathcal{E}' = \mathcal{E}) = (\mathcal{R}_C\ \alpha\ \beta\ \mathcal{E} = \mathcal{E}')$
<proof>
lemma \mathcal{R}_S -inv [simp]: $\llbracket\ \text{bij}\ \alpha;\ \text{bij}\ \beta\ \rrbracket \implies (\mathcal{R}_S\ (\text{inv}\ \alpha)\ (\text{inv}\ \beta)\ \sigma' = \sigma) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma = \sigma')$
<proof>

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

<proof>

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

<proof>

3.3.4 Equivalence

definition $\text{eq-states} :: ('r, 'l, 'v) \text{global-state} \Rightarrow ('r, 'l, 'v) \text{global-state} \Rightarrow \text{bool}$ ($- \approx -$ [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]:

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

<proof>

lemma eq-statesE [elim]:

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

<proof>

lemma $\alpha\beta$ -refl: $s \approx s$ *<proof>*

lemma $\alpha\beta$ -trans: $s \approx s' \Longrightarrow s' \approx s'' \Longrightarrow s \approx s''$

<proof>

lemma $\alpha\beta$ -sym: $s \approx s' \Longrightarrow s' \approx s$

<proof>

3.4 Distributive laws

3.4.1 Expression

lemma $\text{renaming-distr-completion}$ [simp]:

$$\mathcal{R}_E \alpha \beta (\mathcal{E}[e]) = ((\mathcal{R}_C \alpha \beta \mathcal{E})[\mathcal{R}_E \alpha \beta e])$$

<proof>

3.4.2 Store

lemma $\text{renaming-distr-combination}$ [simp]:

$$\mathcal{R}_S \alpha \beta (\sigma;;\tau) = (\mathcal{R}_S \alpha \beta \sigma;;\mathcal{R}_S \alpha \beta \tau)$$

<proof>

lemma $\text{renaming-distr-store}$ [simp]:

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

<proof>

3.4.3 Global

lemma $\text{renaming-distr-global}$ [simp]:

$\text{bij } \alpha \implies \mathcal{R}_G \alpha \beta (s(r \mapsto ls)) = (\mathcal{R}_G \alpha \beta s)(\alpha r \mapsto \mathcal{R}_L \alpha \beta ls)$
 $\text{bij } \alpha \implies \mathcal{R}_G \alpha \beta (s(r := \text{None})) = (\mathcal{R}_G \alpha \beta s)(\alpha r := \text{None})$
 ⟨proof⟩

3.5 Miscellaneous laws

lemma *rename-empty* [simp]:

$\mathcal{R}_S \alpha \beta \varepsilon = \varepsilon$
 $\mathcal{R}_G \alpha \beta \varepsilon = \varepsilon$
 ⟨proof⟩

3.6 Swaps

lemma *swap-bij*:

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

lemma *id-trivial-update* [simp]: $id(x := x) = id$ ⟨proof⟩

lemma *eliminate-renaming-val-expr* [simp]:

fixes

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

shows

$l \notin LID_V v \implies \mathcal{R}_V \alpha (\beta(l := l')) v = \mathcal{R}_V \alpha \beta v$
 $l \notin LID_E e \implies \mathcal{R}_E \alpha (\beta(l := l')) e = \mathcal{R}_E \alpha \beta e$
 $r \notin RID_V v \implies \mathcal{R}_V (\alpha(r := r')) \beta v = \mathcal{R}_V \alpha \beta v$
 $r \notin RID_E e \implies \mathcal{R}_E (\alpha(r := r')) \beta e = \mathcal{R}_E \alpha \beta e$
 ⟨proof⟩

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

$r \notin RID_C \mathcal{E} \implies \mathcal{R}_C (\alpha(r := r')) \beta \mathcal{E} = \mathcal{R}_C \alpha \beta \mathcal{E}$
 $l \notin LID_C \mathcal{E} \implies \mathcal{R}_C \alpha (\beta(l := l')) \mathcal{E} = \mathcal{R}_C \alpha \beta \mathcal{E}$
 ⟨proof⟩

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

$r \notin RID_V v \implies r' \notin RID_V v \implies \mathcal{R}_V (id(r := r', r' := r)) id v = v$
 $l \notin LID_V v \implies l' \notin LID_V v \implies \mathcal{R}_V id (id(l := l', l' := l)) v = v$
 ⟨proof⟩

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

$r \notin RID_E e \implies r' \notin RID_E e \implies \mathcal{R}_E (id(r := r', r' := r)) id e = e$
 $l \notin LID_E e \implies l' \notin LID_E e \implies \mathcal{R}_E id (id(l := l', l' := l)) e = e$
 ⟨proof⟩

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

$r \notin RID_C \mathcal{E} \implies r' \notin RID_C \mathcal{E} \implies \mathcal{R}_C (id(r := r', r' := r)) id \mathcal{E} = \mathcal{E}$
 $l \notin LID_C \mathcal{E} \implies l' \notin LID_C \mathcal{E} \implies \mathcal{R}_C id (id(l := l', l' := l)) \mathcal{E} = \mathcal{E}$
 ⟨proof⟩

lemma *eliminate-swap-store-rid* [*simp, intro*]:
 $r \notin RID_S \sigma \implies r' \notin RID_S \sigma \implies \mathcal{R}_S (id(r := r', r' := r)) id \sigma = \sigma$
 ⟨*proof*⟩

lemma *eliminate-swap-store-lid* [*simp, intro*]:
 $l \notin LID_S \sigma \implies l' \notin LID_S \sigma \implies \mathcal{R}_S id (id(l := l', l' := l)) \sigma = \sigma$
 ⟨*proof*⟩

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

lemma *eliminate-swap-global-lid* [*simp, intro*]:
 $l \notin LID_G s \implies l' \notin LID_G s \implies \mathcal{R}_G id (id(l := l', l' := l)) s = s$
 ⟨*proof*⟩

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

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

end

4.2 Trivial model

fun *constant-function* :: ('r,'l,'v) expr ⇒ 'v ⇒ ('r,'l,'v) expr ⇒ ('r,'l,'v) expr
where

constant-function e x e' = VE (CV Unit)

lemma *constant-function-models-substitution*:
substitution constant-function ⟨proof⟩

4.3 Example model

4.3.1 Preliminaries

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

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

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

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

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

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

lemma *var-renaming-preserves-size*:

fixes

v :: ('r,'l,'v) val **and**

e :: ('r,'l,'v) expr **and**

α :: 'r ⇒ 'r' **and**

β :: 'l ⇒ 'l' **and**

ζ :: 'v ⇒ 'v'

shows

size (map-val α β ζ v) = size v

size (map-expr α β ζ e) = size e

⟨proof⟩

4.3.2 Definition

function

nat-subst_V :: ('r,'l,nat) expr ⇒ nat ⇒ ('r,'l,nat) val ⇒ ('r,'l,nat) expr **and**

nat-subst_E :: ('r,'l,nat) expr ⇒ nat ⇒ ('r,'l,nat) expr ⇒ ('r,'l,nat) expr

where

nat-subst_V e x (CV const) = VE (CV const)

| *nat-subst_V* e x (Var x') = (if x = x' then e else VE (Var x'))

| *nat-subst_V* e x (Loc l) = VE (Loc l)

| *nat-subst_V* e x (Rid r) = VE (Rid r)

| *nat-subst_V* e x (Lambda y e') = VE (

if x = y then

Lambda y e'

else


```

    let z = Suc (Max ( $\mathcal{V}_E e' \cup \mathcal{V}_E e$ )) in
      Lambda z (nat-substE e x ( $\mathcal{R}\mathcal{V}_E$  (id(y := z)) e'))
  | nat-substE e x (VE v') = nat-substV e x v'
  | nat-substE e x (Apply l r) = Apply (nat-substE e x l) (nat-substE e x r)
  | nat-substE e x (Ite e1 e2 e3) = Ite (nat-substE e x e1) (nat-substE e x e2)
  (nat-substE e x e3)
  | nat-substE e x (Ref e') = Ref (nat-substE e x e')
  | nat-substE e x (Read e') = Read (nat-substE e x e')
  | nat-substE e x (Assign l r) = Assign (nat-substE e x l) (nat-substE e x r)
  | nat-substE e x (Rfork e') = Rfork (nat-substE e x e')
  | nat-substE e x (Rjoin e') = Rjoin (nat-substE e x e')
  ⟨proof⟩
termination
  ⟨proof⟩

```

4.3.3 Proof obligations

lemma *nat-subst_E-distr*:

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

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

⟨proof⟩

lemma *nat-subst_E-introduces-no-rids*:

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

shows RID_E (nat-subst_E e x e') \subseteq RID_E e \cup RID_E e'

⟨proof⟩

lemma *nat-subst_E-introduces-no-lids*:

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

shows LID_E (nat-subst_E e x e') \subseteq LID_E e \cup LID_E e'

⟨proof⟩

lemma *nat-subst_E-models-substitution*: substitution nat-subst_E

⟨proof⟩

end

5 Operational Semantics

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

theory *OperationalSemantics*

imports *Substitution*

begin

context *substitution*

begin

5.1 Definition

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

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

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

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

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

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

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

| *fork*: $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e]) \implies r' \notin \text{RID}_G\ s \implies \text{revision-step } r\ s\ (s(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (\text{Rid } r')], r' \mapsto (\sigma;;\tau, \varepsilon, e))))$

| *join*: $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE\ (\text{Rid } r'))]) \implies s\ r' = \text{Some } (\sigma', \tau', VE\ v) \implies \text{revision-step } r\ s\ (s(r := \text{Some } (\sigma, (\tau;;\tau'), \mathcal{E}[VE\ (CV\ \text{Unit})]), r' := \text{None}))$

| *join_ε*: $s\ r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE\ (\text{Rid } r'))]) \implies s\ r' = \text{None} \implies \text{revision-step } r\ s\ \varepsilon$

inductive-cases *revision-stepE* [*elim*, *consumes 1*, *case-names app ifTrue ifFalse new get set fork join join_ε*]:

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 $\text{LID}_G\ s' \subseteq \text{LID}_G\ s$

<proof>

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 $\text{RID}_G\ s' \subseteq \text{RID}_G\ s$

<proof>

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

<proof>

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

<proof>

5.3 Domain subsumption

5.3.1 Definitions

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

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

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

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

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

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

<proof>

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]:

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

<proof>

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]:

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

<proof>

5.3.2 The theorem

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

assumes

\mathcal{S}_G -s: $\mathcal{S}_G \ s$ **and**

r-in-dom: $r \in \text{dom } s$

shows $\mathcal{A} \ r \ r \ s$

<proof>

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>

5.4 Relaxed definition of the operational semantics

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

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

r s ($s(r \mapsto (\sigma, \tau, \mathcal{E}[\text{subst } (VE \ v) \ x \ e]))$)

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

s ($s(r \mapsto (\sigma, \tau, \mathcal{E}[e1]))$)

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

s ($s(r \mapsto (\sigma, \tau, \mathcal{E}[e2]))$)

| *new*: s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Ref } (VE \ v)]) \Longrightarrow l \notin \bigcup \{ \text{doms } ls \mid ls. ls \in \text{ran } s \}$

\Longrightarrow *revision-step-relaxed* r s ($s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE \ (Loc \ l)]))$)

| *get*: s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Read } (VE \ (Loc \ l))]) \Longrightarrow$ *revision-step-relaxed* r s ($s(r$

$\mapsto (\sigma, \tau, \mathcal{E}[VE \ (the \ ((\sigma;;\tau) \ l))])$)

| *set*: s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Assign } (VE \ (Loc \ l)) \ (VE \ v)]) \Longrightarrow$ *revision-step-relaxed*

r s ($s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE \ (CV \ Unit)]))$)

| *fork*: s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rfork } e]) \Longrightarrow r' \notin \text{RID}_G \ s \Longrightarrow$ *revision-step-relaxed* r

s ($s(r \mapsto (\sigma, \tau, \mathcal{E}[VE \ (Rid \ r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e))$)

| *join*: s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE \ (Rid \ r'))]) \Longrightarrow s$ $r' = \text{Some } (\sigma', \tau', VE$

$v) \Longrightarrow$ *revision-step-relaxed* r s ($s(r := \text{Some } (\sigma, (\tau;;\tau'), \mathcal{E}[VE \ (CV \ Unit)]), r' := \text{None})$)

| *join $_\varepsilon$* : s $r = \text{Some } (\sigma, \tau, \mathcal{E}[\text{Rjoin } (VE \ (Rid \ r'))]) \Longrightarrow s$ $r' = \text{None} \Longrightarrow$ *revision-step-relaxed* r s ε

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

revision-step-relaxed r s s'

end

end

6 Executions

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

determinacy.

```
theory Executions
  imports OperationalSemantics
begin
```

```
context substitution
begin
```

6.1 Generalizing the original transition

6.1.1 Definition

```
definition steps :: ('r,'l,'v) global-state rel ([ $\rightsquigarrow$ ]) 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'
  <proof>
```

```
lemma valid-stepE [dest]:
  s  $\rightsquigarrow$  s'  $\Longrightarrow$   $\exists r$ . revision-step r s s'
  <proof>
```

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' <proof>
```

6.2 Properties

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

<proof>

lemma *inductive-invariant-is-execution-invariant*: *reachable s* \Longrightarrow *inductive-invariant P* \Longrightarrow *P s*

<proof>

6.3.2 Subsumption is invariant

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

<proof>

corollary *reachable-imp-S_G*: *reachable s* \Longrightarrow \mathcal{S}_G *s*

<proof>

lemma *transition-relations-equivalent*: *reachable s* \Longrightarrow *revision-step r s s'* = *revision-step-relaxed r s s'*

<proof>

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
 $finite (RID_V v)$
 $finite (RID_E e)$
 $finite (LID_V v)$
 $finite (LID_E e)$
 $\langle proof \rangle$

lemma *store-finite-upd* [intro]:
 $finite (RID_S \tau) \implies finite (RID_S (\tau(l := None)))$
 $finite (LID_S \tau) \implies finite (LID_S (\tau(l := None)))$
 $\langle proof \rangle$

lemma *finite-state-imp-restriction-finite* [intro]:
 $finite (RID_G s) \implies finite (RID_G (s(r := None)))$
 $finite (LID_G s) \implies finite (LID_G (s(r := None)))$
 $\langle proof \rangle$

lemma *local-state-of-finite-restricted-global-state-is-finite* [intro]:
 $s \ r' = Some \ ls \implies finite (RID_G (s(r := None))) \implies r \neq r' \implies finite (RID_L \ ls)$
 $s \ r' = Some \ ls \implies finite (LID_G (s(r := None))) \implies r \neq r' \implies finite (LID_L \ ls)$
 $\langle proof \rangle$

lemma *empty-map-finite* [simp]:
 $finite (RID_S \varepsilon)$
 $finite (LID_S \varepsilon)$
 $finite (RID_G \varepsilon)$
 $finite (LID_G \varepsilon)$
 $\langle proof \rangle$

lemma *finite-combination* [intro]:
 $finite (RID_S \sigma) \implies finite (RID_S \tau) \implies finite (RID_S (\sigma;;\tau))$
 $finite (LID_S \sigma) \implies finite (LID_S \tau) \implies finite (LID_S (\sigma;;\tau))$
 $\langle proof \rangle$

lemma *RID_G-finite-invariant*:
assumes
 $step: revision-step \ r \ s \ s'$ **and**
 $fin: finite (RID_G \ s)$
shows
 $finite (RID_G \ s')$
 $\langle proof \rangle$

lemma *RID_L-finite-invariant*:
assumes
 $step: revision-step \ r \ s \ s'$ **and**
 $fin: finite (LID_G \ s)$
shows
 $finite (LID_G \ s')$

<proof>

lemma *reachable-imp-identifiers-finite:*

assumes *reach: reachable s*

shows

finite (RID_G s)

finite (LID_G s)

<proof>

lemma *reachable-imp-identifiers-available:*

assumes

reachable (s :: ('r,'l,'v) global-state)

shows

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

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

<proof>

6.3.4 Reachability is invariant

lemma *initial-state-reachable:*

assumes *program-expr e*

shows *reachable ($\varepsilon(r \mapsto (\varepsilon, \varepsilon, e))$)*

<proof>

lemma *reachability-closed-under-execution-step:*

assumes

reach: reachable s and

step: revision-step r s s'

shows *reachable s'*

<proof>

lemma *reachability-closed-under-execution: reachable s $\implies s \rightsquigarrow^* s' \implies$ reachable s'*

<proof>

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

shows (*revision-step* $r \ s \ s'$) = ($s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [\text{subst } (\text{VE } v) \ x \ e])))$) (**is** $?l = ?r$)

<proof>

lemma *ifTrue-deterministic* [*simp*]:

assumes

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

shows (*revision-step* $r \ s \ s'$) = ($s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [e1])))$) (**is** $?l = ?r$)

<proof>

lemma *ifFalse-deterministic* [*simp*]:

assumes

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

shows (*revision-step* $r \ s \ s'$) = ($s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [e2])))$) (**is** $?l = ?r$)

<proof>

lemma *new-pseudodeterministic* [*simp*]:

assumes

s-r: $s \ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Ref } (\text{VE } v)])$

shows (*revision-step* $r \ s \ s'$) = ($\exists l. l \notin \text{LID}_G \ s \wedge s' = (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E} [\text{VE } (\text{Loc } l)])))$) (**is** $?l = ?r$)

<proof>

lemma *get-deterministic* [*simp*]:

assumes

s-r: $s \ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Read } (\text{VE } (\text{Loc } l))])$

shows (*revision-step* $r \ s \ s'$) = ($l \in \text{dom } (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [\text{VE } (\text{the } ((\sigma;;\tau) \ l)]))))$) (**is** $?l = ?r$)

<proof>

lemma *set-deterministic* [*simp*]:

assumes

s-r: $s \ r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Assign } (\text{VE } (\text{Loc } l)) (\text{VE } v)])$

shows (*revision-step* $r \ s \ s'$) = ($l \in \text{dom } (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E} [\text{VE } (\text{CV } \text{Unit}]))))$) (**is** $?l = ?r$)

<proof>

lemma *fork-pseudodeterministic* [*simp*]:

assumes

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

shows (*revision-step* $r \ s \ s'$) = ($\exists r'. r' \notin \text{RID}_G \ (s(r \mapsto (\sigma, \tau, \mathcal{E} [\text{Rfork } e])) \wedge s' = (s(r \mapsto (\sigma, \tau, \mathcal{E} [\text{VE } (\text{Rid } r')]), r' \mapsto (\sigma;;\tau, \varepsilon, e))))$) (**is** $?l = ?r$)

<proof>

lemma *rjoin-deterministic* [*simp*]:

assumes

s-r: $s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [R\text{join } (VE (Rid\ r'))])$ **and**

s-r': $s\ r' = \text{Some } (\sigma', \tau', VE\ v)$

shows (*revision-step* $r\ s\ s'$) = ($s' = (s(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E} [VE (CV\ Unit)]), r' := None))$) (**is** $?l = ?r$)

<proof>

lemma *rjoin_ε-deterministic* [*simp*]:

assumes

s-r: $s\ r = \text{Some } (\sigma, \tau, \mathcal{E} [R\text{join } (VE (Rid\ r'))])$ **and**

s-r': $s\ r' = None$

shows (*revision-step* $r\ s\ s'$) = ($s' = \varepsilon$) (**is** $?l = ?r$)

<proof>

7.2 Strong local confluence

7.2.1 Local determinism

lemma *local-determinism*:

assumes

left: *revision-step* $r\ s_1\ s_2$ **and**

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

shows $s_2 \approx s_2'$

<proof>

7.2.2 General principles

lemma *SLC-sym*:

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

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

<proof>

lemma *SLC-commute*:

$\llbracket s_3 = s_3'; \text{revision-step } r' s_2 s_3; \text{revision-step } r s_2' s_3' \rrbracket \implies$

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

<proof>

7.2.3 Case join-epsilon

lemma *SLC-join_ε*:

assumes

s₁-r: $s_1\ r = \text{Some } (\sigma, \tau, \mathcal{E}[R\text{join } (VE (Rid\ r'))])$ **and**

s₂: $s_2 = \varepsilon$ **and**

side: $s_1\ r'' = None$ **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*⟩

7.2.4 Case join

lemma *join-and-local-commute:*

assumes
 $s_2 = s_1(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (CV \text{ Unit})]), r'' := \text{None})$
 $s_2' = s_1(r' \mapsto ls)$
 $r \neq r'$
 $r' \neq r''$
 $\text{revision-step } r' s_2 (s_1(r := \text{Some } (\sigma, \tau;;\tau', \mathcal{E}[\text{VE } (CV \text{ Unit})]), r'' := \text{None}, r' := \text{Some } ls))$
 $s_2' r = \text{Some } (\sigma, \tau, \mathcal{E} [\text{Rjoin } (VE (Rid r''))])$
 $s_2' r'' = \text{Some } (\sigma', \tau', VE v)$
shows
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$
 ⟨*proof*⟩

lemma *SLC-join:*

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

7.2.5 Case local

lemma *local-steps-commute:*

assumes
 $s_2 = s_1(r \mapsto x)$
 $s_2' = s_1(r' \mapsto y)$
 $\text{revision-step } r' (s_1(r \mapsto x)) (s_1(r \mapsto x, r' \mapsto y))$
 $\text{revision-step } r (s_1(r' \mapsto y)) (s_1(r' \mapsto y, r \mapsto x))$
shows
 $\exists s_3 s_3'. s_3 \approx s_3' \wedge (\text{revision-step } r' s_2 s_3 \vee s_2 = s_3) \wedge (\text{revision-step } r s_2' s_3' \vee s_2' = s_3')$
 ⟨*proof*⟩

lemma *local-and-fork-commute:*

assumes

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

shows

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

lemma SLC-app:

assumes

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

shows

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

lemma SLC-ifTrue:

assumes

$s_1\text{-}r: s_1\ r = Some (\sigma, \tau, \mathcal{E}[Ite (VE (CV T)) e1 e2])$ **and**
 $s_2: s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[e1]))$ **and**
 $right: revision\text{-}step\ r'\ s_1\ s_2'$ **and**
 $neg: r \neq r'$

shows

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

lemma SLC-ifFalse:

assumes

$s_1\text{-}r: s_1\ r = Some (\sigma, \tau, \mathcal{E}[Ite (VE (CV F)) e1 e2])$ **and**
 $s_2: s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[e2]))$ **and**
 $right: revision\text{-}step\ r'\ s_1\ s_2'$ **and**
 $neg: r \neq r'$

shows

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

lemma SLC-set:

assumes

$s_1\text{-}r: s_1\ r = Some (\sigma, \tau, \mathcal{E}[Assign (VE (Loc l)) (VE v)])$ **and**

$s_2: s_2 = s_1(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE (CV Unit)]))$ **and**
side: $l \in \text{dom}(\sigma; \tau)$ **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>

lemma *SLC-get:*

assumes

$s_1\text{-}r: s_1 r = \text{Some}(\sigma, \tau, \mathcal{E}[\text{Read}(VE(Loc\ l))])$ **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>

7.2.6 Case new

lemma *SLC-new:*

assumes

$s_1\text{-}r: s_1 r = \text{Some}(\sigma, \tau, \mathcal{E}[\text{Ref}(VE\ v)])$ **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**
neg: $r \neq r'$ **and**
reach: *reachable* $(s_1 :: ('r, 'l, 'v)\ \text{global-state})$ **and**
lid-inf: *infinite* $(UNIV :: 'l\ \text{set})$

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>

7.2.7 Case fork

lemma *SLC-fork:*

assumes

$s_1\text{-}r: s_1 r = \text{Some}(\sigma, \tau, \mathcal{E}[\text{Rfork}\ e])$ **and**
 $s_2: s_2 = (s_1(r \mapsto (\sigma, \tau, \mathcal{E}[VE(\text{Rid}\ \text{left-forkee}])), \text{left-forkee} \mapsto (\sigma; \tau, \epsilon, e)))$ **and**
side: *left-forkee* $\notin RID_G\ s_1$ **and**
right: *revision-step* $r' s_1 s_2'$ **and**
neg: $r \neq r'$ **and**
reach: *reachable* $(s_1 :: ('r, 'l, 'v)\ \text{global-state})$ **and**
rid-inf: *infinite* $(UNIV :: 'r\ \text{set})$

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⟩

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⟩

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⟩

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⟩

7.3.2 Mimicking diagrams

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

declare *bind-eq-Some-conv* [*simp*]

lemma *in-renamed-in-unlabelled-val*:

$\text{bij } \alpha \implies (\alpha r \in \text{RID}_V (\mathcal{R}_V \alpha \beta v)) = (r \in \text{RID}_V v)$
 $\text{bij } \beta \implies (\beta l \in \text{LID}_V (\mathcal{R}_V \alpha \beta v)) = (l \in \text{LID}_V v)$
 <proof>

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

$\text{bij } \alpha \implies (\alpha r \in \text{RID}_E (\mathcal{R}_E \alpha \beta v)) = (r \in \text{RID}_E v)$
 $\text{bij } \beta \implies (\beta l \in \text{LID}_E (\mathcal{R}_E \alpha \beta v)) = (l \in \text{LID}_E v)$
 <proof>

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

assumes

bij- α : $\text{bij } \alpha$ **and**

bij- β : $\text{bij } \beta$

shows

$(\alpha r \in \text{RID}_S (\mathcal{R}_S \alpha \beta \sigma)) = (r \in \text{RID}_S \sigma)$

$(\beta l \in \text{LID}_S (\mathcal{R}_S \alpha \beta \sigma)) = (l \in \text{LID}_S \sigma)$

<proof>

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

assumes

bij- α : $\text{bij } \alpha$ **and**

bij- β : $\text{bij } \beta$

shows

$(\alpha r \in \text{RID}_L (\mathcal{R}_L \alpha \beta ls)) = (r \in \text{RID}_L ls)$

$(\beta l \in \text{LID}_L (\mathcal{R}_L \alpha \beta ls)) = (l \in \text{LID}_L ls)$

<proof>

lemma *in-renamed-in-unlabelled-global:*

assumes

bij- α : $\text{bij } \alpha$ **and**

bij- β : $\text{bij } \beta$

shows

$(\alpha r \in \text{RID}_G (\mathcal{R}_G \alpha \beta s)) = (r \in \text{RID}_G s)$

$(\beta l \in \text{LID}_G (\mathcal{R}_G \alpha \beta s)) = (l \in \text{LID}_G s)$

<proof>

lemma *mimicking:*

assumes

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

bij- α : $\text{bij } \alpha$ **and**

bij- β : $\text{bij } \beta$

shows *revision-step* $(\alpha r) (\mathcal{R}_G \alpha \beta s) (\mathcal{R}_G \alpha \beta s')$

<proof>

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>

lemma *mimic-trans*:

assumes

$s_1\text{-eq-}s_1'$: $s_1 \approx s_1'$ **and**

s_1s_2 : $s_1 \rightsquigarrow^* s_2$

shows $\exists s_2'. s_2 \approx s_2' \wedge s_1' \rightsquigarrow^* s_2'$

<proof>

7.3.3 Strip diagram

lemma *strip-lemma*:

assumes

s_1s_2 : $s_1 \rightsquigarrow^* s_2$ **and**

s_1s_2' : $s_1 \rightsquigarrow^= s_2'$ **and**

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

lid-inf : $\text{infinite}(UNIV :: 'l \text{ set})$ **and**

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

shows $\exists s_3 s_3'. s_3 \approx s_3' \wedge s_2 \rightsquigarrow^* s_3 \wedge s_2' \rightsquigarrow^* s_3'$

<proof>

7.3.4 Confluence diagram

lemma *confluence-modulo-equivalence*:

assumes

s_1s_2 : $s_1 \rightsquigarrow^* s_2$ **and**

s_1s_2' : $s_1' \rightsquigarrow^* s_2'$ **and**

equiv : $s_1 \approx s_1'$ **and**

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

lid-inf : $\text{infinite}(UNIV :: 'l \text{ set})$ **and**

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

shows $\exists s_3 s_3'. s_3 \approx s_3' \wedge s_2 \rightsquigarrow^* s_3 \wedge s_2' \rightsquigarrow^* s_3'$

<proof>

7.4 Determinacy

theorem *determinacy*:

assumes

prog-expr : $\text{program-expr } e$ **and**

$e\text{-terminates-in-}s$: $e \downarrow s$ **and**

$e\text{-terminates-in-}s'$: $e \downarrow s'$ **and**

lid-inf : $\text{infinite}(UNIV :: 'l \text{ set})$ **and**

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

shows $s \approx s'$

<proof>

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>.