Abstract

The concurrent refinement algebra developed here is designed to provide a foundation for rely/guarantee reasoning about concurrent programs. The algebra builds on a complete lattice of commands by providing sequential composition, parallel composition and a novel weak conjunction operator. The weak conjunction operator coincides with the lattice supremum providing its arguments are non-aborting, but aborts if either of its arguments do. Weak conjunction provides an abstract version of a guarantee condition as a guarantee process. We distinguish between models that distribute sequential composition over non-deterministic choice from the left (referred to as being conjunctive in the refinement calculus literature) and those that don’t. Least and greatest fixed points of monotone functions are provided to allow recursion and iteration operators to be added to the language. Additional iteration laws are available for conjunctive models. The rely quotient of processes $c$ and $i$ is the process that, if executed in parallel with $i$ implements $c$. It represents an abstract version of a rely condition generalised to a process.
Contents

1 Overview ............................... 4

2 Refinement Lattice ................. 4

3 Sequential Operator ............... 7
   3.1 Basic sequential .................. 7
   3.2 Distributed sequential .......... 8

4 Parallel Operator ................. 10
   4.1 Basic parallel operator .......... 10
   4.2 Distributed parallel ............. 10

5 Weak Conjunction Operator .... 12
   5.1 Distributed weak conjunction ... 13

6 Concurrent Refinement Algebra ... 14

7 Galois Connections and Fusion Theorems 16
   7.1 Lower Galois connections ...... 17
   7.2 Greatest fixpoint fusion theorems 18
   7.3 Upper Galois connections ....... 19
   7.4 Least fixpoint fusion theorems ... 20

8 Iteration ............................. 21
   8.1 Possibly infinite iteration ...... 21
   8.2 Finite iteration ................... 23
   8.3 Infinite iteration ................. 24
   8.4 Combined iteration .............. 25

9 Sequential composition for conjunctive models 26

10 Infimum nat lemmas ............ 28

11 Iteration for conjunctive models 30

12 Rely Quotient Operator ....... 35
   12.1 Basic rely quotient .......... 35
   12.2 Distributed rely quotient ..... 39

13 Conclusions ....................... 42
A Differences to earlier paper
1 Overview

The theories provided here were developed in order to provide support for rely/guarantees concurrency [6, 5]. The theories provide a quite general concurrent refinement algebra that builds on a complete lattice of commands by adding sequential and parallel composition operators as well as recursion. A novel weak conjunction operator is also added as this allows one to build more general specifications. The theories are based on the paper by Hayes [3], however there are some differences that have been introduced to correct and simplify the algebra and make it more widely applicable. See the appendix for a summary of the differences.

The basis of the algebra is a complete lattice of commands (Section 2). Sections 3, 4 and 5 develop laws for sequential composition, parallel composition and weak conjunction, respectively, based on the refinement lattice. Section 6 brings the above theories together. Section 7 adds least and greatest fixed points and there associated laws, which allows finite, possibly infinite and strictly infinite iteration operators to be defined in Section 8 in terms of fixed points.

The above theories do not assume that sequential composition is conjunctive. Section 9 adds this assumption and derives a further set of laws for sequential composition and iterations.

Section 12 builds on the general theory to provide a rely quotient operator that can be used to provide a general rely/guarantee framework for reasoning about concurrent programs.

2 Refinement Lattice

theory Refinement-Lattice
imports Main
begin

unbundle lattice-syntax

The underlying lattice of commands is complete and distributive. We follow the refinement calculus tradition so that ⊔ is non-deterministic choice and c ⊑ d means c is refined (or implemented) by d.

declare [[show-sorts]]

Remove existing notation for quotient as it interferes with the rely quotient

no-notation Equiv-Relations.quotient (infixl '///' 90)
class refinement-lattice = complete-distrib-lattice
begin

The refinement lattice infimum corresponds to non-deterministic choice for commands.

abbreviation refine :: 'a ⇒ 'a ⇒ bool (infix ⊑ 50)
where
c ⊑ d ≡ less-eq c d

abbreviation refine-strict :: 'a ⇒ 'a ⇒ bool (infix ⊏ 50)
where
c ⊏ d ≡ less c d

Non-deterministic choice is monotonic in both arguments

lemma inf-mono-left: a ⊑ b =⇒ a ⊓ c ⊑ b ⊓ c
using inf-mono by auto

lemma inf-mono-right: c ⊑ d =⇒ a ⊓ c ⊑ a ⊓ d
using inf-mono by auto

Binary choice is a special case of choice over a set.

lemma Inf2-inf: ∏ {f x | x. x ∈ {c, d}} = f c ⊓ f d
proof −
  have {f x | x. x ∈ {c, d}} = {f c, f d} by blast
  then have ∏ {f x | x. x ∈ {c, d}} = ∏ {f c, f d} by simp
  also have ... = f c ⊓ f d by simp
  finally show ?thesis .
qed

Helper lemma for choice over indexed set.

lemma INF-inf: (∏ x∈X. f x) = (∏ {f x | x. x ∈ X})
  by (simp add: Setcompr-eq-image)

lemma (in −) INF-absorb-args: (∏ i j. (f::nat ⇒ 'c::complete-lattice) (i + j)) = (∏ k. f
k)
proof (rule order-class.order.antisym)
  show (∏ k.f k) ≤ (∏ i j.f (i + j))
    by (simp add: complete-lattice-class.INF-lower complete-lattice-class.le-INF-iff)
next
  have ∏k. ∃i j.f (i + j) ≤ f k
    by (metis Nat.add-0-right order-refl)
then have $\forall k. \exists i. (\bigcap j. f (i + j)) \leq f k$ 
by (meson UNIV-I complete-lattice-class.INF-lower2)

then show $(\bigcap i. j. f (i + j)) \leq (\bigcap k. f k)$
by (simp add: complete-lattice-class.INF-mono)

qed

lemma (in –) nested-Collect: \{f y | y \in \{g x | x \in X\}\} = \{f (g x) | x \in X\}
by blast

A transition lemma for INF distributivity properties, going from Inf to INF, qualified version followed by a straightforward one.

lemma Inf-distrib-INF-qual:
fixes f :: 'a \Rightarrow 'a
assumes qual: P \{d x | x \in X\}
assumes f-Inf-distrib: $\forall c D. P D \Longrightarrow f c (\bigcap D) = \bigcap \{f c d | d \in D\}$
shows f c (\bigcap x\in X. d x) = (\bigcap x\in X. c f (d x))

proof –
  have f c (\bigcap x\in X. d x) = f c (\bigcap \{d x | x \in X\}) by (simp add: INF-Inf)
also have ... = (\bigcap \{f c dx | dx \in \{d x | x \in X\}\}) by (simp add: qual f-Inf-distrib)
also have ... = (\bigcap \{f c (d x) | x \in X\}) by (simp only: nested-Collect)
also have ... = (\bigcap x\in X. f c (d x)) by (simp add: INF-Inf)
finally show ?thesis .

qed

lemma Inf-distrib-INF:
fixes f :: 'a \Rightarrow 'a
assumes f-Inf-distrib: $\forall c D. f c (\bigcap D) = \bigcap \{f c d | d \in D\}$
shows f c (\bigcap x\in X. d x) = (\bigcap x\in X. f c (d x))
by (simp add: Setcompr-eq-image f-Inf-distrib image-comp)

end

lemmas refine-trans = order.trans

More transitivity rules to make calculational reasoning smoother

declare ord-eq-le-trans[trans]
declare ord-le-eq-trans[trans]
declare dual-order.trans[trans]

abbreviation
dist-over-sup :: ('a::refinement-lattice \Rightarrow 'a) \Rightarrow bool
where
\[ \text{dist-over-sup} \; F \equiv (\forall \; X . \; F (\bigsqcup X) = (\bigsqcup x \in X . \; F (x))) \]

abbreviation
\[ \text{dist-over-inf} :: (\forall a :: \text{refinement-lattice} \Rightarrow a) \Rightarrow \text{bool} \]
where
\[ \text{dist-over-inf} \; F \equiv (\forall \; X . \; F (\bigcap X) = (\bigcap x \in X . \; F (x))) \]

3 Sequential Operator

theory Sequential
imports Refinement-Lattice
begin

3.1 Basic sequential

The sequential composition operator "\; ; \;" is associative and has identity nil but it is not commutative. It has \( \bot \) as a left annihilator.

locale seq =
fixes seq :: (\forall a :: \text{refinement-lattice} \Rightarrow a \Rightarrow a) \Rightarrow a
assumes seq-bot [simp]: \( \bot ; c = \bot \)

locale nil =
fixes nil :: (\forall a :: \text{refinement-lattice} \Rightarrow a)

The monoid axioms imply "\; ; \;" is associative and has identity nil. Abort is a left annihilator of sequential composition.

locale sequential = seq + nil + seq: monoid seq nil
begin

declare seq.assoc [algebra-simps, field-simps]

lemmas seq-assoc = seq.assoc
lemmas seq-nil-right = seq.right-neutral
lemmas seq-nil-left = seq.left-neutral

end
3.2 Distributed sequential

Sequential composition distributes across arbitrary infima from the right but only across the binary (finite) infima from the left and hence it is monotonic in both arguments. We consider left distribution first. Note that Section 9 considers the case in which the weak-seq-inf-distrib axiom is strengthened to an equality.

locale seq-distrib-left = sequential +
  assumes weak-seq-inf-distrib:
  \((c::'a::refinement-lattice);(d_0 \cap d_1) \subseteq (c;d_0 \cap c;d_1)\)
begin

Left distribution implies sequential composition is monotonic is its right argument

lemma seq-mono-right: \(c_0 \subseteq c_1 \implies d : c_0 \subseteq d : c_1\)
  by (metis inf.absorb_iff2 le-inff weak-seq-inf-distrib)

lemma seq-bot-right [simp]: \(c;\bot \subseteq c\)
  by (metis bot.extremum seq.right-neutral seq-mono-right)

end

locale seq-distrib-right = sequential +
  assumes Inf-seq-distrib:
  \((\prod C) : d = (\prod (c::'a::refinement-lattice)\in C. c ; d)\)
begin

lemma INF-seq-distrib: \((\prod c\in C. f c) ; d = (\prod c\in C. f c ; d)\)
  using Inf-seq-distrib by (auto simp add: image-comp)

lemma inf-seq-distrib: \((c_0 \cap c_1) ; d = (c_0 ; d \cap c_1 ; d)\)
  proof
    have \((c_0 \cap c_1) ; d = (\prod \{c_0, c_1\}) ; d\) by simp
    also have ... = \((\prod c\in\{c_0, c_1\}, c ; d)\) by (fact Inf-seq-distrib)
    also have ... = \((c_0 ; d) \cap (c_1 ; d)\) by simp
    finally show ?thesis .
  qed

lemma seq-mono-left: \(c_0 \subseteq c_1 \implies c_0 ; d \subseteq c_1 ; d\)
  by (metis inf.absorb_iff2 inf-seq-distrib)

lemma seq-top [simp]: \(\top ; c = \top\)
  proof
    have \(\top ; c = (\prod a\in\{\} . a ; c)\)
by (metis Inf-empty Inf-seq-distrib)
thus ?thesis
by simp
qed

primrec seq-power :: 'a ⇒ nat ⇒ 'a (infixr "^" 80) where
  seq-power-0: a ^ 0 = nil
| seq-power-Suc: a ^ Suc n = a ; (a ^ n)

notation (latex output)
  seq-power (\cdot) [1000] 1000

notation (HTML output)
  seq-power (\cdot) [1000] 1000

lemma seq-power-front: (a ^ n) ; a = a ; (a ^ n)
  by (induct n, simp-all add: seq-assoc)

lemma seq-power-split-less: i < j ⇒ (b ^ j) = (b ^ i) ; (b ^ (j - i))
  proof (induct j arbitrary: i type: nat)
  case 0
  thus ?case by simp
  next
  case (Suc j)
  have b ^ Suc j = b ; (b ^ i) ; (b ^ (j - i))
    using Suc.hyps Suc.prems less-Suc-eq seq-assoc by auto
  also have ... = (b ^ i) ; b ; (b ^ (j - i)) by (simp add: seq-power-front)
  also have ... = (b ^ i) ; (b ^ Suc j) by force
  finally show ?case .
  qed

end

locale seq-distrib = seq-distrib-right + seq-distrib-left
begin

lemma seq-mono: c_1 ⊑ d_1 ⇒ c_2 ⊑ d_2 ⇒ c_1;c_2 ⊑ d_1;d_2
  using seq-mono-left seq-mono-right by (metis inf.orderE le-infI2)

end

9
4 Parallel Operator

theory Parallel
imports Refinement-Lattice
begin

4.1 Basic parallel operator

The parallel operator is associative, commutative and has unit skip and has as an annihilator the lattice bottom.

locale skip =
  fixes skip :: 'a::refinement-lattice (skip)

locale par =
  fixes par :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl 75)
  assumes abort-par: ⊥ ∥ c = ⊥

locale parallel = par + skip + par: comm-monoid par skip
begin

lemmas [algebra-simps, field-simps] =
  par.assoc
  par.commute
  par.left-commute

lemmas par-assoc = par.assoc
lemmas par-commute = par.commute
lemmas par-skip = par.right-neutral
lemmas par-skip-left = par.left-neutral

end

4.2 Distributed parallel

The parallel operator distributes across arbitrary non-empty infima.

locale par-distrib = parallel +
  assumes par-Inf-distrib: D ≠ {} ⇒ c ∥ (∩ D) = (∩ d∈D. c ∥ d)

begin
lemma Inf-par-distrib: $D \neq \emptyset \Longrightarrow (\bigsqcap D) \parallel c = (\bigsqcap d \in D. d \parallel c)$
using par-Inf-distrib par-commute by simp

lemma par-INF-distrib: $X \neq \emptyset \Longrightarrow c \parallel (\bigsqcap x \in X. d x) = (\bigsqcap x \in X. c \parallel d x)$
using par-INF-distrib by (auto simp add: image-comp)

lemma INF-par-distrib: $X \neq \emptyset \Longrightarrow (\bigsqcap x \in X. d x) \parallel c = (\bigsqcap x \in X. d x \parallel c)$
using par-INF-distrib par-commute by (metis mono-tags, lifting INF-cong)

lemma INF-INF-par-distrib: $X \neq \emptyset \Longrightarrow Y \neq \emptyset \Longrightarrow (\bigsqcap x \in X. c x) \parallel (\bigsqcap y \in Y. d y) = (\bigsqcap x \in X. \bigsqcap y \in Y. c x \parallel d y)$
proof −
assume nonempty-X: $X \neq \emptyset$
assume nonempty-Y: $Y \neq \emptyset$
have $(\bigsqcap x \in X. c x) \parallel (\bigsqcap y \in Y. d y) = (\bigsqcap x \in X. \bigsqcap y \in Y. c x \parallel d y)$
using INF-par-distrib by (meson nonempty-Y)
also have . . . = $(\bigsqcap x \in X. \bigsqcap y \in Y. c x \parallel d y)$ using par-INF-distrib by (metis nonempty-Y)
thus ?thesis by (simp add: calculation)
qed

lemma inf-par-distrib: $(c_0 \sqcap c_1) \parallel d = (c_0 \parallel d) \sqcap (c_1 \parallel d)$
proof −
have $(c_0 \sqcap c_1) \parallel d = (\bigsqcap \{c_0, c_1\}) \parallel d$ by simp
also have . . . = $(\bigsqcap c \in \{c_0, c_1\}. c \parallel d)$ using Inf-par-distrib by (meson insert-not-empty)
also have . . . = $c_0 \parallel d \sqcap c_1 \parallel d$ by simp
finally show ?thesis .
qed

lemma inf-par-distrib2: $d \parallel (c_0 \sqcap c_1) = (d \parallel c_0) \sqcap (d \parallel c_1)$
using inf-par-distrib par-commute by auto

lemma inf-par-product: $(a \sqcap b) \parallel (c \sqcap d) = (a \parallel c) \sqcap (b \parallel d)$
by (simp add: inf-commute inf-par-distrib inf-sup-aci(3))

lemma par-mono: $c_1 \subseteq d_1 \Longrightarrow c_2 \subseteq d_2 \Longrightarrow c_1 \parallel d_2 \subseteq c_2 \parallel d_1 \parallel d_2$
by (metis inf.orderE le-inf-iff order-refl inf-par-distrib par-commute)

end
end
5 Weak Conjunction Operator

theory Conjunction
imports Refinement-Lattice
begin

The weak conjunction operator \(\sqcap\) is similar to least upper bound (\(\sqcup\)) but is abort strict, i.e. the lattice bottom is an annihilator: \(c \sqcap \bot = \bot\). It has identity the command chaos that allows any non-aborting behaviour.

locale chaos =
  fixes chaos :: 'a::refinement-lattice (chaos)

locale conj =
  fixes conj :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl \(\sqcap\) 80)
  assumes conj-bot-right: \(c \sqcap \bot = \bot\)

Conjunction forms an idempotent, commutative monoid (i.e. a semi-lattice), with identity chaos.

locale conjunction = conj + chaos + conj: semilattice-neutr conj chaos
begin

lemmas [algebra-simps, field-simps] =
  conj.assoc
  conj.commute
  conj.left-commute

lemmas conj-assoc = conj.assoc
lemmas conj-commute = conj.commute
lemmas conj-idem = conj.idem
lemmas conj-chaos = conj.right-neutral
lemmas conj-chaos-left = conj.left-neutral

lemma conj-bot-left [simp]: \(\bot \sqcap c = \bot\)
using conj-bot-right local.conj-commute by fastforce

lemma conj-not-bot: \(a \sqcap b \neq \bot\) \(\Rightarrow\) \(a \neq \bot \land b \neq \bot\)
using conj-bot-right by auto

lemma conj-distrib1: \(c \sqcap (d_0 \sqcap d_1) = (c \sqcap d_0) \sqcap (c \sqcap d_1)\)
by (metis conj-assoc conj-commute conj-idem)

end
5.1 Distributed weak conjunction

The weak conjunction operator distributes across arbitrary non-empty infima.

```
locale conj-distrib = conjunction +
  assumes Inf-conj-distrib: D \neq \{\} \implies (\bigsqcap D) \sqcap c = (\bigsqcap d \in D. d \sqcap c)
```

begin

lemma conj-Inf-distrib: D \neq \{\} \implies c \sqcap (\bigsqcap D) = (\bigsqcap d \in D. c \sqcap d)
using Inf-conj-distrib conj-commute by auto

lemma inf-conj-distrib: (c \sqcap c_1) \sqcap d = (c_0 \sqcap d) \sqcap (c_1 \sqcap d)
proof
  have (c_0 \sqcap c_1) \sqcap d = (\bigsqcap \{c_0, c_1\}) \sqcap d by simp
  also have ... = (\bigsqcap c \in \{c_0, c_1\}. c \sqcap d) by (rule Inf-conj-distrib, simp)
  also have ... = (c_0 \sqcap d) \sqcap (c_1 \sqcap d) by simp
  finally show \?thesis .
qed

lemma inf-conj-product: (a \sqcap b) \sqcap (c \sqcap d) = (a \sqcap d) \sqcap (b \sqcap d)
by (metis inf-conj-distrib conj-commute inf-assoc)

lemma conj-mono: c_0 \sqsubseteq d_0 \implies c_1 \sqsubseteq d_1 
  \implies c_0 \sqcap c_1 \sqsubseteq d_0 \sqcap d_1
by (metis inf-conj-distrib conj-commute inf-assoc)

lemma conj-mono-left: c_0 \sqsubseteq c_1 \implies c_0 \sqcap d \sqsubseteq c_1 \sqcap d
by (simp add: conj-mono)

lemma conj-mono-right: c_0 \sqsubseteq c_1 \implies d \sqcap c_0 \sqsubseteq d \sqcap c_1
by (simp add: conj-mono)

lemma conj-refine: c_0 \sqsubseteq d \implies c_1 \sqsubseteq d 
  \implies c_0 \sqcap c_1 \sqsubseteq d
by (metis conj-idem conj-mono)

lemma refine-to-conj: c \sqsubseteq d_0 \implies c \sqsubseteq d_1 
  \implies c \sqsubseteq d_0 \sqcap d_1
by (metis conj-idem conj-mono)

lemma conjoin-non-aborting: chaos \sqsubseteq c \implies d \sqsubseteq d \sqcap c
by (metis conj-mono order.refl conj-chaos)

lemma conjunction-sup: c \sqcap d \sqsubseteq c \sqcup d
by (simp add: conj-refine)
```
lemma conjunction-sup-nonaborting:
  assumes chaos ⊑ c and chaos ⊑ d
  shows c ∩ d = c ⊔ d
proof (rule antisym)
  show c ⊔ d ⊑ c ∩ d using assms(1) assms(2) conjoin-non-aborting local.conf-commute
  by fastforce
next
  show c ∩ d ⊑ c ⊔ d by (metis conjunction-sup)
qed

lemma conjoin-top: chaos ⊑ c ⇒ c ∩ ⊤ = ⊤
by (simp add: conjunction-sup-nonaborting)

end

end

6 Concurrent Refinement Algebra

This theory brings together the three main operators: sequential composition, parallel composition and conjunction, as well as the iteration operators.

theory CRA
imports
  Sequential
  Conjunction
  Parallel
begin
Locale sequential-parallel brings together the sequential and parallel operators and relates their identities.
locale sequential-parallel = seq-distrib + par-distrib +
  assumes nil-par-nil: nil || nil ⊑ nil
  and skip-nil: skip ⊑ nil
  and skip-skip: skip ⊑ skip:skip
begin
lemma nil-absorb: nil || nil = nil using nil-par-nil skip-nil par-skip
by (metis inf.absorb-iff2 inf.orderE inf-par-distrib2)

lemma skip-absorb [simp]: skip:skip = skip
by (metis antisym seq-mono-right seq-nil-right skip-skip skip-nil)

end

end
Locale conjunction-parallel brings together the weak conjunction and parallel operators and relates their identities. It also introduces the interchange axiom for conjunction and parallel.

locale conjunction-parallel = conj-distrib + par-distrib +
  assumes chaos-par-top: \( \top \sqsubseteq \text{chaos} \parallel \top \)
  assumes chaos-par-chaos: \( \text{chaos} \sqsubseteq \text{chaos} \parallel \text{chaos} \)
  assumes parallel-interchange: \((c_0 \parallel c_1) \sqcap (d_0 \parallel d_1) \sqsubseteq (c_0 \sqcap d_0) \parallel (c_1 \sqcap d_1)\)
begin
lemma chaos-skip: \( \text{chaos} \sqsubseteq \text{skip} \)
proof -
  have \( \text{chaos} = (\text{chaos} \parallel \text{skip}) \sqcap (\text{skip} \parallel \text{chaos}) \) by simp
  then have \( \cdots \sqsubseteq (\text{chaos} \sqcap \text{skip}) \parallel (\text{skip} \sqcap \text{chaos}) \) using parallel-interchange by blast
  thus \?thesis by auto
qed

lemma chaos-par-chaos-eq: \( \text{chaos} = \text{chaos} \parallel \text{chaos} \)
by (metis antisym chaos-par-chaos chaos-skip order-refl par-mono par-skip)

lemma nonabort-par-top: \( \text{chaos} \sqsubseteq c \Longrightarrow c \parallel \top = \top \)
by (metis chaos-par-top par-mono top extremum-uniqueI)

lemma skip-conj-top: \( \text{skip} \sqcap \top = \top \)
by (simp add: chaos-skip conjoin-top)

lemma conj-distrib2: \( c \sqsubseteq c \parallel (d_0 \parallel d_1) \sqsubseteq (c \sqcap d_0) \parallel (c \sqcap d_1) \)
proof -
  assume \( c \sqsubseteq c \parallel c \)
  then have \( c \sqcap (d_0 \parallel d_1) \sqsubseteq (c \parallel c) \sqcap (d_0 \parallel d_1) \) by (metis conj-mono order.refl)
  thus \?thesis by (metis parallel-interchange refine-trans)
qed
end

Locale conjunction-sequential brings together the weak conjunction and sequential operators. It also introduces the interchange axiom for conjunction and sequential.

locale conjunction-sequential = conj-distrib + seq-distrib +
  assumes chaos-seq-chaos: \( \text{chaos} \sqsubseteq \text{chaos}:\text{chaos} \)
  assumes sequential-interchange: \((c_0;c_1) \sqcap (d_0;d_1) \sqsubseteq (c_0 \sqcap d_0);(c_1 \sqcap d_1) \)
begin
lemma chaos-nil: \(\text{chaos} \sqsubseteq \text{nil}\)
by (metis conj-chaos local.conj-commute seq-nil-left seq-nil-right
sequential-interchange)

lemma chaos-seq-absorb: \(\text{chaos} = \text{chaos};\text{chaos}\)
proof (rule antisym)
  show \(\text{chaos} \subseteq \text{chaos};\text{chaos}\) by (simp add: chaos-seq-chaos)
next
  show \(\text{chaos};\text{chaos} \subseteq \text{chaos}\) using chaos-nil
    using seq-mono-left seq-nil-left by fastforce
qed

lemma seq-bot-conj: \(c;\bot \sqcap d \sqsubseteq (c \sqcap d);\bot\)
  by (metis (no-types) conj-bot-left seq-nil-right sequential-interchange)

lemma conj-seq-bot-right [simp]: \(c;\bot \sqcap c = c;\bot\)
proof (rule antisym)
  show lr: \(c;\bot \sqcap c \sqsubseteq c;\bot\) by (metis seq-bot-conj conj-idem)
next
  show rl: \(c;\bot \sqsubseteq c;\bot \sqcap c\)
by (metis conj-idem conj-mono-right seq-bot-right)
qed

lemma conj-distrib3: \(c \sqsubseteq c;c \Rightarrow c \sqcap (d_0 ; d_1) \subseteq (c \sqcap d_0);(c \sqcap d_1)\)
proof –
  assume \(c \sqsubseteq c;c\)
  then have \(c \sqcap (d_0;d_1) \subseteq (c;c) \sqcap (d_0;d_1)\) by (metis conj-mono order.refl)
  thus \(?\text{thesis}\) by (metis sequential-interchange refine-trans)
qed

end

Locale cra brings together sequential, parallel and weak conjunction.
locale cra = sequential-parallel + conjunction-parallel + conjunction-sequential
end

7 Galois Connections and Fusion Theorems

theory Galois-Connections
The concept of Galois connections is introduced here to prove the fixed-point fusion lemmas. The definition of Galois connections used is quite simple but encodes a lot of information. The material in this section is largely based on the work of the Eindhoven Mathematics of Program Construction Group [1] and the reader is referred to their work for a full explanation of this section.

7.1 Lower Galois connections

lemma Collect-2set [simp]: \{F x | x = a \lor x = b\} = \{F a, F b\}
by auto

locale lower-galois-connections
begin

definition l-adjoint :: ('a::refinement-lattice \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) (\♭ [201] 200)
where (F\♭) x \equiv \{y. x \sqsubseteq F y\}

lemma dist-inf-mono:
  assumes distF: dist-over-inf F
  shows mono F
proof
  fix x :: 'a and y :: 'a
  assume x \sqsubseteq y
  then have F x = F (x \cap y) by (simp add: le-iff-inf)
  also have ... = F x \cap F y
  proof --
  from distF
  have F (\{x, y\}) = \{F x, F y\} by (drule-tac x = \{x, y\} in spec, simp)
  then show F (x \cap y) = F x \cap F y by simp
  qed
  finally show F x \sqsubseteq F y by (metis le-iff-inf)
  qed

lemma l-cancellation: dist-over-inf F \Rightarrow x \sqsubseteq (F \circ F\♭) x
proof --
  assume dist: dist-over-inf F

  define Y where Y = \{F y | y. x \sqsubseteq F y\}
define $X$ where $X = \{x\}$

have $(\forall y \in Y. (\exists x \in X. x \sqsubseteq y))$ using $X\text{-def} Y\text{-def}$ CollectD singletonI by auto
then have $\bigsqcap X \sqsubseteq \bigsqcap Y$ by (simp add: Inf-mono)
then have $x \sqsubseteq \bigsqcap \{F y \mid y \sqsubseteq F y\}$ by (simp add: $X\text{-def} Y\text{-def}$)
then have $x \sqsubseteq F (\bigsqcap \{y \sqsubseteq F y\})$ by (simp add: dist le-INF-iff)
thus $?\text{thesis}$ by (metis comp-def l-adjoint-def)

qed

lemma $l$-galois-connection: $\text{dist-over-inf} F \implies ((F^\circ) x \sqsubseteq y) \iff (x \sqsubseteq F y)$
proof
assume $x \sqsubseteq F y$
then have $\bigsqcap \{y \sqsubseteq F y\} \subseteq y$ by (simp add: Inf-lower)
thus $(F^\circ) x \sqsubseteq y$ by (metis l-adjoint-def)
next
assume $\text{dist}: \text{dist-over-inf} F$ then have $\text{monoF}$: $\text{mono} F$ by (simp add: dist-inf-mono)
assume $(F^\circ) x \sqsubseteq y$ then have $a$: $F ((F^\circ) x) \sqsubseteq F y$ by (simp add: $\text{monoD} \text{ monoF}$)
have $x \sqsubseteq F ((F^\circ) x)$ using $\text{dist l-cancellation}$ by simp
thus $x \sqsubseteq F y$ using $a$ by auto
qed

lemma $v$-simple-fusion: $\text{mono} G \implies \forall x. ((F \circ G) x \sqsubseteq (H \circ F) x) \implies F (\text{gfp} G) \sqsubseteq \text{gfp} H$
by (metis comp-eq-dest-lhs gfp-unfold gfp-upperbound)

7.2 Greatest fixpoint fusion theorems

Combining lower Galois connections and greatest fixed points allows elegant proofs of the weak fusion lemmas.

theorem fusion-gfp-geq:
assumes $\text{monoH}$: $\text{mono} H$
and $\text{distribF}$: $\text{dist-over-inf} F$
and $\text{comp-geq}$: $\forall x. ((H \circ F) x \sqsubseteq (F \circ G) x)$
shows $\text{gfp} H \sqsubseteq F (\text{gfp} G)$
proof —
have $(\text{gfp} H) \sqsubseteq (F \circ F^\circ) (\text{gfp} H)$ using $\text{distribF} \text{ l-cancellation}$ by simp
then have $H (\text{gfp} H) \sqsubseteq H ((F \circ F^\circ) (\text{gfp} H))$ by (simp add: $\text{monoD} \text{ monoH}$)
then have $H (\text{gfp} H) \sqsubseteq F ((G \circ F^\circ) (\text{gfp} H))$ using $\text{comp-geq}$ by (metis comp-def refine-trans)
then have $(F^\circ) (H (\text{gfp} H)) \sqsubseteq (G \circ F^\circ) (\text{gfp} H)$ using $\text{distribF}$ by (metis $\text{mono-tags}$ $l$-galois-connection)
then have $(F^\circ) (\text{gfp} H) \sqsubseteq (\text{gfp} G)$ by (metis comp-apply gfp-unfold gfp-upperbound monoH)
thus \( \text{gfp } H \subseteq F (\text{gfp } G) \) using \textit{distribF} by (metis (mono-tags) l-galois-connection)

qed

\textbf{theorem} fusion-gfp-eq:
\textbf{assumes} \textit{monoH}: mono \( H \) \textbf{and} \textit{monoG}: mono \( G \)
\textbf{and} \textit{distF}: dist-over-inf \( F \)
\textbf{and} \textit{fgh-comp}: \( \forall x. ((F \circ G) x = (H \circ F) x) \)
\textbf{shows} \( F (\text{gfp } G) = \text{gfp } H \)

\textbf{proof} (rule antisym)
\textbf{show} \( F (\text{gfp } G) \subseteq (\text{gfp } H) \) by (metis fgh-comp le-less v-simple-fusion monoG)
\textbf{next}
\textbf{have} \( \forall x. ((H \circ F) x \subseteq (F \circ G) x) \) using fgh-comp by auto
\textbf{then show} \( \text{gfp } H \subseteq F (\text{gfp } G) \) using \( \text{monoH} \ \text{distF} \ \text{fusion-gfp-geq} \) by blast

qed

end

7.3 Upper Galois connections

\textbf{locale} upper-galois-connections
\begin{align*}
\textbf{begin} \\
\textbf{definition} &\quad u\text{-adjoint} :: \langle 'a::refinement-lattice \Rightarrow 'a \Rightarrow 'a \rangle \\
&\quad \textit{where} \\
&\quad (F^\#) \, x \equiv \bigsqcup \{ y . F \, y \subseteq x \} \\
\textbf{lemma} &\quad \textit{dist-sup-mono}: \\
&\quad \textbf{assumes} \textit{distF}: \textit{dist-over-sup} \, F \\
&\quad \textbf{shows} \, \textit{mono} \, F \\
\textbf{proof} \\
&\quad \textbf{fix} \, x :: \langle 'a \rangle \textbf{ and} \, y :: \langle 'a \rangle \\
&\quad \textbf{assume} \, x \subseteq y \\
&\quad \textbf{then have} \, F \, y = F \, (x \sqcup y) \textbf{ by} \, (\text{simp add: le-iff-sup}) \\
&\quad \textbf{also have} \, \ldots = F \, x \sqcup F \, y \\
&\quad \textbf{proof} - \\
&\quad \quad \textbf{from} \textit{distF} \\
&\quad \quad \textbf{have} \, F \, (\bigsqcup \{ x, y \}) = \bigsqcup \{ F \, x, F \, y \} \textbf{ by} \, (\text{drule-tac} \, x = \{ x, y \} \, \text{in spec, simp}) \\
&\quad \quad \textbf{then show} \, F \, (x \sqcup y) = F \, x \sqcup F \, y \textbf{ by} \, \text{simp} \\
&\quad \quad \textbf{qed} \\
&\quad \textbf{finally show} \, F \, x \subseteq F \, y \textbf{ by} \, (\text{metis le-iff-sup}) \\
\textbf{qed}
\end{align*}

end
lemma u-cancellation: \( \text{dist-over-sup } F \implies (F \circ F^\#) x \sqsubseteq x \)

proof –
assume \( \text{dist: dist-over-sup } F \)
define \( Y \) where \( Y = \{ F y \mid y. F y \sqsubseteq x \} \)
define \( X \) where \( X = \{ x \} \)

have \( (\forall y \in Y. (\exists x \in X. y \sqsubseteq x)) \) using X-def Y-def CollectD singletonI by auto
then have \( \bigsqcup Y \sqsubseteq \bigsqcup X \) by (simp add: Sup-mono)
then have \( \bigsqcup \{ F y \mid y. F y \sqsubseteq x \} \sqsubseteq x \) by (simp add: X-def Y-def)
then have \( F (\bigsqcup \{ y. F y \sqsubseteq x \}) \sqsubseteq x \) using SUP-le-iff dist by fastforce
thus \( \text{thesis by (metis comp-def u-adjoint-def)} \)

qed

lemma u-galois-connection: \( \text{dist-over-sup } F \implies (F x \sqsubseteq y) \iff (x \sqsubseteq (F^\#) y) \)

proof
assume \( \text{dist: dist-over-sup } F \) then have \( \text{monoF: mono F} \) by (simp add: dist-sup-mono)
assume \( x \sqsubseteq (F^\#) y \) then have \( a: F x \sqsubseteq F (\bigsqcup (F^\#) y) \) by (simp add: monoD monoF)
have \( F (\bigsqcup (F^\#) y) \sqsubseteq y \) using dist u-cancellation by simp
thus \( x \sqsubseteq y \) using \( a \) by auto
next
assume \( F x \sqsubseteq y \)
then have \( x \sqsubseteq \bigsqcup \{ x. F x \sqsubseteq y \} \) by (simp add: Sup-upper)
thus \( x \sqsubseteq (F^\#) y \) by (metis u-adjoint-def)

qed

lemma u-simple-fusion: \( \text{mono } H \implies \forall x. ((F \circ G) x \sqsubseteq (G \circ H) x) \implies \text{lfp } F \sqsubseteq G \text{ (lfp } H) \)
by (metis comp-def lfp-lowerbound lfp-unfold)

7.4 Least fixpoint fusion theorems

Combining upper Galois connections and least fixed points allows elegant proofs of the strong fusion lemmas.

theorem fusion-lfp-leq:
assumes \( \text{monoH: mono } H \)
and \( \text{distribF: dist-over-sup } F \)
and \( \text{comp-leq: } (F \circ G) x \sqsubseteq (H \circ F) x \)
shows \( \text{lfp } (lfp G) \sqsubseteq (lfp H) \)

proof –
have \( ((F \circ F^\#) (lfp H)) \sqsubseteq lfp H \) using distribF u-cancellation by simp
then have \( H ((F \circ F^\#) (lfp H)) \sqsubseteq H \) (lfp H) by (simp add: monoD monoH)
then have \( F ((G \circ F^\#) (lfp H)) \sqsubseteq H \) (lfp H) using comp-leq by (metis comp-def refine-trans)
then have \((G \circ F^#) \ (lfp \ H) \subseteq (F^#) \ (H \ (lfp \ H))\) using distribF by (metis (mono-tags) u-galois-connection)

then have \((lfp \ G) \subseteq (F^#) \ (lfp \ H)\) by (metis comp-def def-lfp-unfold lfp-lowerbound monoH)

thus \(F \ (lfp \ G) \subseteq (lfp \ H)\) using distribF by (metis (mono-tags) u-galois-connection)

qed

theorem fusion-lfp-eq:

assumes monoH: mono H and monoG: mono G and distF: dist-over-sup F and fgh-comp: \(\forall x. \ ((F \circ G) \ x = (H \circ F) \ x)\)

shows \(F \ (lfp \ G) = (lfp \ H)\)

proof (rule antisym)

show \(lfp \ H \subseteq F \ (lfp \ G)\) by (metis monoG fgh-comp eq-iff upper-galois-connections.u-simple-fusion)

next

have \(\forall x. \ (F \circ G) \ x \subseteq (H \circ F) \ x\) using fgh-comp by auto

then show \(F \ (lfp \ G) \subseteq (lfp \ H)\) using monoH distF fusion-lfp-leq by blast

qed

end

end

8 Iteration

theory Iteration

imports

Galois-Connections

CRA

begin

8.1 Possibly infinite iteration

Iteration of finite or infinite steps can be defined using a least fixed point.

locale finite-or-infinite-iteration = seq-distrib + upper-galois-connections

begin

definition iter :: 

\(\forall a. \ (\omega :: [103] \ 102)\)

where

\(c^\omega \equiv lfp \ (\lambda x. \ nil \ \in \ c:x)\)


lemma iter-step-mono: mono (\(\lambda x. \text{nil} \sqcap c; x\))
  by (meson inf-mono order-refl seq-mono-right mono-def)

This fixed point definition leads to the two core iteration lemmas: folding and induction.

**Theorem iter-unfold:** \(c^\omega = \text{nil} \sqcap c; c^\omega\)
  using iter-def iter-step-mono lfp-unfold by auto

**Lemma iter-induct-nil:** \(\text{nil} \sqcap c; x \sqsubseteq x \Rightarrow c^\omega \sqsubseteq x\)
  by (simp add: iter-def lfp-lowerbound)

**Lemma iter0:** \(c^\omega \sqsubseteq c\)
  by (metis inf-le2 iter0 iter-unfold order.trans seq-mono-right seq-nil-right)

**Lemma iter1:** \(c^\omega \sqsubseteq c\)
  by (metis inf-le2 iter0 iter-unfold order.trans seq-mono-right seq-nil-right)

**Lemma iter2** [simp]: \(c^\omega; c^\omega = c^\omega\)

**Proof (rule antisym)**
  - show \(c^\omega; c^\omega \sqsubseteq c^\omega\) using iter0 seq-mono-right by fastforce

**Next**
  - have \(a : \text{nil} \sqcap c; c^\omega; c^\omega \sqsubseteq \text{nil} \sqcap c; c^\omega \sqcap c; c^\omega; c^\omega\)
    by (metis inf-greatest inf-le2 inf-mono iter0 order-refl seq-distrib-leftseq-mono-right
    seq-distrib-left-axioms seq-nil-right)
  - then have \(b : \ldots = c^\omega \sqcap c; c^\omega; c^\omega\) using iter-unfold by auto
  - then have \(c : \ldots = (\text{nil} \sqcap c; c^\omega); c^\omega\) by (simp add: inf-seq-distrib)
  - thus \(c^\omega \sqsubseteq c^\omega; c^\omega\) using a iter-induct-nil iter-unfold seq-assoc by auto

**QED**

**Lemma iter-mono:** \(c \sqsubseteq d \Rightarrow c^\omega \sqsubseteq d^\omega\)

**Proof**
  - assume \(c \sqsubseteq d\)
  - then have \(\text{nil} \sqcap c; d^\omega \sqsubseteq d; d^\omega\) by (metis inf.absorb-iff2 inf-left-commute inf-seq-distrib)
  - then have \(\text{nil} \sqcap c; d^\omega \sqsubseteq d^\omega\) by (metis inf.bounded-iff inf-sup-ord(1) iter-unfold)
  - thus \(?thesis\) by (simp add: iter-induct-nil)

**QED**

**Lemma iter-abort:** \(\bot = \text{nil}^\omega\)
  by (simp add: antisym iter-induct-nil)

**Lemma nil-iter:** \(\top^\omega = \text{nil}\)
  by (metis (no-types) inf-top.right-neutral iter-unfold seq-top)
8.2 Finite iteration

Iteration of a finite number of steps (Kleene star) is defined using the greatest fixed point.

```plaintext
locale finite-iteration = seq-distrib + lower-galois-connections
begin

definition
fiter :: 'a ⇒ 'a (⋆ [101] 100)
where
  c⋆ ≡ gfp (λ x. nil ⊓ c;x)

lemma fin-iter-step-mono: mono (λ x. nil ⊓ c;x)
  by (meson inf-mono order-refl seq-mono-right mono-def)

This definition leads to the two core iteration lemmas: folding and induction.

lemma fiter-unfold: c⋆ = nil ⊓ c;c⋆
  using fiter-def gfp-unfold fin-iter-step-mono by auto

lemma fiter-induct-nil: x ⊑ nil ⊓ c;x =⇒ x ⊑ c⋆
  by (simp add: fiter-def gfp-upperbound)

lemma fiter0: c⋆ ⊑ nil
  by (metis fiter0 fiter-unfold inf-le2 order.trans seq-mono-right seq-nil-right)

lemma fiter1: c⋆ ⊑ c
  by (metis fiter0 fiter-unfold inf.le2 order.trans seq-mono-right seq-nil-right)

lemma fiter-induct-eq: c⋆;d = gfp (λ x. c;x ⊓ d)
proof —
  define F where F = (λ x. x;d)
  define G where G = (λ x. nil ⊓ c;x)
  define H where H = (λ x. c;x ⊓ d)

  have FG: F ∘ G = (λ x. c;x;d ⊓ d) by (simp add: F-def G-def comp-def inf-commute inf-seq-distrib)
  have HF: H ∘ F = (λ x. c;x;d ⊓ d) by (metis comp-def seq-assoc H-def F-def)

  have adjoint: dist-over-inf F using Inf-seq-distrib F-def by simp
```
have monoH: mono H
  by (metis H-def inf-mono-left monoI seq-distrib-left seq-mono-right seq-distrib-left-axioms)
have monoG: mono G by (metis G-def inf-mono-right mono-def seq-mono-right)
have \( \forall x. ((F \circ G) x = (H \circ F) x) \) using FG HF by simp
then have \( F \) (gfp G) = gfp H using adjoint monoG monoH fusion-gfp-eq by blast
then have \( \lambda x. (\text{nil} \sqcap c; x); d = \text{gfp} (\lambda x. c; x \sqcap d) \) using F-def G-def H-def
inf-commute by simp
  thus \( \text{thesis} \) by (metis fiter-def)
qed

theorem fiter-induct: \( x \sqsubseteq d \sqcap c; x \rightarrow x \sqsubseteq c^*; d \)
proof
  assume \( x \sqsubseteq d \sqcap c; x \)
  then have \( x \sqsubseteq c; x \sqcap d \) using inf-commute by simp
  then have \( x \sqsubseteq \text{gfp} (\lambda x. c; x \sqcap d) \) by (simp add: gfp-upperbound)
  thus \( \text{thesis} \) by (metis (full-types) fiter-induct-eq)
qed

lemma fiter2 [simp]: \( c^*; c^* = c^* \)
proof
  have lr: \( c^*; c^* \sqsubseteq c^* \) using fiter0 seq-mono-right seq-nil-right by fastforce
  have rl: \( c^* \sqsubseteq c^*; c^* \) by (metis fiter-induct fiter-unfold inf_right-idem order-refl)
  thus \( \text{thesis} \) by (simp add: antisym lr)
qed

lemma fiter3 [simp]: \( (c^*)^* = c^* \)
by (metis dual-order.refl fiter0 fiter1 fiter2 fiter-induct eq.commute eq.absorb1 seq-nil-right)

lemma fiter-mono: \( c \sqsubseteq d \rightarrow c^* \sqsubseteq d^* \)
proof
  assume \( c \sqsubseteq d \)
  then have \( c^* \sqsubseteq \text{nil} \sqcap d; c^* \) by (metis fiter0 fiter1 inf.bounded_iff refine-trans seq-mono-left)
  thus \( \text{thesis} \) by (metis seq-nil-right fiter-induct)
qed

end

8.3 Infinite iteration

Iteration of infinite number of steps can be defined using a least fixed point.

locale infinite-iteration = seq-distrib + lower-galois-connections
begin
**definition**

infiter :: 'a ⇒ 'a (\(-\infty \to 105\) 106)

**where**

c^\infty \equiv \text{lfp} (\lambda x. c;x)

**lemma** infiter-step-mono: mono (\lambda x. c;x)

by (meson inf-mono order-refl seq-mono-right mono-def)

This definition leads to the two core iteration lemmas: folding and induction.

**theorem** infiter-unfold: c^\infty = c;c^\infty

using infiter-def infiter-step-mono lfp-unfold by auto

**lemma** infiter-induct: c;x ⊑ x ⇒ c^\infty ⊑ x

by (simp add: infiter-def lfp-lowerbound)

**theorem** infiter-unfold-any: c^\infty = (c ^ i) ; c^\infty

**proof** (induct i)

thus ?case by simp

next

case (Suc i)

thus ?case using infiter-unfold seq-assoc seq-power-Suc by auto

qed

**lemma** infiter-annil: c^\infty;x = c^\infty

**proof**

have \forall a. (\bot ::'a) ⊑ a

by auto

thus ?thesis

by (metis (no-types) eq-iff inf.cobounded2 infiter-induct infiter-unfold inf-sup-ord(1)
seq-assoc seq-bot weak-seq-inf-distrib seq-nil-right)

qed

end

### 8.4 Combined iteration

The three different iteration operators can be combined to show that finite iteration refines finite-or-infinite iteration.

**locale** iteration = finite-or-infinite-iteration + finite-iteration +

infinite-iteration

begin
lemma refine-iter: $c^\omega \sqsubset c^*$
  by (metis seq-nil-right order.refl iter-unfold fiter-induct)

lemma iter-absorption [simp]: $(c^\omega)^* = c^\omega$
proof (rule antisym)
  show $(c^\omega)^* \sqsubseteq c^\omega$ by (metis fiter1)
next
  show $c^\omega \sqsubseteq (c^\omega)^*$ by (metis fiter1 fiter-induct inf-left-idem iter2 iter-unfold seq-nil-right sup.cobounded2 sup.orderE sup-commute)
qed

lemma infiter-inf-top: $c^\infty = c^\omega ; \top$
proof –
  have lr: $c^\infty \sqsubseteq c^\omega ; \top$
  proof –
  have $c ; (c^\omega ; \top) = \mathit{nil} ; \top \sqcap c ; c^\omega ; \top$
    using semigroup.assoc seq.semigroup-axioms by fastforce
  then show ?thesis
  by (metis (no-types) eq-refl finite-or-infinite-iteration.iter-unfold
      finite-or-infinite-iteration-axioms fiter-induct
      seq-distrib-right.inf-seq-distrib seq-distrib-right-axioms)
qed

have rl: $c^\omega ; \top \sqsubseteq c^\infty$
  by (metis inf-le2 infiter-annil infiter-unfold iter-induct-nil seq-mono-left)
  thus ?thesis using antisym-conv lr by blast
qed

lemma infiter-fiter-top:
  shows $c^\infty \sqsubseteq c^* ; \top$
  by (metis eq-iff fiter-induct inf-top-left infiter-unfold)

lemma inf-ref-infiter: $c^\omega \sqsubseteq c^\infty$
  using infiter-unfold iter-induct-nil by auto

end

end

\section{Sequential composition for conjunctive models}

theory Conjunctive-Sequential
imports Sequential

26
Sequential left-distributivity is only supported by conjunctive models but does not apply in general. The relational model is one such example.

locale seq-finite-conjunctive = seq-distrib-right +
  assumes seq-inf-distrib: c:(d₀ ∩ d₁) = c:d₀ ∩ c:d₁
begin

sublocale seq-distrib-left
by (simp add: seq-distrib-left.intro seq-distrib-left-axioms.intro seq-inf-distrib sequential-axioms)
end

locale seq-infinite-conjunctive = seq-distrib-right +
  assumes seq-Inf-distrib: D ≠ {} ⇒ c ; ∩ D = (∩ d∈D. c ; d)
begin

sublocale seq-distrib
proof unfold-locales
fix c::'a and d₀::'a and d₁::'a
have {d₀, d₁} ≠ {} by simp
then have c : ∩ {d₀, d₁} = ∩ {c ; d | d ∈ {d₀, d₁}} using seq-Inf-distrib
proof
  have ∩ ((c::'a::refinement-lattice) D::'a::refinement-lattice set. D ≠ {} ⇒ c ; ∩ D = (∩ d::'a::refinement-lattice∈D. c ; d) ∩ {d₀::'a::refinement-lattice, d₁::'a::refinement-lattice} ≠ {} ) by presburger
  qed
  also have ... = c ; d₀ ∩ c ; d₁ by (simp only: Inf2-inf)
  finally show ?thesis by simp
qed

lemma seq-INF-distrib: X ≠ {} ⇒ c ; (∩ x∈X. d x) = (∩ x∈X. c ; d x)
proof
  assume xne: X ≠ {}
  have a: c ; (∩ x∈X. d x) = c ; ∩ (d ' X) by auto
  also have b: ... = (∩ d∈(d ' X). c ; d) by (meson image-is-empty seq-Inf-distrib xne)
  also have c: ... = (∩ x∈X. c ; d x) by (simp add: image-comp)
  finally show ?thesis by (simp add: b image-comp)

27
qed

lemma seq-INF-distrib-UNIV: \( c \colon (\prod x. d x) = (\prod x. c \cdot d x) \)
by (simp add: seq-INF-distrib)

lemma INF-INF-seq-distrib: \( Y \neq \{\} \implies (\prod x. c x) ; (\prod y \in Y. d y) = (\prod x \in X. \prod y \in Y. c x \cdot d y) \)
by (simp add: INF-seq-distrib seq-INF-distrib)

lemma INF-INF-seq-distrib-UNIV: \( \prod x. c x) ; (\prod y \in Y. d y) = (\prod x \in X. \prod y \in Y. c x ; d y) \)
by (simp add: INF-INF-seq-distrib)

end
end

10 Infimum nat lemmas

theory Infimum-Nat
imports Refinement-Lattice
begin

locale infimum-nat
begin

lemma INF-partition-nat3:
fixes f :: nat \Rightarrow nat \Rightarrow 'a::refinement-lattice
shows \( (\prod j \in \{ j. i = j \} \cup \{ j. i < j \} \cup \{ j. j < i \}. f i j) \)
proof
have univ-part: \( UNIV = \{ j. i = j \} \cup \{ j. i < j \} \cup \{ j. j < i \} \) by auto
have \( (\prod j \in \{ j. i = j \} \cup \{ j. i < j \} \cup \{ j. j < i \}. f i j) = \)
\( (\prod j \in \{ j. i = j \}. f i j) \cap \)
\( (\prod j \in \{ j. i < j \}. f i j) \cap \)
\( (\prod j \in \{ j. j < i \}. f i j) \) by (metis INF-union)
with univ-part show thesis by simp
qed

lemma INF-INF-partition-nat3:
fixes f :: nat \Rightarrow nat \Rightarrow 'a::refinement-lattice
proof -
  have \((\prod i. \prod j. f i j) = (\prod i. (\prod j(\mid j = i\mid). f i j))\) \n    by (simp add: INF-partition-nat3)
  also have \(\ldots = ((\prod i. (\prod j(\mid j = i\mid). f i j))\) \n    by (simp add: INF-inf-distrib)
  finally show ?thesis.
qed

lemma INF-nat-minus:
  \((\prod i\in\{0 < i\}. f i) = (\prod i. f (Suc i))\) \n  by (metis greaterThan-0 greaterThan-def range-composition)

lemma INF-nat-plus:
  fixes \(f : \mathbb{N} \Rightarrow \alpha \Rightarrow \text{refinement-lattice}\)
  shows \((\prod j\mid j < i\mid). f (j - i) = (\prod k\mid k < i\mid). f k)\)
  apply (rule antisym)
  apply (rule INF-mono, simp)
  apply (metis add.right-neutral add-diff-cancel-left add-less-cancel-left order-refl)
  apply (rule INF-mono, simp)
  by (meson order-refl zero-less-diff)

lemma INF-INF-guarded-switch:
  fixes \(f : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \alpha \Rightarrow \text{refinement-lattice}\)
  shows \((\prod i. \prod j\mid j < i\mid). f j (i - j) = (\prod i. \prod j\mid j < i\mid). f j (i - j))\)
  proof (rule antisym)
    have \((\forall j i. j < i \implies \exists i. \exists j<i. f j (i - j) \subseteq f j (ii - jj))\)
      by blast
    then have \((\forall j i. j < i \implies \exists i. \prod j\mid j < i\mid). f j (i - j) \subseteq f j (ii - jj))\)
      by (meson INF-lower mem-Collect-eq)
    then have \((\forall j i. j < i \implies (\prod i. \prod j\mid j < i\mid). f j (i - j) \subseteq f jj (ii - jj))\)
      by (meson reflexive INF-lower dual-order.trans)
    then have \((\forall j i. (\prod i. \prod j\mid j < i\mid). f j (i - j) \subseteq (\prod ii\mid ii < ii\mid). f jj (ii - jj))\)
      by (metis (mono-tags, lifting) INF-greatest mem-Collect-eq)
    then have \((\prod i. \prod j\mid j < i\mid). f j (i - j) \subseteq (\prod jj. \prod ii\mid ii < ii\mid). f jj (ii - jj))\)
      by (simp add: INF-greatest)
then show \((\prod i. \prod j \in \{j. j < i\}. f j (i - j)) \subseteq (\prod j. \prod i \in \{i. j < i\}. f j (i - j))\)
by simp

next
have \(\\land ii jj. jj < ii \Longrightarrow \exists j. \exists i > j. f j (i - j) \subseteq f jj (ii - jj)\)
by blast
then have \(\\land ii jj. jj < ii \Longrightarrow (\prod i. \prod j \in \{i. j < i\}. f j (i - j)) \subseteq f jj (ii - jj)\)
by (meson INF-lower mem-Collect-eq)
then have \(\\land ii jj. jj < ii \Longrightarrow (\prod j. \prod i \in \{i. j < i\}. f j (i - j)) \subseteq f jj (ii - jj)\)
by (meson UNIV-I INF-lower dual-order-trans)
then have \(\\land ii jj. jj < ii \Longrightarrow (\prod j. \prod i \in \{i. j < i\}. f j (i - j)) \subseteq f jj (ii - jj)\)
by (simp add: INF-greatest)
then have \(\\land ii jj. jj < ii \Longrightarrow (\prod j. \prod i \in \{i. j < i\}. f j (i - j)) \subseteq f jj (ii - jj)\)
by simp
qed

end

end

11 Iteration for conjunctive models

theory Conjunctive-Iteration
imports
  Conjunctive-Sequential
  Iteration
  Infimum-Nat
begin
Sequential left-distributivity is only supported by conjunctive models but does not apply in general. The relational model is one such example.
locale iteration-finite-conjunctive = seq-finite-conjunctive + iteration
begin
lemma isolation: \(c^\omega = c^* \cap c^\infty\)
proof
  define \(F\) where \(F = (\lambda x. c^* \cap x)\)
  define \(G\) where \(G = (\lambda x. c x)\)
  define \(H\) where \(H = (\lambda x. \text{nil} \cap c x)\)

  have \(FG: F \circ G = (\lambda x. c^* \cap c x)\) using F-def G-def by auto

end
have $HF: H \circ F = (\lambda x. \text{nil} \sqcap c;(c^* \sqcap x))$ using $F$-def $H$-def by auto

have adjoint: dist-over-sup $F$ by (simp add: $F$-def inf-Sup)
have mono$H$: mono $H$ by (metis $H$-def inf-mono monoI order-refl seq-mono-right)
have mono$G$: mono $G$ by (metis $G$-def inf.absorb-iff2 monoI seq-inf-distrib)

have $\forall x. ((F \circ G) x = (H \circ F) x)$ using $FG$ $HF$
  by (metis fiter-unfold inf-sup-aci(2) seq-inf-distrib)
then have $F (\text{lfp } G) = \text{lfp } H$ using adjoint mono$H$ mono$G$ fusion-lfp-eq by blast
then have $c^* \sqcap \text{lfp } (\lambda x. c;x) = \text{lfp } (\lambda x. \text{nil} \sqcap c;x)$
  using $F$-def $G$-def $H$-def by blast
thus $\omega$thesis by (simp add: infiter-def iter-def)
qed

lemma iter-induct-isolate: $c^*;d \sqcap c^\omega = \text{lfp } (\lambda x. d \sqcap c;x)$
proof –
define $F$ where $F = (\lambda x. c^*;d \sqcap x)$
define $G$ where $G = (\lambda x. c;x)$
define $H$ where $H = (\lambda x. d \sqcap c;x)$

have $FG$: $F \circ G = (\lambda x. c^*;d \sqcap c;x)$ using $F$-def $G$-def by auto
have $HF$: $H \circ F = (\lambda x. d \sqcap c;c^*;d \sqcap c;x)$ using $F$-def $H$-def weak-seq-inf-distrib
  by (metis comp-apply inf.commute inf.left-commute seq-assoc seq-inf-distrib)
have unroll: $c^*;d = (\text{nil} \sqcap c;c^*);d$ using fiter-unfold by auto
have distribute: $c^*;d = d \sqcap c;c^*;d$ by (simp add: unroll inf-seq-distrib)
have $FGx$: $(F \circ G) x = d \sqcap c;c^*;d \sqcap c;x$ using $FG$ distribute by simp

have adjoint: dist-over-sup $F$ by (simp add: $F$-def inf-Sup)
have mono$H$: mono $H$ by (metis $H$-def inf-mono monoI order-refl seq-mono-right)
have mono$G$: mono $G$ by (metis $G$-def inf.absorb-iff2 monoI seq-inf-distrib)

have $\forall x. ((F \circ G) x = (H \circ F) x)$ using $FGx$ $HF$ by (simp add: $FG$ distribute)
then have $F (\text{lfp } G) = \text{lfp } H$ using adjoint mono$H$ mono$G$ fusion-lfp-eq by blast
then have $c^*;d \sqcap \text{lfp } (\lambda x. c;x) = \text{lfp } (\lambda x. d \sqcap c;x)$
  using $F$-def $G$-def $H$-def by blast
thus $\omega$thesis by (simp add: infiter-def)
qed

lemma iter-induct-eq: $c^\omega;d = \text{lfp } (\lambda x. d \sqcap c;x)$
proof –
  have $c^\omega;d = c^*;d \sqcap c^\omega;d$ by (simp add: isolation inf-seq-distrib)
  then have $c^*;d \sqcap c^\omega;d = c^*;d \sqcap c^\omega$ by (simp add: infiter-annil)
  then have $c^*;d \sqcap c^\omega = \text{lfp } (\lambda x. d \sqcap c;x)$ by (simp add: iter-induct-isolate)
thus \( ?\text{thesis} \)
by (simp add: \( \langle c^\omega \rangle ; d = c^* ; d \sqcap c^\omega ; d \rangle , \langle c^* ; d \sqcap c^\omega ; d = c^* ; d \sqcap c^\omega \rangle \))

dqed

lemma iter-induct: \( d \sqcap c ; x \sqsubseteq x \implies c^\omega ; d \sqsubseteq x \)
by (simp add: iter-induct-eq lfp-lowerbound)

lemma iter-isolate: \( c^* ; d \sqcap c^\omega = c^\omega ; d \)
by (simp add: iter-induct-eq iter-induct-isolate)

lemma iter-isolate2: \( c ; c^* ; d \sqcap c^\omega = c ; c^\omega ; d \)
by (metis infiter-unfold iter-isolate seq-assoc seq-inf-distrib)

lemma iter-decomp: \( (c \sqcap d)^\omega = c^\omega ; (d ; c^\omega)^\omega \)
proof (rule antisym)
  have \( c ; c^\omega ; (d ; c^\omega)^\omega \sqcap (d ; c^\omega)^\omega \subseteq c^\omega ; (d ; c^\omega)^\omega \)
    by (metis inf-commute order refl inf_seq-distrib seq-nil-left iter-unfold)
  thus \( (c \sqcap d)^\omega \subseteq c^\omega ; (d ; c^\omega)^\omega \)
    by (metis inf . left-commute iter-induct-nil iter-unfold seq-assoc inf_seq-distrib)
next
  have \( (c ; (c \sqcap d)^\omega \sqcap (c \sqcap d)^\omega) \sqsubseteq (c \sqcap d)^\omega \)
    by (metis inf-commute order refl inf_seq-distrib iter-unfold)
  then have \( a ; c^\omega ; (d ; (c \sqcap d)^\omega \sqcap \text{nil}) \subseteq (c \sqcap d)^\omega \)
    by (metis inf . left-commute iter-induct-nil iter-unfold seq-assoc inf_seq-distrib)
  proof
    have \( \text{nil} \sqcap d ; (c \sqcap d)^\omega \sqcap c ; (c \sqcap d)^\omega \subseteq (c \sqcap d)^\omega \)
      by (metis eq_iff inf . semigroup_axioms inf_commute inf_seq_distrib iter_unfold semi_group_assoc)
    thus \( ?\text{thesis} \) using iter_induct_eq by (metis inf_sup_aci(1) iter_induct)
  qed
then have \( d ; c^\omega ; (d ; (c \sqcap d)^\omega \sqcap \text{nil}) \sqsubseteq (c \sqcap d)^\omega \sqcap \text{nil} \)
  by (metis inf_monoid order refl seq_assoc seq_mono)
then have \( (d ; c^\omega)^\omega \sqsubseteq (c \sqcap d)^\omega \sqcap \text{nil} \)
  by (metis inf_commute iter_induct_nil)
then have \( c^\omega ; (d ; c^\omega)^\omega \sqsubseteq c^\omega ; (d ; (c \sqcap d)^\omega \sqcap \text{nil}) \)
  by (metis order refl seq_mono)
thus \( c^\omega ; (d ; c^\omega)^\omega \sqsubseteq (c \sqcap d)^\omega \) using a refine_trans by blast

dqed

lemma iter-leapfrog-var: \( (c ; d)^\omega ; c \subseteq c ; (d ; c)^\omega \)
proof
  have \( c \sqcap d ; c ; (d ; c)^\omega \subseteq c ; (d ; c)^\omega \)
    by (metis iter_unfold order_refl seq_assoc seq_inf_distrib seq_nil_right)
  thus \( ?\text{thesis} \) using iter_induct_eq by (metis iter_induct seq_assoc)
  qed

32
lemma iter-leapfrog: $c; (d; c)^\omega = (c; d)^\omega ; c$

proof (rule antisym)
  show $(c; d)^\omega ; c \sqsubseteq c; (d; c)^\omega$ by (metis iter-leapfrog-var)
next
  have $(d; c)^\omega \sqsubseteq ((d; c)^\omega ; d); c \sqcap \text{nil}$ by (metis inf . bounded-iff order refl seq-assoc seq-mono iter-unfold iter1 iter2)
  then have $(d; c)^\omega \sqsubseteq (d; (c; d)^\omega); c \sqcap \text{nil}$ by (metis inf . absorb-iff2 inf . boundedE inf-assoc iter-leapfrog-var inf-seq-distrib)
  then have $c; (d; c)^\omega \sqsubseteq c; d; (c; d)^\omega; c \sqcap \text{nil}; c$ using inf . bounded-iff seq-assoc seq-mono-right seq-nil-left seq-nil-right by fastforce
  thus $c; (d; c)^\omega \sqsubseteq (c; d)^\omega ; c$ by (metis inf-commute inf-seq-distrib iter-unfold)
qed

lemma fiter-leapfrog: $c; (d; c)^* = (c; d)^* ; c$

proof –
  have lr: $c; (d; c)^* \sqsubseteq (c; d)^* ; c$
proof –
  have $(d; c)^* = \text{nil} \sqcap d; c; (d; c)^*$
    by (meson finite-iteration.fiter-unfold finite-iteration-axioms)
  then show ?thesis
    by (metis fiter-induct seq-assoc seq-distrib-left weak-seq-inf-distrib seq-distrib-left-axioms seq-nil-right)
qed

have rl: $(c; d)^* ; c \sqsubseteq c; (d; c)^*$

proof –
  have al: $(c; d)^* ; c = c \sqcap c; d; (c; d)^* ; c$
    by (metis finite-iteration.fiter-unfold finite-iteration-axioms inf-seq-distrib seq-nil-left)
  have a2: $(c; d)^* ; c \sqsubseteq c; (d; c)^* \rightarrow c \sqcap c; d; (c; d)^* ; c \sqsubseteq c; (d; c)^*$ by (simp add: a1)
  then have a3: $(c; \text{nil} \sqcap d; (c; d)^* ; c) \sqsubseteq c; (d; c)^*$
    by (metis a1 eq-iff fiter-unfold lr seq-assoc seq-inf-distrib seq-nil-right)
  have a4: $(\text{nil} \sqcap d; (c; d)^* ; c) \sqsubseteq (d; c)^* \rightarrow \text{c}; (\text{nil} \sqcap d; (c; d)^* ; c) \sqsubseteq c; (d; c)^*$
    using seq-mono-right by blast
  have a5: $(\text{nil} \sqcap d; (c; d)^* ; c) \sqsubseteq (d; c)^*$
proof –
  have f1: $d; (c; d)^* ; c \sqcap \text{nil} = d; ((c; d)^* ; c) \sqcap \text{nil} \sqcap \text{nil}$
    by (simp add: seq-assoc)
  have $d; (c; d)^* ; c \sqcap \text{nil} = d; ((c; d)^* ; c)$
    by (metis (no-types) a1 inf-sup-aci(I) seq-assoc seq-finite-conjunctive.seq-inf-distrib seq-finite-conjunctive-axioms seq-nil-right)
  then show ?thesis
    using f1 by (metis (no-types) finite-iteration.fiter-induct finite-iteration-axioms
locale iteration-infinite-conjunctive = seq-infinite-conjunctive + iteration + infimum-nat

begin

lemma fiter-seq-choice: \( c^* = (\bigcap i : \text{nat}. \ c :^i) \)
proof (rule antisym)
  show \( c^* \subseteq (\bigcap i. \ c :^i) \)
  proof (rule INF-greatest)
    fix \( i \)
    show \( c^* \subseteq c :^i \)
    proof (induct \( i \) type: nat)
      case 0
      show \( c^* \subseteq c :^0 \) by (simp add: fiter0)
    next
      case (Suc \( n \))
      have \( c^* \subseteq c ; c^* \) by (metis fiter-unfold inf-le2)
      also have \( ... \subseteq c ; (c :^n) \) using Suc.hyps by (simp only: seq-mono-right)
      also have \( ... = c :^n \) Suc \( n \) by simp
      finally show \( c^* \subseteq c :^n \).
    qed
  qed
next
  have \( (\bigcap i. \ c :^i) \subseteq (c :^0) \cap (\bigcap i. \ c :^\text{Suc} \ i) \)
  by (meson INF-greatest INF-lower UNIV-I le-inf-iff)
  also have \( ... = \text{nil} \cap (\bigcap i. \ c ; (c :^i)) \) by simp
  also have \( ... = \text{nil} \cap c ; (\bigcap i. \ c :^i) \) by (simp add: seq-INF-distrib)
  finally show \( (\bigcap i. \ c :^i) \subseteq c^* \) using fiter-induct by fastforce
qed

lemma fiter-seq-choice-nonempty: \( c ; c^* = (\bigcap i \in \{i. \ 0 < i\} . \ c :^i) \)
proof --
  have \( (\bigcap i \in \{i. \ 0 < i\} . \ c :^i) = (\bigcap i. \ c :^i \text{ (Suc } i)) \) by (simp add: INF-nat-shift)
  also have \( ... = (\bigcap i. \ c ; (c :^i)) \) by simp
also have ... = c ; (\bigcap i. c ; ^ i) by (simp add: seq-INF-distrib-UNIV)
also have ... = c ; c* by (simp add: fiter-seq-choice)
finally show ?thesis by simp
qed

locale conj-iteration = cra + iteration-infinite-conjunctive
begin

lemma conj-distrib4: c* \bigcap d* \subseteq (c \bigcap d)*
proof -
  have c* \bigcap d* = (nil \bigcap (c;c*)) \bigcap d* by (metis fiter-unfold)
  then have c* \bigcap d* = (nil \bigcap d*) \bigcap ((c;c*) \bigcap d*) by (simp add: inf-conj-distrib)
  then have c* \bigcap d* \subseteq nil \bigcap ((c;c*) \bigcap (d;d*)) by (metis conj-idem fiter0 fiter-unfold inf.bounded-iff inf-le2 local.conj-mono)
  then have c* \bigcap d* \subseteq ni \bigcap ((c \bigcap d);(c* \bigcap d*)) by (meson inf-mono-right order.trans sequential-interchange)
  thus ?thesis by (metis seq-nil-right fiter-induct)
qed

end

end

12 Rely Quotient Operator

The rely quotient operator is used to generalise a Jones-style rely condition to a
process [5]. It is defined in terms of the parallel operator and a process \( i \) repre-
senting interference from the environment.

theory Rely-Quotient
imports
  CRA
  Conjunctive-Iteration
begin

12.1 Basic rely quotient

The rely quotient of a process \( c \) and an interference process \( i \) is the most general
process \( d \) such that \( c \) is refined by \( d \parallel i \). The following locale introduces the
definition of the rely quotient $c // i$ as a non-deterministic choice over all processes $d$ such that $c$ is refined by $d \parallel i$.

locale rely-quotient = par-distrib + conjunction-parallel

begin

definition rely-quotient :: $\tau \Rightarrow $ $\tau \Rightarrow $ $\tau$ (infixl '/') 85

where

$c // i \equiv \bigsqcap \{ d. (c \sqsubseteq d \parallel i) \}$

Any process $c$ is implemented by itself if the interference is skip.

lemma quotient-identity: $c // skip = c$

proof

have $c // skip = \bigsqcap \{ d. (c \sqsubseteq d \parallel skip) \}$ by (metis rely-quotient-def)
then have $c // skip = \bigsqcap \{ d. (c \sqsubseteq d) \}$ by (metis mono-tags, lifting) Collect-cong par-skip
thus thesis by (metis Inf-greatest Inf-lower2 dual-order.antisym dual-order.refl mem-Collect-eq)

qed

Provided the interference process $i$ is non-aborting (i.e. it refines chaos), any process $c$ is refined by its rely quotient with $i$ in parallel with $i$. If interference $i$ was allowed to be aborting then, because $(c // \bot) \parallel \bot$ equals $\bot$, it does not refine $c$ in general.

theorem rely-quotient:

assumes nonabort-i: chaos $\sqsubseteq i$

shows $c \sqsubseteq (c // i) \parallel i$

proof

define $D$ where $D = \{ d \parallel i \mid d. (c \sqsubseteq d \parallel i) \}$
define $C$ where $C = \{ c \}$

have $(\forall d \in D. (\exists c \in C. c \sqsubseteq d))$ using D-def C-def CollectD singletonI by auto
then have $\bigsqcap C \subseteq (\bigsqcap D)$ by (simp add: Inf-mono)
then have $c \subseteq \bigsqcap \{ d \parallel i \mid d. (c \sqsubseteq d \parallel i) \}$ by (simp add: C-def D-def)
also have $... = \bigsqcap \{ d \parallel i \mid d. (c \sqsubseteq d \parallel i) \}$ by simp
also have $... = \bigsqcap \{ d \mid d. (c \sqsubseteq d \parallel i) \} \parallel i$ by (simp add: INF-Inf)
also have $... = \bigsqcap \{ d \mid d. (c \sqsubseteq d \parallel i) \} \parallel i$

proof (cases $\{ d \mid d. (c \sqsubseteq d \parallel i) \} = \{ \}$)

assume $\{ d \mid d. (c \sqsubseteq d \parallel i) \} = \{ \}$
then show $(\bigsqcap d \in \{ d. (c \sqsubseteq d \parallel i) \}. d \parallel i) = \bigsqcap \{ d \mid d. (c \sqsubseteq d \parallel i) \} \parallel i$ using nonabort-i Collect-empty-eq top-greatest nonabort-par-top par-commute by fastforce

next
assume $a$: $\{ d \mid d. (c \sqsubseteq d \parallel i) \} \neq \{ \}$

36
have $b$: \{$d. (c \sqsubseteq d \parallel i)$\} $\neq \emptyset$ using $a$ by blast
then have $(\bigsqcap \{d. (c \sqsubseteq d \parallel i)\}. d \parallel i) = \bigsqcap \{d. (c \sqsubseteq d \parallel i)\} \parallel i$
  using Inf-par-distrib by simp
then show \(?thesis\) by auto
qed
also have \(\ldots = (c /\!\!/ i) \parallel i\) by (metis rely-quotient-def)
finally show \(?thesis\).
qed

The following theorem represents the Galois connection between the parallel operator (upper adjoint) and the rely quotient operator (lower adjoint). This basic relationship is used to prove the majority of the theorems about rely quotient.

**theorem** rely-refinement:
  **assumes** nonabort-i: chaos $\sqsubseteq i$
  **shows** $c /\!\!/ i \sqsubseteq d$ $\iff c \sqsubseteq d \parallel i$

**proof**
assume $a$: $c /\!\!/ i \sqsubseteq d$

have $c \sqsubseteq (c /\!\!/ i) \parallel i$ using rely-quotient nonabort-i by simp

thus $c \sqsubseteq d \parallel i$ using par-mono $a$
  by (metis inf.absorb-iff2 inf-commute le-infI2 order-refl)

next

assume $b$: $c \sqsubseteq d \parallel i$

then have $\bigsqcap \{d. (c \sqsubseteq d \parallel i)\} \subseteq d$ by (simp add: Inf-lower)

thus $c /\!\!/ i \sqsubseteq d$ by (metis rely-quotient-def)

qed

Refining the “numerator” in a quotient, refines the quotient.

**lemma** rely-mono:
  **assumes** c-refsto-d: $c \sqsubseteq d$
  **shows** $(c /\!\!/ i) \sqsubseteq (d /\!\!/ i)$

**proof**

have $\bigsqcap f. ((d \sqsubseteq f \parallel i) \implies \exists e. (c \sqsubseteq e \parallel i) \land (e \sqsubseteq f))$
  using c-refsto-d order.trans by blast

then have $b$: $\bigsqcap \{e. (c \sqsubseteq e \parallel i)\} \subseteq \bigsqcap \{f. (d \sqsubseteq f \parallel i)\}$
  by (metis Inf-mono mem-Collect-eq)

show \(?thesis\) using rely-quotient-def $b$ by simp

qed

Refining the “denominator” in a quotient, gives a reverse refinement for the quotients. This corresponds to weaken rely condition law of Jones [5], i.e. assuming less about the environment.

**lemma** weaken-rely:
  **assumes** i-refsto-j: $i \sqsubseteq j$
shows \((c // j) \subseteq (c // i)\)

proof

- have \(\\wedge f. ((c \subseteq f \parallel i) \implies \exists e. (c \subseteq e \parallel j) \land (e \subseteq f))\)
  using i-refsto-j order_trans
  by (metis inf.absorb_iff2 inf-le1 inf-par-distrib inf-sup-ord(2) par-commute)

- then have \(b: \bigcap\{ e. (c \subseteq e \parallel j)\} \subseteq \bigcap\{ f. (c \subseteq f \parallel i)\}\)
  by (metis Inf-mono mem-Collect-eq)

- show \(?thesis using rely-quotient-def b by simp\)

qed

theorem par-nonabort:

assumes nonabort-i: chaos \subseteq i

assumes nonabort-j: chaos \subseteq j

shows chaos \subseteq i \parallel j

by (meson chaos-par-chaos nonabort-i nonabort-j order-trans par-mono)

Nesting rely quotients of \(j\) and \(i\) means the same as a single quotient which is the parallel composition of \(i\) and \(j\).

lemma nested-rely:

assumes j-nonabort: chaos \subseteq j

shows \((c // j) // i = c // (i \parallel j)\)

proof (rule antisym)

- show \((c // j) // i \subseteq c // (i \parallel j)\)

  proof
  - have \(\\wedge f. ((c \subseteq f \parallel i \parallel j) \implies \exists e. (c \subseteq e \parallel j) \land (e \subseteq f \parallel i))\) by blast
  - then have \(\bigcap\{ d. (\bigcap\{ e. (c \subseteq e \parallel j)\} \subseteq d \parallel i)\} \subseteq \bigcap\{ f. (c \subseteq f \parallel i \parallel j)\}\)
    by (simp add: Collect_mono Inf_lower Inf_superset_mono)
  - thus \(?thesis using local.rely-quotient-def par-assoc by auto\)

  qed

next

- show \(c // (i \parallel j) \subseteq (c // j) // i\)

  proof
  - have \(c \subseteq \bigcap\{ e. (c \subseteq e \parallel j)\} \parallel j\)
    using j-nonabort local.rely-quotient-def rely-quotient by auto
  - then have \(\land d. (\bigcap\{ e. (c \subseteq e \parallel j)\} \subseteq d \parallel i \implies (c \subseteq d \parallel i \parallel j)\)
    by (meson j-nonabort order_trans rely-refinement)
  - thus \(?thesis using simp add: Collect_mono Inf-superset_mono local.rely-quotient-def par-assoc\)

  qed

qed

end
12.2 Distributed rely quotient

locale rely-distrib = rely-quotient + conjunction-sequential
begin

The following is a fundamental law for introducing a parallel composition of process to refine a conjunction of specifications. It represents an abstract view of the parallel introduction law of Jones [5].

lemma introduce-parallel:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  shows c ⊓ d ⊑ (j ⊓ (c // i)) || (i ⊓ (d // j))
proof
  have a: c ⊑ (c // i) || i using nonabort-i nonabort-j rely-quotient
  by auto
  have b: d ⊑ j || (d // j) using rely-quotient par-commute
  by (simp add: nonabort-j)
  have c ⊓ d ⊑ ((c // i) ⊓ j || (i ⊓ (d // j))
  using parallel-interchange refine-trans calculation by blast
  also have interchange: c ⊓ d ⊑ ((c // i) ⊓ (d // j))
  using a b by (metis conj-mono)
  thus ?thesis using interchange by (simp add: local.conj-commute)
qed

Rely quotients satisfy a range of distribution properties with respect to the other operators.

lemma distribute-rely-conjunction:
  assumes nonabort-i: chaos ⊑ i
  shows (c ⊓ d) // i ⊑ (c // i) ⊓ (d // i)
proof
  have c ⊓ d ⊑ ((c // i) || i) ⊓ ((d // i) || i) using conj-mono rely-quotient
  by (simp add: nonabort-i)
  then have c ⊓ d ⊑ ((c // i) ⊓ i) || (i ⊓ i)
  by (metis parallel-interchange refine-trans)
  then have c ⊓ d ⊑ ((c // i) ⊓ (d // i)) || i by (metis conj-idem)
  thus ?thesis using rely-refinement by (simp add: nonabort-i)
qed

lemma distribute-rely-choice:
  assumes nonabort-i: chaos ⊑ i
  shows (c ⊓ d) // i ⊑ (c // i) ⊓ (d // i)
proof
  have c ⊓ d ⊑ ((c // i) || i) ⊓ ((d // i) || i)
  by (metis nonabort-i inf-mono rely-quotient)
  then have c ⊓ d ⊑ ((c // i) ⊓ (d // i)) || i by (metis inf-par-distrib)
  thus ?thesis by (metis nonabort-i rely-refinement)

qed
lemma distribute-rely-parallel1:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  shows \((c \parallel d) \parallel (i \parallel j) \subseteq (c \parallel i) \parallel (d \parallel j)\)
proof -
  have \((c \parallel d) \subseteq ((c \parallel i) \parallel i) \parallel ((d \parallel j) \parallel j)\)
      using par-mono rely-quotient nonabort-i nonabort-j by simp
then have \((c \parallel d) \subseteq (c \parallel i) \parallel (d \parallel j) \parallel i\) by (metis par-associ par-commute)
thus ?thesis using par-associ par-commute rely-refinement
by (metis nonabort-i nonabort-j par-nonabort)
qed

lemma distribute-rely-parallel2:
  assumes nonabort-i: chaos ⊑ i
  assumes i-par-i: i \parallel i \subseteq i
  shows \((c \parallel d) \parallel i \subseteq (c \parallel i) \parallel (d \parallel i)\)
proof -
  have \((c \parallel d) \parallel i \subseteq ((c \parallel i) \parallel (d \parallel i))\) using assms(1) using weaken-rely
  by (simp add: i-par-i par-nonabort)
then have \((c \parallel d) \parallel i \subseteq (c \parallel i) \parallel (d \parallel i)\) by (metis refine-trans)
thus ?thesis by (metis distribute-rely-parallel1 refine-trans nonabort-i)
qed

lemma distribute-rely-sequential:
  assumes nonabort-i: chaos ⊑ i
  assumes \(\forall c. (\forall d. ((c \parallel i) ; (d \parallel j) \subseteq (c ; d) \parallel (i ; j)))\)
  shows \((c ; d) \parallel i \subseteq (c \parallel i) ; (d \parallel i)\)
proof -
  have \((c ; d) \subseteq ((c \parallel i) \parallel i) ; ((d \parallel i) \parallel i)\)
      by (metis rely-quotient nonabort-i seq-mono)
then have \((c ; d) \subseteq (c \parallel i) ; (d \parallel i)\) by (metis refine-trans)
thus ?thesis by (metis rely-refinement nonabort-i)
qed

lemma distribute-rely-sequential-event:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  assumes nonabort-e: chaos ⊑ e
  assumes \(\forall c. (\forall d. ((c \parallel i) ; e ; (d \parallel j) \subseteq (c ; e ; d) \parallel (i ; e ; j)))\)
  shows \((c ; e ; d) \parallel (i ; e ; j) \subseteq (c \parallel i) ; e ; (d \parallel j)\)
proof -
  have \((c ; e ; d) \subseteq ((c \parallel i) \parallel i) ; e ; ((d \parallel j) \parallel j)\)
by (metis order.refl rely-quotient nonabort-i nonabort-j seq-mono)
then have c:e:d ⊑ ((c // i):e:(d // j)) || (i:e:j) using assms
by (metis refine-trans)
thus ?thesis using rely-refinement nonabort-i nonabort-j nonabort-e
by (simp add: Inf-lower local.rely-quotient-def)
qed

lemma introduce-parallel-with-rely:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j0: chaos ⊑ j0
  assumes nonabort-j1: chaos ⊑ j1
  shows (c ∩ d) // j0 ⊑ (c ∩ (j0 // (j0 || i))) || (j0 ∩ (d // (j1 || i)))
proof —
  have (c ∩ d) // i ⊑ (c // i) ∩ (d // i)
  by (metis distribute-rely-conjunction nonabort-i)
  then have (c ∩ d) // i ⊑ (j1 ∩ (c // (j0 || i))) || (j0 ∩ (d // (j1 || i)))
  by (metis introduce-parallel nonabort-j0 nonabort-j1 inf-assoc inf.absorb-iff)
thus ?thesis by (simp add: nested-rely nonabort-i)
qed

lemma introduce-parallel-with-rely-guarantee:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j0: chaos ⊑ j0
  assumes nonabort-j1: chaos ⊑ j1
  shows (j1 || j0) ∩ (c ∩ d) // i ⊑ (j1 ∩ (c // (j0 || i))) || (j0 ∩ (d // (j1 || i)))
proof —
  have (j1 || j0) ∩ (c ∩ d) // i ⊑ (j1 || j0) ∩ ((j1 ∩ (c // (j0 || i))) || (j0 ∩ (d // (j1 || i))))
  by (metis introduce-parallel-with-rely nonabort-j0 nonabort-j1 conj-mono order.refl)
  also have ... ⊑ c // (j0 || i)) || (j0 ∩ (d // (j1 || i)))
  by (metis conj-assoc parallel-interchange)
finally show ?thesis by (metis conj-idem)
qed

lemma wrap-rely-guar:
  assumes nonabort-rg: chaos ⊑ rg
  and skippable: rg ⊑ skip
  shows c ⊑ rg ∩ c // rg
proof —
  have c = c // skip by (simp add: quotient-identity)
  also have ... ⊑ c // rg by (simp add: skippable weaken-rely nonabort-rg)
  also have ... ⊑ rg ∩ c // rg using conjoin-non-aborting conj-commute nonabort-rg
  by auto
locale rely-distrib-iteration = rely-distrib + iteration-finite-conjunctive

lemma distribute-rely-iteration:
  assumes nonabort-i: chaos ⊑ i
  assumes (∀ c. (∀ d. ((c ∥ i);(d ∥ i)) ⊑ (c;d) ∥ i)))
  shows (cω,d) // i ⊑ (c // i)ω;(d // i)
proof –
  have d △ c ; ((c // i)ω;(d // i) ∥ i) ⊑ ((d // i) ∥ i) △ ((c // i) ∥ i);((c // i)ω;(d // i) ∥ i)
  by (metis inf-mono order.refl rely-quotient nonabort-i seq-mono)
also have ... ⊑ ((d // i) ∥ i) △ ((c // i);(c // i)ω;(d // i) ∥ i)
  using assms inf-mono-right seq-assoc by fastforce
also have ... = ((c // i)ω;(d // i)) ∥ i
  by (simp add: inf-par-distrib)
also have ... = ((c // i)ω;(d // i)) ∥ i
  by (metis iter-unfold inf-seq-distrib seq-nil-left)
finally show thesis by (metis rely-refinement nonabort-i iter-induct)
qed

end

13 Conclusions

The theories presented here provide a quite abstract view of the rely/guarantee
approach to concurrent program refinement. A trace semantics for this theory has
been developed [2]. The concurrent refinement algebra is general enough to also
form the basis of a more concrete rely/guarantee approach based on a theory of
atomic steps and synchronous parallel and weak conjunction operators [4].

Acknowledgements. This research was supported by Australian Research Coun-
cil Grant grant DP130102901 and EPSRC (UK) Taming Concurrency grant. This
research has benefited from feedback from Robert Colvin, Chelsea Edmonds, Ned Hoy, Cliff Jones, Larissa Meinicke, and Kirsten Winter.

A Differences to earlier paper

This appendix summarises the differences between these Isabelle theories and the earlier paper [3]. We list the changes to the axioms but not all the flow on effects to lemmas.

1. The earlier paper assumes \( c ; (d_0 \sqcap d_1) = (c ; d_0) \sqcap (c ; d_1) \) but here we separate the case where this is only a refinement from left to right (Section 3) from the equality case (Section 9).

2. The earlier paper assumes \( (\bigsqcup C) \| d = (\bigsqcup c \in C. c \| d) \) but in Section 4 we assume this only for non-empty \( C \) and furthermore assume that parallel is abort strict, i.e. \( \bot \| c = c \).

3. The earlier paper assumes \( c \sqcap (\bigsqcup D) = (\bigsqcup d \in D. c \sqcap d) \). In Section 5 that assumption is not made because it does not hold for the model we have in mind [2] but we do assume \( c \sqcap \bot = \bot \).

4. In Section 6 we add the assumption \( \text{nil} \sqsubseteq \text{nil} \| \text{nil} \) to locale sequential-parallel.

5. In Section 6 we add the assumption \( \top \sqsubseteq \text{chaos} \| \top \).

6. In Section 6 we assume only \( \text{chaos} \sqsubseteq \text{chaos} \| \text{chaos} \) whereas in the paper this is an equality (the reverse direction is straightforward to prove).

7. In Section 6 axiom \( \text{chaos-skip} (\text{chaos} \sqsubseteq \text{skip}) \) has been dropped because it can be proven as a lemma using the parallel-interchange axiom.

8. In Section 6 we add the assumption \( \text{chaos} \sqsubseteq \text{chaos} ; \text{chaos} \).

9. Section 9 assumes \( D \neq \{\} \Rightarrow c ; \bigsqcap D = (\bigsqcap d \in D. c ; d) \). This distribution axiom is not considered in the earlier paper.

10. Because here parallel does not distribute over an empty non-deterministic choice (see point 2 above) in Section 12 the theorem rely-quotient needs to assume the interference process \( i \) is non-aborting (refines chaos). This also affects many lemmas in this section that depend on theorem rely-quotient.
References


