Formalization of Concurrent Revisions

Roy Overbeek

October 26, 2026

Abstract

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

Contents

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

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

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

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

lemma *dom-combination-dom-union*: $dom\ (\tau;;\tau') = dom\ \tau \cup dom\ \tau'$
  $\langle proof \rangle$

1.2 Values, expressions and execution contexts

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

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

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

1.3 Plugging and decomposing

fun *plug* :: $('r,'l,'v)$ *cntxt* $\Rightarrow$ $('r,'l,'v)$ *expr* $\Rightarrow$ $('r,'l,'v)$ *expr* (**infix** $\langle\lhd\rangle$ *60*) **where**
  $\Box \lhd e = e$
| $ApplyL_{\mathcal{E}}\ \mathcal{E}\ e1 \lhd e = Apply\ (\mathcal{E} \lhd e)\ e1$
| $ApplyR_{\mathcal{E}}\ val\ \mathcal{E} \lhd e = Apply\ (VE\ val)\ (\mathcal{E} \lhd e)$
| $Ite_{\mathcal{E}}\ \mathcal{E}\ e1\ e2 \lhd e = Ite\ (\mathcal{E} \lhd e)\ e1\ e2$
| $Ref_{\mathcal{E}}\ \mathcal{E} \lhd e = Ref\ (\mathcal{E} \lhd e)$
| $Read_{\mathcal{E}}\ \mathcal{E} \lhd e = Read\ (\mathcal{E} \lhd e)$
| $AssignL_{\mathcal{E}}\ \mathcal{E}\ e1 \lhd e = Assign\ (\mathcal{E} \lhd e)\ e1$
| $AssignR_{\mathcal{E}}\ l\ \mathcal{E} \lhd e = Assign\ (VE\ (Loc\ l))\ (\mathcal{E} \lhd e)$
| $Rjoin_{\mathcal{E}}\ \mathcal{E} \lhd e = Rjoin\ (\mathcal{E} \lhd e)$

translations
  $\mathcal{E}[x] \rightleftharpoons \mathcal{E} \lhd x$

lemma *injective-cntxt* [*simp*]: $(\mathcal{E}[e1] = \mathcal{E}[e2]) = (e1 = e2)$ $\langle proof \rangle$

lemma *VE-empty-cntxt* [*simp*]: $(VE\ v = \mathcal{E}[e]) = (\mathcal{E} = \Box \wedge VE\ v = e)$ $\langle proof \rangle$

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

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

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

inductive *decompose* :: $('r,'l,'v)$ *expr* $\Rightarrow$ $('r,'l,'v)$ *cntxt* $\Rightarrow$ $('r,'l,'v)$ *expr* $\Rightarrow$ *bool* **where**
  top-redex: $redex\ e \Longrightarrow decompose\ e\ \Box\ e$
| *lapply*: $[\![\ \neg redex\ (Apply\ e_1\ e_2);\ decompose\ e_1\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Apply\ e_1\ e_2)\ (ApplyL_{\mathcal{E}}\ \mathcal{E}\ e_2)\ r$
| *rapply*: $[\![\ \neg redex\ (Apply\ (VE\ v)\ e);\ decompose\ e\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Apply\ (VE\ v)\ e)\ (ApplyR_{\mathcal{E}}\ v\ \mathcal{E})\ r$
| *ite*: $[\![\ \neg redex\ (Ite\ e_1\ e_2\ e_3);\ decompose\ e_1\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Ite\ e_1\ e_2\ e_3)\ (Ite_{\mathcal{E}}\ \mathcal{E}\ e_2\ e_3)\ r$
| *ref*: $[\![\ \neg redex\ (Ref\ e);\ decompose\ e\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Ref\ e)\ (Ref_{\mathcal{E}}\ \mathcal{E})\ r$
| *read*: $[\![\ \neg redex\ (Read\ e);\ decompose\ e\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Read\ e)\ (Read_{\mathcal{E}}\ \mathcal{E})\ r$
| *lassign*: $[\![\ \neg redex\ (Assign\ e_1\ e_2);\ decompose\ e_1\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Assign\ e_1\ e_2)\ (AssignL_{\mathcal{E}}\ \mathcal{E}\ e_2)\ r$
| *rassign*: $[\![\ \neg redex\ (Assign\ (VE\ (Loc\ l))\ e_2);\ decompose\ e_2\ \mathcal{E}\ r\ ]\!] \Longrightarrow decompose\ (Assign\ (VE\ (Loc\ l))\ e_2)\ (AssignR_{\mathcal{E}}\ l\ \mathcal{E})\ r$

| *rjoin*: $[\![\neg \mathit{redex}\ (\mathit{Rjoin}\ e);\ \mathit{decompose}\ e\ \mathcal{E}\ r\ ]\!] \Longrightarrow \mathit{decompose}\ (\mathit{Rjoin}\ e)\ (\mathit{Rjoin}_{\mathcal{E}}\ \mathcal{E})\ r$

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

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

lemma *unique-decomposition*: *decompose e* $\mathcal{E}_1$ $r_1 \Longrightarrow$ *decompose e* $\mathcal{E}_2$ $r_2 \Longrightarrow \mathcal{E}_1 = \mathcal{E}_2 \wedge r_1 = r_2$
⟨*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* ⇀ ('*r*,'*l*,'*v*) *val*
type-synonym ('*r*,'*l*,'*v*) *local-state* = ('*r*,'*l*,'*v*) *store* × ('*r*,'*l*,'*v*) *store* × ('*r*,'*l*,'*v*) *expr*
type-synonym ('*r*,'*l*,'*v*) *global-state* = '*r* ⇀ ('*r*,'*l*,'*v*) *local-state*

fun *doms* :: ('*r*,'*l*,'*v*) *local-state* ⇒ '*l set* **where**
doms (σ,τ,*e*) = *dom* σ ∪ *dom* τ

fun *LID-snapshot* :: ('*r*,'*l*,'*v*) *local-state* ⇒ ('*r*,'*l*,'*v*) *store* (‹-$_\sigma$› *200*) **where**
LID-snapshot (σ,τ,*e*) = σ

fun *LID-local-store* :: ('*r*,'*l*,'*v*) *local-state* ⇒ ('*r*,'*l*,'*v*) *store* (‹-$_\tau$› *200*) **where**
LID-local-store (σ,τ,*e*) = τ

fun *LID-expression* :: ('*r*,'*l*,'*v*) *local-state* ⇒ ('*r*,'*l*,'*v*) *expr* (‹-$_e$› *200*) **where**
LID-expression (σ,τ,*e*) = *e*

end

2 Occurrences

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

theory *Occurrences*
imports *Data*
begin

2.1 Definitions

2.1.1 Revision identifiers

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

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

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

2.1.2 Location identifiers

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

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

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

2.2 Inference rules

2.2.1 Introduction and elimination rules

lemma RID_SI [*intro*]: $\sigma\ l = Some\ v \Longrightarrow r \in RID_V\ v \Longrightarrow r \in RID_S\ \sigma$
⟨*proof*⟩

lemma RID_SE [*elim*]: $r \in RID_S\ \sigma \Longrightarrow (\bigwedge l\ v.\ \sigma\ l = Some\ v \Longrightarrow r \in RID_V\ v \Longrightarrow P) \Longrightarrow P$
⟨*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*]:
$[\![\ r \in RID_L\ s;\ (\bigwedge \sigma\ \tau\ e.\ s = (\sigma,\ \tau,\ e) \Longrightarrow (r \in RID_S\ \sigma \Longrightarrow P) \Longrightarrow (r \in RID_S\ \tau \Longrightarrow P) \Longrightarrow (r \in RID_E\ e \Longrightarrow P) \Longrightarrow P) \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma RID_GI [*intro*]:
$s\ r = Some\ v \Longrightarrow r \in RID_G\ s$
$s\ r' = Some\ ls \Longrightarrow r \in RID_L\ ls \Longrightarrow r \in RID_G\ s$

⟨*proof*⟩

lemma *RID*$_G$*E* [*elim*]:
assumes
r ∈ *RID*$_G$ *s*
r ∈ *dom s* ⟹ *P*
⋀*r*′ *ls*. *s r*′ = *Some ls* ⟹ *r* ∈ *RID*$_L$ *ls* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *LID*$_S$*I* [*intro*]:
σ l = *Some v* ⟹ *l* ∈ *LID*$_S$ *σ*
σ l′ = *Some v* ⟹ *l* ∈ *LID*$_V$ *v* ⟹ *l* ∈ *LID*$_S$ *σ*
⟨*proof*⟩

lemma *LID*$_S$*E* [*elim*]:
assumes
l ∈ *LID*$_S$ *σ*
l ∈ *dom σ* ⟹ *P*
⋀*l*′ *v*. *σ l*′ = *Some v* ⟹ *l* ∈ *LID*$_V$ *v* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *LID*$_L$*I* [*intro*]:
assumes *s* = (*σ*, *τ*, *e*)
shows
r ∈ *LID*$_S$ *σ* ⟹ *r* ∈ *LID*$_L$ *s*
r ∈ *LID*$_S$ *τ* ⟹ *r* ∈ *LID*$_L$ *s*
r ∈ *LID*$_E$ *e* ⟹ *r* ∈ *LID*$_L$ *s*
⟨*proof*⟩

lemma *LID*$_L$*E* [*elim*]:
⟦ *l* ∈ *LID*$_L$ *s*; (⋀*σ τ e*. *s* = (*σ*, *τ*, *e*) ⟹ (*l* ∈ *LID*$_S$ *σ* ⟹ *P*) ⟹ (*l* ∈ *LID*$_S$ *τ* ⟹ *P*) ⟹ (*l* ∈ *LID*$_E$ *e* ⟹ *P*) ⟹ *P*) ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *LID*$_G$*I* [*intro*]: *s r* = *Some ls* ⟹ *l* ∈ *LID*$_L$ *ls* ⟹ *l* ∈ *LID*$_G$ *s*
⟨*proof*⟩

lemma *LID*$_G$*E* [*elim*]: *l* ∈ *LID*$_G$ *s* ⟹ (⋀*r ls*. *s r* = *Some ls* ⟹ *l* ∈ *LID*$_L$ *ls* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

2.2.2 Distributive laws

lemma *ID-distr-completion* [*simp*]:
RID$_E$ (*E*[*e*]) = *RID*$_C$ *E* ∪ *RID*$_E$ *e*
LID$_E$ (*E*[*e*]) = *LID*$_C$ *E* ∪ *LID*$_E$ *e*
⟨*proof*⟩

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*⟩

lemma *in-restricted-store-in-unrestricted-store* [*intro*]:
$r \in RID_S\ (\sigma(l := None)) \Longrightarrow r \in RID_S\ \sigma$
$l' \in LID_S\ (\sigma(l := None)) \Longrightarrow l' \in LID_S\ \sigma$
⟨*proof*⟩

lemma *in-restricted-store-in-updated-store* [*intro*]:
$r \in RID_S\ (\sigma(l := None)) \Longrightarrow r \in RID_S\ (\sigma(l \mapsto v))$
$l' \in LID_S\ (\sigma(l := None)) \Longrightarrow l' \in LID_S\ (\sigma(l \mapsto v))$
⟨*proof*⟩

lemma *ID-distr-store* [*simp*]:
$RID_S\ (\tau(l \mapsto v)) = RID_S\ (\tau(l := None)) \cup RID_V\ v$
$LID_S\ (\tau(l \mapsto v)) = insert\ l\ (LID_S\ (\tau(l := None)) \cup LID_V\ v)$
⟨*proof*⟩

lemma *ID-distr-local* [*simp*]:
$LID_L\ (\sigma,\tau,e) = LID_S\ \sigma \cup LID_S\ \tau \cup LID_E\ e$
$RID_L\ (\sigma,\tau,e) = RID_S\ \sigma \cup RID_S\ \tau \cup RID_E\ e$
⟨*proof*⟩

lemma *ID-restricted-global-subset-unrestricted*:
$LID_G\ (s(r := None)) \subseteq LID_G\ s$
$RID_G\ (s(r := None)) \subseteq RID_G\ s$
⟨*proof*⟩

lemma *in-restricted-global-in-unrestricted-global* [*intro*]:
$r' \in RID_G\ (s(r := None)) \Longrightarrow r' \in RID_G\ s$
$l \in LID_G\ (s(r := None)) \Longrightarrow l \in LID_G\ s$
⟨*proof*⟩

lemma *in-restricted-global-in-updated-global* [*intro*]:
$r' \in RID_G\ (s(r := None)) \Longrightarrow r' \in RID_G\ (s(r \mapsto ls))$
$l \in LID_G\ (s(r := None)) \Longrightarrow l \in LID_G\ (s(r \mapsto ls))$
⟨*proof*⟩

lemma *ID-distr-global* [*simp*]:
$RID_G\ (s(r \mapsto ls)) = insert\ r\ (RID_G\ (s(r := None)) \cup RID_L\ ls)$
$LID_G\ (s(r \mapsto ls)) = LID_G\ (s(r := None)) \cup LID_L\ ls$
⟨*proof*⟩

lemma *restrictions-inwards* [*simp*]:
$x \neq x' \Longrightarrow f(x := Some\ y,\ x' := None) = (f(x' := None,\ x := Some\ y))$
⟨*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*]:
$r \in RID_S\ (\sigma;;\tau) \Longrightarrow r \notin RID_S\ \sigma \Longrightarrow r \in RID_S\ \tau$
$r \in RID_S\ (\sigma;;\tau) \Longrightarrow r \notin RID_S\ \tau \Longrightarrow r \in RID_S\ \sigma$
$l \in LID_S\ (\sigma;;\tau) \Longrightarrow l \notin LID_S\ \sigma \Longrightarrow l \in LID_S\ \tau$
$l \in LID_S\ (\sigma;;\tau) \Longrightarrow l \notin LID_S\ \tau \Longrightarrow l \in LID_S\ \sigma$
⟨*proof*⟩

lemma *in-mapped-in-component-of-combination* [*intro*]:
assumes
maps-to-v: $(\sigma;;\tau)\ l = Some\ v$ **and**
l'-in-v: $l' \in LID_V\ v$
shows
$l' \notin LID_S\ \tau \Longrightarrow l' \in LID_S\ \sigma$
$l' \notin LID_S\ \sigma \Longrightarrow l' \in LID_S\ \tau$
⟨*proof*⟩

lemma *elim-trivial-restriction* [*simp*]: $l \notin LID_S\ \tau \Longrightarrow (\tau(l := None)) = \tau$
⟨*proof*⟩

lemma *ID-distr-global-conditional*:
$s\ r = Some\ ls \Longrightarrow RID_G\ s = insert\ r\ (RID_G\ (s(r := None)) \cup RID_L\ ls)$
$s\ r = Some\ ls \Longrightarrow LID_G\ s = LID_G\ (s(r := None)) \cup LID_L\ ls$
⟨*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* :: $('r \Rightarrow 'r) \Rightarrow ('l \Rightarrow 'l) \Rightarrow ('r,'l,'v)\ val \Rightarrow ('r,'l,'v)\ val$ $(\langle\mathcal{R}_V\rangle)$ **where**
$\mathcal{R}_V\ \alpha\ \beta\ v \equiv map\text{-}val\ \alpha\ \beta\ id\ v$

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

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

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

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

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

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

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

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

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

3.2 Introduction rules

lemma $\mathcal{R}_S I$ [*intro*]:
assumes
bij-β: *bij* β **and**
none-case: $\bigwedge l.\ \sigma\ l =$ *None* $\Longrightarrow \sigma'\ (\beta\ l) =$ *None* **and**
some-case: $\bigwedge l\ v.\ \sigma\ l =$ *Some* $v \Longrightarrow \sigma'\ (\beta\ l) =$ *Some* $(\mathcal{R}_V\ \alpha\ \beta\ v)$
shows
$\mathcal{R}_S\ \alpha\ \beta\ \sigma = \sigma'$
⟨*proof*⟩

lemma $\mathcal{R}_G I$ [*intro*]:
assumes
bij-α: *bij* α **and**
none-case: $\bigwedge r.\ s\ r = None \Longrightarrow s'\ (\alpha\ r) = None$ **and**
some-case: $\bigwedge r\ \sigma\ \tau\ e.\ s\ r = Some\ (\sigma,\tau,e) \Longrightarrow s'\ (\alpha\ r) = Some\ (\mathcal{R}_L\ \alpha\ \beta\ (\sigma,\tau,e))$
shows
$\mathcal{R}_G\ \alpha\ \beta\ s = s'$
⟨*proof*⟩

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*]: $[\![\ bij\ \beta;\ bij\ \beta'\ ]\!] \Longrightarrow \mathcal{R}_S\ \alpha'\ \beta'\ (\mathcal{R}_S\ \alpha\ \beta\ s) = \mathcal{R}_S\ (\alpha' \circ \alpha)\ (\beta' \circ \beta)\ s$
⟨*proof*⟩
lemma $\mathcal{R}_L$-*comp* [*simp*]: $[\![\ bij\ \beta;\ bij\ \beta'\ ]\!] \Longrightarrow \mathcal{R}_L\ \alpha'\ \beta'\ (\mathcal{R}_L\ \alpha\ \beta\ ls) = \mathcal{R}_L\ (\alpha' \circ \alpha)\ (\beta' \circ \beta)\ ls$
⟨*proof*⟩
lemma $\mathcal{R}_G$-*comp* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \alpha';\ bij\ \beta;\ bij\ \beta'\ ]\!] \Longrightarrow \mathcal{R}_G\ \alpha'\ \beta'\ (\mathcal{R}_G\ \alpha\ \beta\ s) = \mathcal{R}_G\ (\alpha' \circ \alpha)\ (\beta' \circ \beta)\ s$
⟨*proof*⟩

3.3.3 Inverse

lemma $\mathcal{R}_V$-*inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_V\ (inv\ \alpha)\ (inv\ \beta)\ v' = v) = (\mathcal{R}_V\ \alpha\ \beta\ v = v')$
⟨*proof*⟩
lemma $\mathcal{R}_E$-*inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_E\ (inv\ \alpha)\ (inv\ \beta)\ e' = e) = (\mathcal{R}_E\ \alpha\ \beta\ e = e')$
⟨*proof*⟩
lemma $\mathcal{R}_C$-*inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_C\ (inv\ \alpha)\ (inv\ \beta)\ \mathcal{E}' = \mathcal{E}) = (\mathcal{R}_C\ \alpha\ \beta\ \mathcal{E} = \mathcal{E}')$
⟨*proof*⟩
lemma $\mathcal{R}_S$-*inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_S\ (inv\ \alpha)\ (inv\ \beta)\ \sigma' = \sigma) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma = \sigma')$
⟨*proof*⟩

lemma $\mathcal{R}_L$*-inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_L\ (inv\ \alpha)\ (inv\ \beta)\ ls' = ls) = (\mathcal{R}_L\ \alpha\ \beta\ ls = ls')$
⟨*proof*⟩
lemma $\mathcal{R}_G$*-inv* [*simp*]: $[\![\ bij\ \alpha;\ bij\ \beta\ ]\!] \Longrightarrow (\mathcal{R}_G\ (inv\ \alpha)\ (inv\ \beta)\ s' = s) = (\mathcal{R}_G\ \alpha\ \beta\ s = s')$
⟨*proof*⟩

3.3.4 Equivalence

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

lemma *eq-statesI* [*intro*]:
$\mathcal{R}_G\ \alpha\ \beta\ s = s' \Longrightarrow bij\ \alpha \Longrightarrow bij\ \beta \Longrightarrow s \approx s'$
⟨*proof*⟩

lemma *eq-statesE* [*elim*]:
$s \approx s' \Longrightarrow (\bigwedge \alpha\ \beta.\ \mathcal{R}_G\ \alpha\ \beta\ s = s' \Longrightarrow bij\ \alpha \Longrightarrow bij\ \beta \Longrightarrow P) \Longrightarrow P$
⟨*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 *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 *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 *renaming-distr-store* [*simp*]:
$bij\ \beta \Longrightarrow \mathcal{R}_S\ \alpha\ \beta\ (\sigma(l \mapsto v)) = (\mathcal{R}_S\ \alpha\ \beta\ \sigma)(\beta\ l \mapsto \mathcal{R}_V\ \alpha\ \beta\ v)$
⟨*proof*⟩

3.4.3 Global

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

bij $\alpha \Longrightarrow \mathcal{R}_G\ \alpha\ \beta\ (s(r \mapsto ls)) = (\mathcal{R}_G\ \alpha\ \beta\ s)(\alpha\ r \mapsto \mathcal{R}_L\ \alpha\ \beta\ ls)$
bij $\alpha \Longrightarrow \mathcal{R}_G\ \alpha\ \beta\ (s(r := None)) = (\mathcal{R}_G\ \alpha\ \beta\ s)(\alpha\ r := None)$
⟨*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*:
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
$v :: ('r,'l,'v)\ val$ **and**
$e :: ('r,'l,'v)\ expr$
shows
$l \notin LID_V\ v \Longrightarrow \mathcal{R}_V\ \alpha\ (\beta(l := l'))\ v = \mathcal{R}_V\ \alpha\ \beta\ v$
$l \notin LID_E\ e \Longrightarrow \mathcal{R}_E\ \alpha\ (\beta(l := l'))\ e = \mathcal{R}_E\ \alpha\ \beta\ e$
$r \notin RID_V\ v \Longrightarrow \mathcal{R}_V\ (\alpha(r := r'))\ \beta\ v = \mathcal{R}_V\ \alpha\ \beta\ v$
$r \notin RID_E\ e \Longrightarrow \mathcal{R}_E\ (\alpha(r := r'))\ \beta\ e = \mathcal{R}_E\ \alpha\ \beta\ e$
⟨*proof*⟩

lemma *eliminate-renaming-cntxt* [*simp*]:
$r \notin RID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ (\alpha(r := r'))\ \beta\ \mathcal{E} = \mathcal{R}_C\ \alpha\ \beta\ \mathcal{E}$
$l \notin LID_C\ \mathcal{E} \Longrightarrow \mathcal{R}_C\ \alpha\ (\beta(l := l'))\ \mathcal{E} = \mathcal{R}_C\ \alpha\ \beta\ \mathcal{E}$
⟨*proof*⟩

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

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

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

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

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

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

lemma *eliminate-swap-global-lid* [*simp*, *intro*]:
$l \notin LID_G\ s \Longrightarrow l' \notin LID_G\ s \Longrightarrow \mathcal{R}_G\ id\ (id(l := l',\ l' := l))\ s = s$
⟨*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*]: $r \in RID_E\ (subst\ (VE\ v)\ x\ e) \Longrightarrow r \notin RID_E\ e \Longrightarrow r \in RID_V\ v$
⟨*proof*⟩

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

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

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

lemma *var-renaming-preserves-size*:
fixes
v :: $('r,'l,'v)$ *val* **and**
e :: $('r,'l,'v)$ *expr* **and**
α :: $'r \Rightarrow 'r'$ **and**
β :: $'l \Rightarrow 'l'$ **and**
ζ :: $'v \Rightarrow 'v'$
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* $\Rightarrow$ *nat* $\Rightarrow$ $('r,'l,nat)$ *val* $\Rightarrow$ $('r,'l,nat)$ *expr* **and**
nat-subst$_E$ :: $('r,'l,nat)$ *expr* $\Rightarrow$ *nat* $\Rightarrow$ $('r,'l,nat)$ *expr* $\Rightarrow$ $('r,'l,nat)$ *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' ∪ $\mathcal{V}_E$ e)) *in*
Lambda z (*nat-subst*$_E$ e x ($\mathcal{RV}_E$ (*id*(y := z)) e')))
| *nat-subst*$_E$ e x (*VE* v') = *nat-subst*$_V$ e x v'
| *nat-subst*$_E$ e x (*Apply* l r) = *Apply* (*nat-subst*$_E$ e x l) (*nat-subst*$_E$ e x r)
| *nat-subst*$_E$ e x (*Ite e1 e2 e3*) = *Ite* (*nat-subst*$_E$ e x *e1*) (*nat-subst*$_E$ e x *e2*)
(*nat-subst*$_E$ e x *e3*)
| *nat-subst*$_E$ e x (*Ref* e') = *Ref* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Read* e') = *Read* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Assign* l r) = *Assign* (*nat-subst*$_E$ e x l) (*nat-subst*$_E$ e x r)
| *nat-subst*$_E$ e x (*Rfork* e') = *Rfork* (*nat-subst*$_E$ e x e')
| *nat-subst*$_E$ e x (*Rjoin* e') = *Rjoin* (*nat-subst*$_E$ e x e')
⟨*proof*⟩
termination
⟨*proof*⟩

4.3.3 Proof obligations

lemma *nat-subst*$_E$*-distr*:
fixes e :: ('*r*,'*l*,*nat*) *expr*
shows $\mathcal{R}_E$ α β (*nat-subst*$_E$ e x e') = *nat-subst*$_E$ ($\mathcal{R}_E$ α β e) x ($\mathcal{R}_E$ α β e')
⟨*proof*⟩

lemma *nat-subst*$_E$*-introduces-no-rids*:
fixes e' :: ('*r*,'*l*,*nat*) *expr*
shows *RID*$_E$ (*nat-subst*$_E$ e x e') ⊆ *RID*$_E$ e ∪ *RID*$_E$ e'
⟨*proof*⟩

lemma *nat-subst*$_E$*-introduces-no-lids*:
fixes e' :: ('*r*,'*l*,*nat*) *expr*
shows *LID*$_E$ (*nat-subst*$_E$ e x e') ⊆ *LID*$_E$ e ∪ *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* :: $'r \Rightarrow ('r,'l,'v)$ *global-state* $\Rightarrow ('r,'l,'v)$ *global-state* $\Rightarrow$ *bool*
where
app: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Apply\ (VE\ (Lambda\ x\ e))\ (VE\ v)]) \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[subst\ (VE\ v)\ x\ e])))$
| *ifTrue*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Ite\ (VE\ (CV\ T))\ e1\ e2]) \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[e1])))$
| *ifFalse*: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Ite\ (VE\ (CV\ F))\ e1\ e2]) \Longrightarrow$ *revision-step* $r\ s\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[e2])))$

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

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

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

5.2 Introduction lemmas for identifiers

lemma *only-new-introduces-lids* [*intro*, *dest*]:
assumes
step: *revision-step* $r\ s\ s'$ **and**
not-new: $\bigwedge\sigma\ \tau\ \mathcal{E}\ v.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Ref\ (VE\ v)])$
shows $LID_G\ s' \subseteq LID_G\ s$
⟨*proof*⟩

lemma *only-fork-introduces-rids* [*intro*, *dest*]:
assumes
step: *revision-step* $r\ s\ s'$ **and**
not-fork: $\bigwedge\sigma\ \tau\ \mathcal{E}\ e.\ s\ r \neq Some\ (\sigma,\tau,\mathcal{E}[Rfork\ e])$
shows $RID_G\ s' \subseteq RID_G\ s$
⟨*proof*⟩

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

⟨*proof*⟩

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

5.3 Domain subsumption

5.3.1 Definitions

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

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

lemma *domains-subsume-globallyI* [*intro*]:
$(\bigwedge r\ \sigma\ \tau\ e.\ s\ r = Some\ (\sigma,\tau,e) \Longrightarrow \mathcal{S}\ (\sigma,\tau,e)) \Longrightarrow domains\text{-}subsume\text{-}globally\ s$
⟨*proof*⟩

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

lemma *subsumes-accessibleI* [*intro*]:
$(r_2 \in RID_L\ (the\ (s\ r_1)) \Longrightarrow LID_S\ ((the\ (s\ r_2))_\sigma) \subseteq doms\ (the\ (s\ r_1))) \Longrightarrow \mathcal{A}\ r_1\ r_2\ s$
⟨*proof*⟩

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

lemma *subsumes-accessible-globallyI* [*intro*]:
$(\bigwedge r_1\ \sigma_1\ \tau_1\ e_1\ r_2\ \sigma_2\ \tau_2\ e_2.\ s\ r_1 = Some\ (\sigma_1,\tau_1,e_1) \Longrightarrow s\ r_2 = Some\ (\sigma_2,\tau_2,e_2) \Longrightarrow \mathcal{A}\ r_1\ r_2\ s) \Longrightarrow \mathcal{A}_G\ s$
⟨*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: $r \in dom\ s$
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* ⇒ (*'r*,*'l*,*'v*) *global-state* ⇒ (*'r*,*'l*,*'v*) *global-state* ⇒ *bool* **where**
app: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Apply* (*VE* (*Lambda x e*)) (*VE v*)]) ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*, *$\mathcal{E}$*[*subst* (*VE v*) *x e*])))
| *ifTrue*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Ite* (*VE* (*CV T*)) *e1 e2*]) ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*, *$\mathcal{E}$*[*e1*])))
| *ifFalse*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Ite* (*VE* (*CV F*)) *e1 e2*]) ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*, *$\mathcal{E}$*[*e2*])))

| *new*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Ref* (*VE v*)]) ⟹ *l* ∉ ⋃ { *doms ls* | *ls*. *ls* ∈ *ran s* } ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*(*l* ↦ *v*), *$\mathcal{E}$*[*VE* (*Loc l*)])))
| *get*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Read* (*VE* (*Loc l*))]) ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*, *$\mathcal{E}$*[*VE* (*the* ((*σ*;;*τ*) *l*))])))
| *set*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Assign* (*VE* (*Loc l*)) (*VE v*)]) ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*(*l* ↦ *v*), *$\mathcal{E}$*[*VE* (*CV Unit*)])))

| *fork*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Rfork e*]) ⟹ *r'* ∉ *RID_G s* ⟹ *revision-step-relaxed r s* (*s*(*r* ↦ (*σ*, *τ*, *$\mathcal{E}$*[*VE* (*Rid r'*)]), *r'* ↦ (*σ*;;*τ*, *ε*, *e*)))
| *join*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Rjoin* (*VE* (*Rid r'*))]) ⟹ *s r'* = *Some* (*σ'*, *τ'*, *VE v*) ⟹ *revision-step-relaxed r s* (*s*(*r* := *Some* (*σ*, (*τ*;;*τ'*), *$\mathcal{E}$*[*VE* (*CV Unit*)]), *r'* := *None*))
| *join$_\varepsilon$*: *s r* = *Some* (*σ*, *τ*, *$\mathcal{E}$*[*Rjoin* (*VE* (*Rid r'*))]) ⟹ *s r'* = *None* ⟹ *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* $(\langle[\rightsquigarrow]\rangle)$ **where**
  steps = { (s,s') | s s'. $\exists\, r$. *revision-step* r s s' }

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

lemma *valid-stepI* [*intro*]:
  revision-step r s $s' \Longrightarrow s \rightsquigarrow s'$
  $\langle proof \rangle$

lemma *valid-stepE* [*dest*]:
  $s \rightsquigarrow s' \Longrightarrow \exists\, r$. *revision-step* r s s'
  $\langle proof \rangle$

6.1.2 Closures

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

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

lemma *refl-rewritesI* [*intro*]: $s \rightsquigarrow s' \Longrightarrow s \rightsquigarrow^= s'$ $\langle proof \rangle$

6.2 Properties

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

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

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

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

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

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

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

6.3 Invariants

6.3.1 Inductive invariance

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

lemma *inductive-invariantI* [*intro*]:
$(\bigwedge s.$ *initial-state s* $\Longrightarrow$ *P s*$) \Longrightarrow (\bigwedge s\ s'.\ s \rightsquigarrow s' \Longrightarrow P\ s \Longrightarrow P\ s') \Longrightarrow$ *inductive-invariant P*
$\langle proof \rangle$

lemma *inductive-invariant-is-execution-invariant*: *reachable s* $\Longrightarrow$ *inductive-invariant P* $\Longrightarrow$ *P s*
$\langle proof \rangle$

6.3.2 Subsumption is invariant

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

corollary *reachable-imp-*$\mathcal{S}_G$: *reachable s* $\Longrightarrow \mathcal{S}_G\ s$
$\langle proof \rangle$

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

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*⟩

lemma *store-finite-upd* [*intro*]:
finite (RID_S τ) ⟹ *finite* (RID_S ($\tau(l := None)$))
finite (LID_S τ) ⟹ *finite* (LID_S ($\tau(l := None)$))
⟨*proof*⟩

lemma *finite-state-imp-restriction-finite* [*intro*]:
finite (RID_G s) ⟹ *finite* (RID_G ($s(r := None)$))
finite (LID_G s) ⟹ *finite* (LID_G ($s(r := None)$))
⟨*proof*⟩

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

lemma *empty-map-finite* [*simp*]:
finite (RID_S ε)
finite (LID_S ε)
finite (RID_G ε)
finite (LID_G ε)
⟨*proof*⟩

lemma *finite-combination* [*intro*]:
finite (RID_S σ) ⟹ *finite* (RID_S τ) ⟹ *finite* (RID_S ($\sigma;;\tau$))
finite (LID_S σ) ⟹ *finite* (LID_S τ) ⟹ *finite* (LID_S ($\sigma;;\tau$))
⟨*proof*⟩

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

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*) ⟹ ∃ *r*. *r* ∉ *RID*$_G$ *s*
infinite (*UNIV* :: '*l set*) ⟹ ∃ *l*. *l* ∉ *LID*$_G$ *s*
⟨*proof*⟩

6.3.4 Reachability is invariant

lemma *initial-state-reachable*:
assumes *program-expr e*
shows *reachable* (ε(*r* ↦ (ε,ε,*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* ⟹ *s* ⇝* *s*' ⟹ *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: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Apply\ (VE\ (Lambda\ x\ e))\ (VE\ v)])$
shows $(revision\text{-}step\ r\ s\ s') = (s' = (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [subst\ (VE\ v)\ x\ e]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *ifTrue-deterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Ite\ (VE\ (CV\ T))\ e1\ e2])$
shows $(revision\text{-}step\ r\ s\ s') = (s' = (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [e1]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *ifFalse-deterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Ite\ (VE\ (CV\ F))\ e1\ e2])$
shows $(revision\text{-}step\ r\ s\ s') = (s' = (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [e2]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *new-pseudodeterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Ref\ (VE\ v)])$
shows $(revision\text{-}step\ r\ s\ s') = (\exists\, l.\ l \notin LID_G\ s \wedge s' = (s(r \mapsto (\sigma,\ \tau(l \mapsto v),\ \mathcal{E}\ [VE\ (Loc\ l)]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *get-deterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Read\ (VE\ (Loc\ l))])$
shows $(revision\text{-}step\ r\ s\ s') = (l \in dom\ (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (the\ ((\sigma;;\tau)\ l))]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *set-deterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Assign\ (VE\ (Loc\ l))\ (VE\ v)])$
shows $(revision\text{-}step\ r\ s\ s') = (l \in dom\ (\sigma;;\tau) \wedge s' = (s(r \mapsto (\sigma,\ \tau(l \mapsto v),\ \mathcal{E}\ [VE\ (CV\ Unit)]))))$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fork-pseudodeterministic* [*simp*]:
assumes
s-r: $s\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}\ [Rfork\ e])$
shows $(revision\text{-}step\ r\ s\ s') = (\exists\, r'.\ r' \notin RID_G\ (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [Rfork\ e]))) \wedge s' = (s(r \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r')]),\ r' \mapsto (\sigma;;\tau,\ \varepsilon,\ e))))$ (**is** *?l* = *?r*)

⟨*proof*⟩

lemma *rjoin-deterministic* [*simp*]:
assumes
s-r: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$ [*Rjoin* (*VE* (*Rid* r'))]) **and**
s-r': $s\ r'$ = *Some* (σ', τ', *VE* v)
shows (*revision-step* $r\ s\ s'$) = (s' = (s(r := *Some* (σ, τ;;τ', $\mathcal{E}$ [*VE* (*CV Unit*)]), r' := *None*))) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *rjoin*$_\varepsilon$*-deterministic* [*simp*]:
assumes
s-r: $s\ r$ = *Some* (σ, τ, $\mathcal{E}$ [*Rjoin* (*VE* (*Rid* r'))]) **and**
s-r': $s\ r'$ = *None*
shows (*revision-step* $r\ s\ s'$) = (s' = ε) (**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*:
$\exists\, {s_3}'\ s_3.\ {s_3}' \approx s_3 \wedge$ (*revision-step* $r'\ s_2\ {s_3}' \vee s_2 = {s_3}'$) $\wedge$ (*revision-step* $r\ {s_2}'\ s_3 \vee {s_2}' = s_3$) $\Longrightarrow$
$\exists\, s_3\ {s_3}'.\ s_3 \approx {s_3}' \wedge$ (*revision-step* $r\ {s_2}'\ s_3 \vee {s_2}' = s_3$) $\wedge$ (*revision-step* $r'\ s_2\ {s_3}' \vee s_2 = {s_3}'$)
⟨*proof*⟩

lemma *SLC-commute*:
⟦ $s_3 = {s_3}'$; *revision-step* $r'\ s_2\ s_3$; *revision-step* $r\ {s_2}'\ {s_3}'$ ⟧ $\Longrightarrow$
$s_3 \approx {s_3}' \wedge$ (*revision-step* $r'\ s_2\ s_3 \vee s_2 = s_3$) $\wedge$ (*revision-step* $r\ {s_2}'\ {s_3}' \vee {s_2}' = {s_3}'$)
⟨*proof*⟩

7.2.3 Case join-epsilon

lemma *SLC-join*$_\varepsilon$:
assumes
s_1*-r*: $s_1\ r$ = *Some* (σ, τ, $\mathcal{E}$[*Rjoin* (*VE* (*Rid* r''))]) **and**
s_2: $s_2 = \varepsilon$ **and**
side: $s_1\ r''$ = *None* **and**

right: *revision-step* r' s_1 s_2' **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

7.2.4 Case join

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

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

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
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

lemma *local-and-fork-commute*:

assumes
$s_2 = s_1(r \mapsto x)$
$s_2' = s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e))$
$s_2\ r' = Some\ (\sigma,\ \tau,\ \mathcal{E}[Rfork\ e])$
$r'' \notin RID_G\ s_2$
revision-step $r\ s_2'\ (s_1(r' \mapsto (\sigma,\ \tau,\ \mathcal{E}\ [VE\ (Rid\ r'')]),\ r'' \mapsto (\sigma;;\tau,\ \varepsilon,\ e),\ r \mapsto x))$
$r \neq r'$
$r \neq r''$
shows
$\exists\, s_3\ s_3'.\ (s_3 \approx s_3') \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

lemma *SLC-app*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Apply\ (VE\ (Lambda\ x\ e))\ (VE\ v)])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[subst\ (VE\ v)\ x\ e]))$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

lemma *SLC-ifTrue*:
assumes
s_1-*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-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

lemma *SLC-ifFalse*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma,\ \tau,\ \mathcal{E}[Ite\ (VE\ (CV\ F))\ e1\ e2])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma,\ \tau,\ \mathcal{E}[e2]))$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\textit{revision-step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\textit{revision-step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

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

s_2: $s_2 = s_1(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}[VE\ (CV\ Unit)]))$ **and**
side: $l \in dom\ (\sigma;;\tau)$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

lemma *SLC-get*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}\ [Read\ (VE\ (Loc\ l))])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma, \tau, \mathcal{E}[VE\ (the\ ((\sigma;;\tau)\ l))]))$ **and**
side: $l \in dom\ (\sigma;;\tau)$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (revision\text{-}step\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (revision\text{-}step\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*proof*⟩

7.2.6 Case new

lemma *SLC-new*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}[Ref\ (VE\ v)])$ **and**
s_2: $s_2 = s_1(r \mapsto (\sigma, \tau(l \mapsto v), \mathcal{E}\ [VE\ (Loc\ l)]))$ **and**
side: $l \notin LID_G\ s_1$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$ **and**
reach: *reachable* $(s_1 :: ('r,'l,'v)\ global\text{-}state)$ **and**
lid-inf: *infinite* $(UNIV :: 'l\ set)$
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*⟩

7.2.7 Case fork

lemma *SLC-fork*:
assumes
s_1-*r*: $s_1\ r = Some\ (\sigma, \tau, \mathcal{E}\ [Rfork\ e])$ **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-forkee* $\notin RID_G\ s_1$ **and**
right: *revision-step* $r'\ s_1\ s_2'$ **and**
neq: $r \neq r'$ **and**
reach: *reachable* $(s_1 :: ('r,'l,'v)\ global\text{-}state)$ **and**
rid-inf: *infinite* $(UNIV :: 'r\ set)$
shows

$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\mathit{revision\text{-}step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\mathit{revision\text{-}step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*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* $(s_1 :: ('r,'l,'v)\ \mathit{global\text{-}state})$ **and**
lid-inf: *infinite* $(\mathit{UNIV} :: 'l\ \mathit{set})$ **and**
rid-inf: *infinite* $(\mathit{UNIV} :: 'r\ \mathit{set})$
shows
$\exists\, s_3\ s_3'.\ s_3 \approx s_3' \wedge (\mathit{revision\text{-}step}\ r'\ s_2\ s_3 \vee s_2 = s_3) \wedge (\mathit{revision\text{-}step}\ r\ s_2'\ s_3' \vee s_2' = s_3')$
⟨*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* $(s_1 :: ('r,'l,'v)\ \mathit{global\text{-}state})$ **and**
lid-inf: *infinite* $(\mathit{UNIV} :: 'l\ \mathit{set})$ **and**
rid-inf: *infinite* $(\mathit{UNIV} :: 'r\ \mathit{set})$
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* $(s_1 :: ('r,'l,'v)\ \mathit{global\text{-}state})$ **and**
lid-inf: *infinite* $(\mathit{UNIV} :: 'l\ \mathit{set})$ **and**
rid-inf: *infinite* $(\mathit{UNIV} :: 'r\ \mathit{set})$
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*:

$bij\ \alpha \Longrightarrow (\alpha\ r \in RID_V\ (\mathcal{R}_V\ \alpha\ \beta\ v)) = (r \in RID_V\ v)$
$bij\ \beta \Longrightarrow (\beta\ l \in LID_V\ (\mathcal{R}_V\ \alpha\ \beta\ v)) = (l \in LID_V\ v)$
⟨*proof*⟩

lemma *in-renamed-in-unlabelled-expr*:
$bij\ \alpha \Longrightarrow (\alpha\ r \in RID_E\ (\mathcal{R}_E\ \alpha\ \beta\ v)) = (r \in RID_E\ v)$
$bij\ \beta \Longrightarrow (\beta\ l \in LID_E\ (\mathcal{R}_E\ \alpha\ \beta\ v)) = (l \in LID_E\ v)$
⟨*proof*⟩

lemma *in-renamed-in-unlabelled-store*:
assumes
bij-α: $bij\ \alpha$ **and**
bij-β: $bij\ \beta$
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*⟩

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

lemma *in-renamed-in-unlabelled-global*:
assumes
bij-α: $bij\ \alpha$ **and**
bij-β: $bij\ \beta$
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*⟩

lemma *mimicking*:
assumes
step: $revision\text{-}step\ r\ s\ s'$ **and**
bij-α: $bij\ \alpha$ **and**
bij-β: $bij\ \beta$
shows $revision\text{-}step\ (\alpha\ r)\ (\mathcal{R}_G\ \alpha\ \beta\ s)\ (\mathcal{R}_G\ \alpha\ \beta\ s')$
⟨*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-*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*⟩

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*⟩

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*⟩

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*⟩

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.