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.
A Differences to earlier paper

43
1 Overview

The theories provided here were developed in order to provide support for rely/guarantee 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
  HOL-Library.Lattice-Syntax
begin

The underlying lattice of commands is complete and distributive. We follow the refinement calculus tradition so that $\sqcap$ is non-deterministic choice and $c \sqsubseteq d$ 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
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 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 add.left-neutral order-class.eq-iff)
  then have (∏ k. ∃ i. (∏ j. f (i + j)) ⊑ f k)
    by (meson UNIV-I complete-lattice-class.INF-lower2)
then show \((\prod i j. f (i + j)) \subseteq (\prod k. f k)\)
by (simp add: complete-lattice-class.INF-mono)
qed

lemma nested-Collect: \(\{f y \mid y \in \{g x \mid x \in X\}\} = \{f (g x) \mid x \in X\}\)
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 ⇒ 'a
assumes qual: \(P \{d x \mid x \in X\}\)
assumes f-Inf-distrib: \(\forall c D. P D \Longrightarrow f c (\prod D) = \prod \{f c d \mid d . d \in D\}\)
shows f c (\(\prod x \in X. d x\)) = (\(\prod x \in X. f c (d x)\))
proof –
have f c (\(\prod x \in X. d x\)) = f c (\(\prod \{d x \mid x \in X\}\)) by (simp add: INF-Inf)
also have \(\ldots = (\prod \{f c dx \mid dx. dx \in \{d x \mid x \in X\}\})\) by (simp add: qual f-Inf-distrib)
also have \(\ldots = (\prod \{f c (d x) \mid x. x \in X\})\) by (simp only: nested-Collect)
also have \(\ldots = (\prod x \in X. f c (d x))\) by (simp add: INF-Inf)
finally show \(?thesis\).
qed

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

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 ⇒ 'a) ⇒ bool
where
dist-over-sup F ≡ (∀ X. F (\(\bigsqcup X\)) = (\(\bigsqcup x \in X. F (x)\)))
abbreviation
dist-over-inf :: ('a::refinement-lattice ⇒ 'a) ⇒ bool
where
dist-over-inf F ≡ (∀ X . F (∏ X) = (∏ x∈X. F (x)))
end

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 ⊥ as a left annihilator.

locale seq =
  fixes seq :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl ; 90)
  assumes seq-bot [simp]: ⊥ ; c = ⊥

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

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: 
    \((c::'a::refinement-lattice);(d_0 \sqcap d_1) \sqsubseteq (c;d_0 \sqcap c;d_1)\)
begin

Left distribution implies sequential composition is monotonic is its right argument

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

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

end

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

lemma INF-seq-distrib: \((\bigsqcap c\in C.f c); d = (\bigsqcap c\in C.f c ; d)\)
using Inf-seq-distrib by auto

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

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

lemma seq-top [simp]: \(\top ; c = \top\)
proof
  have \(\top ; c = (\bigsqcap a \in \{\}. a ; c)\)
  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 ((^) [1000] 1000)

notation (HTML output)
seq-power ((^) [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 - i))
using Suc.prems Suc-diff-le seq-assoc by force
finally show ?case .
qed

end

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

lemma seq-mono: c1 ⊑ d1 ⇒ c2 ⊑ d2 ⇒ c1;c2 ⊑ d1;d2
using seq-mono-left seq-mono-right by (metis inf .orderE le-infI2)

end

end
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 ≠ {} ⟹ (∏ D) ∥ c = (∏ d∈D. d ∥ c)
  using par-Inf-distrib par-commute by simp
lemma par-INF-distrib: $X \neq \{\} \implies c \parallel (\bigcap x \in X. d x) = (\bigcap x \in X. c \parallel d x)$
using par-Inf-distrib by auto

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

lemma INF-INF-par-distrib:
$X \neq \{\} \implies Y \neq \{\} \implies (\bigcap x \in X. c x) \parallel (\bigcap y \in Y. d y) = (\bigcap x \in X. \bigcap y \in Y. c x \parallel d y)$
proof –
assume nonempty-X: $X \neq \{\}$
assume nonempty-Y: $Y \neq \{\}$
have $(\bigcap x \in X. c x) \parallel (\bigcap y \in Y. d y) = (\bigcap x \in X. \bigcap y \in Y. c x \parallel d y)$
using INF-par-distrib by (metis nonempty-X)
also have ... = $(\bigcap x \in X. \bigcap y \in Y. c x \parallel d y)$ using par-INF-distrib by (metis nonempty-Y)
thus ?thesis by (simp add: calculation)
qed

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

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

lemma inf-par-product: $(a \cap b) \parallel (c \cap d) = (a \parallel c) \cap (a \parallel d) \cap (b \parallel c) \cap (b \parallel d)$
by (simp add: inf-commute inf-par-distrib inf-par-distrib2 inf-sup-aci(3))

lemma par-mono: $c_1 \subseteq d_1 \implies c_2 \subseteq d_2 \implies c_1 \parallel c_2 \subseteq d_1 \parallel d_2$
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
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 \Rightarrow 'a \Rightarrow 'a  (infixl \( \sqcap \))
  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: \( a \sqcap b \neq \bot \implies a \neq \bot \land b \neq \bot \)
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: $D \neq \emptyset \implies (\bigcap D) \cap c = (\bigcap_{d \in D} d \cap c)$

begin

lemma conj-Inf-distrib: $D \neq \emptyset \implies c \cap (\bigcap D) = (\bigcap_{d \in D} c \cap d)$
using Inf-conj-distrib conj-commute by auto

lemma inf-conj-distrib: $(c_0 \cap c_1) \cap d = (c_0 \cap d) \cap (c_1 \cap d)$
proof
  have $(c_0 \cap c_1) \cap d = (\bigcap \{c_0, c_1\}) \cap d$ by simp
  also have \ldots \by \(rule \ Inf-conj-distrib, \ simp\)
  also have \ldots \by simp
finally show \thesis .
qed

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

lemma conj-mono: $c_0 \subseteq d_0 \implies c_1 \subseteq d_1 \implies c_0 \cap c_1 \subseteq d_0 \cap d_1$
by (metis inf-conj-product inf-right-idem)

lemma conj-mono-left: $c_0 \subseteq c_1 \implies c_0 \cap d \subseteq c_1 \cap d$
by (simp add: conj-mono)

lemma conj-mono-right: $c_0 \subseteq c_1 \implies d \cap c_0 \subseteq d \cap c_1$
by (simp add: conj-mono)

lemma conj-refine: $c_0 \subseteq d \implies c_1 \subseteq d \implies c_0 \cap c_1 \subseteq d$
by (metis conj-idem conj-mono)

lemma refine-to-conj: $c \subseteq d_0 \implies c \subseteq d_1 \implies c \subseteq d_0 \cap d_1$
by (metis conj-idem conj-mono)

lemma conjoin-non-aborting: chaos $\subseteq c \implies d \subseteq d \cap c$
by (metis conj-mono order refl conj-chaos)

lemma conjunction-sup: $c \cap d \subseteq c \sqcup d$
by (simp add: conj-refine)

lemma conjunction-sup-nonabirting:
assumes chaos $\subseteq c$ and chaos $\subseteq d$
shows $c \cap d = c \cup d$
proof (rule antisym)
show \( c \sqcup d \sqsubseteq c \sqcap d \) using \( \text{assms(1)} \) \( \text{assms(2)} \) \text{conjoin-non-aborting local.conj-commute}
by fastforce
next
show \( c \sqcap d \sqsubseteq c \sqcup d \) by (metis conjunction-sup)
qed

lemma conjoin-top: chaos \( \subseteq c \Longrightarrow c \sqcap \top = \top \)
by (simp add: conjunction-sup-nonaborting)
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 \( \parallel \) nil \( \sqsubseteq \) nil
  and skip-nil: skip \( \sqsubseteq \) nil
  and skip-skip: skip \( \sqsubseteq \) skip;skip
begin
lemma nil-absorb: nil \( \parallel \) 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

Locale conjunction-parallel brings together the weak conjunction and parallel operators and relates their identities. It also introduces the interchange axiom for
locale conjunction-parallel = conj-distrib + par-distrib + 
assumes chaos-par-top: T ⊆ chaos || T 
assumes chaos-par-chaos: chaos ⊆ chaos || chaos 
assumes parallel-interchange: (c0 || c1) ⊆ (d0 || d1) ⊆ (c0 ⊆ d0) || (c1 ⊆ d1)
begin

lemma chaos-skip: chaos ⊆ skip
proof
  have chaos = (chaos || skip) ∩ (skip || chaos) by simp
  then have ... ⊆ (chaos ∩ skip) || (skip ∩ chaos) using parallel-interchange by blast
  thus ?thesis by auto
qed

lemma chaos-par-chaos-eq: chaos = chaos || chaos
by (metis antisym chaos-par-chaos chaos-skip order-refl par-mono par-skip)

lemma nonabort-par-top: chaos ⊆ c =⇒ c || T = T
by (metis chaos-par-top par-mono top.extremum-uniqueI)

lemma skip-conj-top: skip ∩ T = T
by (simp add: chaos-skip conjoin-top)

lemma conj-distrib2: c ⊆ c || c =⇒ c ∩ (d0 || d1) ⊆ (c ∩ d0) || (c ∩ d1)
proof
  assume c ⊆ c || c
  then have c ∩ (d0 || d1) ⊆ (c || c) ∩ (d0 || d1) 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: chaos ⊆ chaos;chaos
assumes sequential-interchange: (c0;c1) ∩ (d0;d1) ⊆ (c0 ∩ d0);(c1 ∩ d1)
begin

lemma chaos-nil: chaos ⊆ nil
by (metis conj-chaos local.conj-commute seq-nil-left seq-nil-right sequential-interchange)
lemma chaos-seq-absorb: chaos = chaos;chaos
proof (rule antisym)
  show chaos ⊑ chaos;chaos by (simp add: chaos-seq-chaos)
next
  show chaos;chaos ⊑ chaos using chaos-nil
    using seq-mono-left seq-nil-left by fastforce
qed

lemma seq-bot-conj: c;⊥ ⊑ (c ∩ d);⊥
by (metis (no-types) conj-bot-left seq-nil-right sequential-interchange)

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

lemma conj-distrib3: c ⊑ c;c ⇒ c ∩ (d₀;d₁) ⊑ (c ∩ d₀);(c ∩ d₁)
proof —
  assume c ⊑ c;c
  then have c ∩ (d₀;d₁) ⊑ (c;c) ∩ (d₀;d₁) by (metis conj-mono order refl)
  thus ?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
imports Refinement-Lattice
begin

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 \mid x = a \lor x = b\} = \{F a, F b\}
by auto

locale lower-galois-connections
begin

definition l-adjoint :: \('a\::\text{refinement-lattice} \Rightarrow 'a\) ⇒ ('a ⇒ 'a) (\(\triangleright 201\, 200\))
where
\((F^\triangleright)\, x \equiv \bigsqcap \{y. x \sqsubseteq F y\}\)

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

lemma l-cancellation: dist-over-inf F \implies x \sqsubseteq (F \circ F^\triangleright) x
proof –
assume dist: dist-over-inf F

define Y where \(Y = \{F y \mid y. x \sqsubseteq F y\}\)
define X where \(X = \{x\}\)

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

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

lemma v-simple-fusion: mono \( G \implies \forall x. ((F \circ G) x \sqsubseteq (H \circ F) x) \implies (gfp G) \sqsubseteq gfp H \)
  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 monoH: mono \( H \)
  and distribF: dist-over-inf \( F \)
  and comp-geq: \( \forall x. ((H \circ F) x \sqsubseteq (F \circ G) x) \)
  shows \( gfp H \sqsubseteq F (gfp G) \)
proof
  have \( (gfp H) \sqsubseteq (F \circ F^\circ) (gfp H) \) using distribF l-cancellation by simp
  then have \( H (gfp H) \sqsubseteq H ((F \circ F^\circ) (gfp H)) \) by (simp add: monoD monoH)
  then have \( H (gfp H) \sqsubseteq F ((G \circ F^\circ) (gfp H)) \) using comp-geq by (metis comp-def refine-trans)
  then have \( (F^\circ) (H (gfp H)) \sqsubseteq (G \circ F^\circ) (gfp H) \) using distribF by (metis (mono-tags) l-galois-connection)
  then have \( (F^\circ) (gfp H) \sqsubseteq (gfp G) \) by (metis comp-apply gfp-unfold gfp-upperbound monoH)
  thus \( gfp H \sqsubseteq F (gfp G) \) using distribF by (metis (mono-tags) l-galois-connection)
  qed

theorem fusion-gfp-eq:

assumes monoH: mono H and monoG: mono G
and distF: dist-over-inf F
and fgh-comp: \( \land x. (F \circ G) x = (H \circ F) x \)
shows \( F \left( \text{gfp} \ G \right) = \text{gfp} \ H \)
proof (rule antisym)
  show \( F \left( \text{gfp} \ G \right) \sqsubseteq \left( \text{gfp} \ H \right) \) by (metis fgh-comp le-less v-simple-fusion monoG)
next
  have \( \land x. ((H \circ F) x \sqsubseteq (F \circ G) x) \) using fgh-comp by auto
  then show \( \text{gfp} \ H \sqsubseteq \left( \text{gfp} \ G \right) \) using monoH distF fusion-gfp-geq by blast
qed

7.3 Upper Galois connections

locale upper-galois-connections
begin

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

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

lemma u-cancellation: dist-over-sup F \implies (F \circ F\#) \ x \sqsubseteq x
proof –
  assume 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 $F (\bigsqcup \{ y. F y \sqsubseteq x \}) \sqsubseteq x$ using SUP-le-iff dist by fastforce
thus ?thesis by (metis comp-def u-adjoint-def)

lemma u-galois-connection: dist-over-sup $F = \Rightarrow (F x \sqsubseteq y) \iff (x \sqsubseteq (F#) y)$

proof
assume dist: dist-over-sup $F$ then have monoF: mono $F$ by (simp add: dist-sup-mono)
assume $x \sqsubseteq (F#) y$ then have $a: F x \sqsubseteq F ((F#) y)$ by (simp add: monoD monoF)

then have $F ((F#) y) \sqsubseteq y$ using dist-u-cancellation by simp
thus $F x \sqsubseteq y$ 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: mono $H = \Rightarrow \forall x. ((F \circ G) x \sqsubseteq (G \circ H) x) \Rightarrow \text{lfp } F \sqsubseteq G (\text{lfp } H)$
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 monoH: mono $H$
  and distribF: dist-over-sup $F$
  and comp-leq: $\forall x. ((F \circ G) x \sqsubseteq (H \circ F) x)$
  shows $F (\text{lfp } G) \sqsubseteq (\text{lfp } H)$

proof —
  have $((F \circ F#) (\text{lfp } H)) \sqsubseteq \text{lfp } H$ using distribF u-cancellation by simp
  then have $H ((F \circ F#) (\text{lfp } H)) \sqsubseteq H (\text{lfp } H)$ by (simp add: monoD monoH)
  then have $F ((G \circ F#) (\text{lfp } H)) \sqsubseteq H (\text{lfp } H)$ using comp-leq by (metis comp-def refine-trans)
  then have $(G \circ F#) (\text{lfp } H) \subseteq (F#) (H (\text{lfp } H))$ using distribF by (metis (mono-tags)
  u-galois-connection)
  then have $(\text{lfp } G) \subseteq (F#) (\text{lfp } H)$ by (metis comp-def def-lfp-unfold lfp-lowerbound monoH)
thus $F (\text{lfp } G) \subseteq (\text{lfp } H)$ using distribF by (metis (mono-tags) u-galois-connection)

qed

\textbf{theorem} fusion-lfp-eq:
\textbf{assumes} monoH: $\text{mono } H$ and monoG: $\text{mono } G$
\textbf{and} distF: $\text{dist-over-sup } F$
\textbf{and} fgh-comp: $\bigwedge x. ((F \circ G) \cdot x = (H \circ F) \cdot x)$
\textbf{shows} $F (\text{lfp } G) = (\text{lfp } H)$
\textbf{proof} (rule antisym)
\textbf{show} $\text{lfp } H \subseteq F (\text{lfp } G)$ by (metis monoG fgh-comp eq-iff upper-galois-connections.u-simple-fusion)
\textbf{next}
\textbf{have} $\bigwedge x. (F \circ G) \cdot x \subseteq (H \circ F) \cdot x$ using fgh-comp by auto
\textbf{then show} $F (\text{lfp } G) \subseteq (\text{lfp } H)$ using monoH distF fusion-lfp-leq by blast

qed

end

\section{Iteration}

\textbf{theory} Iteration
\textbf{imports}
\quad \text{Galois-Connections}
\quad \text{CRA}
\textbf{begin}

\subsection{Possibly infinite iteration}

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

\textbf{locale} finite-or-infinite-iteration = seq-distrib + upper-galois-connections
\textbf{begin}

\textbf{definition}
\textit{iter} :: $'a \Rightarrow 'a (\omega \cdot [103] 102)$
\textbf{where}
$c^\omega \equiv \text{lfp} (\lambda x. \text{nil} \sqcap c;x)$

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

end
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: $\text{nil} \sqcap c; x \sqsubseteq x \implies c^\omega \sqsubseteq x$

**By** (simp add: iter-def lfp-lowerbound)

**Lemma** iter0: $c^\omega \sqsubseteq \text{nil}$

**By** (metis iter-unfold sup.orderI sup-inf-absorb)

**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** $c^\omega ; c^\omega \sqsubseteq c^\omega$ by (simp add: 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-left seq-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: $c \sqsubseteq d \implies c^\omega \sqsubseteq d^\omega$

**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)

end
### 8.2 Finite iteration

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

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

**begin**

**definition**

\[ \text{fiter} :: 'a \Rightarrow 'a \ (\cdot \star [101] 100) \]

**where**

\[ c^\ast \equiv \text{gfp} \ (\lambda x. \text{nil} \sqcap c;x) \]

**lemma** `fin-iter-step-mono`: `mono (\lambda x. \text{nil} \sqcap 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^\ast = \text{nil} \sqcap c;c^\ast`

**using** `fiter-def gfp-unfold fin-iter-step-mono by auto`

**lemma** `fiter-induct-nil`: `x \sqsubseteq \text{nil} \sqcap c;x \implies x \sqsubseteq c^\ast`

**by** `(simp add: fiter-def gfp-upperbound)`

**lemma** `fiter0`: `c^\ast \sqsubseteq \text{nil}`

**by** `(metis fiter-unfold inf . coboundedI)`

**lemma** `fiter1`: `c^\ast \sqsubseteq c`

**by** `(metis fiter0 fiter-unfold inf-le2 order . trans seq-mono-right seq-nil-right)`

**lemma** `fiter-induct-eq`: `c^\ast; d = \text{gfp} \ (\lambda x. c;x \sqcap d)`

**proof**

**define** `F` **where**

\[ F = (\lambda x. x;d) \]

**define** `G` **where**

\[ G = (\lambda x. \text{nil} \sqcap c;x) \]

**define** `H` **where**

\[ H = (\lambda x. c;x \sqcap d) \]

**have** `FG`: `F \circ G = (\lambda x. c;x;d \sqcap d)` **by** `(simp add: F-def G-def comp-def inf-commute inf-seq-distrib)`

**have** `HF`: `H \circ F = (\lambda x. c;x;d \sqcap 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 mon0I 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 mono \( G \) mono \( H \) fusion-gfp-eq by blast
then have \((gfp (\lambda x. \text{nil} \sqcap c; x)); d = gfp (\lambda x. c; x \sqcap d)\) using \( F\text{-def} \ G\text{-def} \ H\text{-def} \)
inf-commute by simp
thus \( ?\text{thesis by (metis fiter-def)} \)
qed

**Theorem fiter-induct:** \( x \sqsubseteq d \sqcap c; x \implies x \sqsubseteq c^* \sqcap d \)
proof
\begin{itemize}
\item assume \( x \sqsubseteq d \sqcap c; x \)
\item then have \( x \sqsubseteq c; x \sqcap d \) using inf-commute by simp
\item thus \( ?\text{thesis by (simp (full-types) fiter-induct-eq)} \)
\end{itemize}
qed

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

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

**Lemma fiter-mono:** \( c \sqsubseteq d \implies c^* \sqsubseteq d^* \)
proof
\begin{itemize}
\item assume \( c \sqsubseteq d \)
\item then have \( c^* \sqsubseteq \text{nil} \sqcap d; c^* \) by (metis fiter0 fiter1 fiter2 inf.bounded_iff refine-trans seq-mono-left)
\item thus \( ?\text{thesis by (metis seq-nil-right fiter-induct)} \)
\end{itemize}
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 (-∞ [105] 106)
where
\( c^\infty \equiv \text{lfp} (\lambda x. c;x) \)

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

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:** \( c;x \sqsubseteq x \implies c^\infty \sqsubseteq x \)

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

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

proof (induct \( i \))

\begin{itemize}
  \item case 0
    thus \(?case\) by simp
  \item case (Suc \( i \))
    thus \(?case\) using infiter-unfold seq-assoc seq-power-Suc by auto
\end{itemize}

qed

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

proof 

\begin{itemize}
  \item have \( \forall a. (\bot ::'a) \sqsubseteq a \)
    by auto
  \item 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)
\end{itemize}

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:** \( c^\omega \sqsubseteq c^* \)

by (metis seq-nil-right order.refl iter-unfold fiter-induct)

end
lemma iter-absorption [simp]: \((c^\omega)^* = c^\omega\)
proof (rule antisym)
  show \((c^\omega)^* \subseteq c^\omega\) by (metis fiter1)
next
  show \(c^\omega \subseteq (c^\omega)^*\) 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: \(c^\infty \subseteq c^\omega ; \top\)
  proof
    have \(c : (c^\omega ; \top) = \text{nil} ; \top \sqcap c ; c^\omega ; \top\)
      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 infiter-induct
           seq-distrib-right.iter-seq-distrib seq-distrib-right-axioms)
  qed
  have rl: \(c^\omega ; \top \subseteq c^\infty\)
    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 \(c^\infty \subseteq c^* ; \top\)
  by (metis eq-iff fiter-induct inf-top-left infiter-unfold)

lemma inf-ref-infiter: \(c^\omega \subseteq c^\infty\)
  using infiter-unfold iter-induct-nil by auto

end
end

9 Sequential composition for conjunctive models

theory Conjunctive-Sequential
imports Sequential
begin
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: \( c;(d_0 \cap d_1) = c;d_0 \cap c;d_1 \)
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 =
assumes seq-inf-distrib:
\( D \neq \emptyset \Rightarrow c;\bigcap D = (\bigcap d\in D. c ; d) \)
begin

sublocale seq-distrib
proof unfold-locales
fix \( c \):\('a and \( d_0, d_1: \ 'a\)
have \{d_0, d_1\} \neq \{\} by simp
then have \( c;\bigcap \{d_0, d_1\} = \bigcap \{c : d. d \in \{d_0, d_1\}\} \) using seq-Inf-distrib
proof
  have INFIMUM \{d_0, d_1\} (op ; c) = \bigcap \{c : a | a. a \in \{d_0, d_1\}\}
  using INF-Inf by blast
  then show ?thesis using \( (\forall c: \ 'a::refinement-lattice. D::\ 'a::refinement-lattice set. D \neq \{\} \Rightarrow c;\bigcap D = (\bigcap d::\ 'a::refinement-lattice\in D. c ; d) = \bigcap \{d_0, d_1\} :: \ 'a::refinement-lattice) \) \neq \{\} \) by presburger
qed
also have ... = \( c_0 \cap c ; d_1 \) by (simp only: Inf2-inf)
finally show \( c; (d_0 \cap d_1) \subseteq c ; d_0 \cap c ; d_1 \) by simp
qed

lemma seq-INF-distrib: \( X \neq \{\} \Rightarrow c; (\bigcap X. d x) = (\bigcap x\in X. c ; d x) \)
proof
  assume xne: \( X \neq \{\} \)
  have \( a: c; (\bigcap x\in X. d x) = c;\bigcap (d \cdot X) \) by auto
  also have \( b: \ldots = (\bigcap d\in (d \cdot X). c ; d) \) by (meson image-is-empty seq-Inf-distrib xne)
  also have \( c: \ldots = (\bigcap x\in X. c ; d x) \) by simp
  finally show ?thesis by (simp add: b)
qed

lemma seq-INF-distrib-UNIV: \( c ; (\bigcap x. d x) = (\bigcap x. c ; d x) \)
by (simp add: seq-INF-distrib)

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

lemma INF-INF-seq-distrib-UNIV: \( (\prod_{x \cdot c \cdot x}) \cdot (\prod_{y \cdot d \cdot y}) = (\prod_{x, y \cdot c \cdot x \cdot d \cdot y}) \)
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 \( (\prod_{j \cdot f \cdot i \cdot j}) = (\prod_{j \cdot f \cdot i \cdot j \cdot i = j \cdot f \cdot i \cdot j}) \cap (\prod_{j \cdot f \cdot i \cdot j \cdot i < j \cdot f \cdot i \cdot j}) \cap (\prod_{j \cdot f \cdot i \cdot j \cdot j < i \cdot f \cdot i \cdot j}) \)
proof
have univ-part: \( UNIV = \{j. i = j\} \cup \{j. i < j\} \cup \{j. j < i\} \)
by auto

have \( (\prod_{j \in \{j. i = j\} \cup \{j. i < j\} \cup \{j. j < i\}} f \cdot i \cdot j) = \\
(\prod_{j \in \{j. i = j\}} f \cdot i \cdot j) \cap \\
(\prod_{j \in \{j. i < j\}} f \cdot i \cdot j) \cap \\
(\prod_{j \in \{j. j < i\}} f \cdot i \cdot j) \)
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
shows \( (\prod_{i \cdot \prod_{j \cdot f \cdot i \cdot j}} = (\prod_{i \cdot \prod_{j \in \{j. i = j\}} f \cdot i \cdot j} \cap (\prod_{i \cdot \prod_{j \in \{j. i < j\}} f \cdot i \cdot j} \cap


28
\begin{enumerate}
\item \(\bigcap i. \bigcap j | j < i \cdot f i j\)
\item \textbf{proof} \hspace{1cm} \begin{enumerate}
\item \textbf{have} \(\bigcap i. \bigcap j (f i j) = (\bigcap i. (\bigcap j | j = i \cdot f i j) \cap \bigcap j | j < i \cdot f i j) \cap \bigcap j | j < i \cdot f i j)\)
\item \textbf{by} (simp add: INF-partition-nat3)
\item \textbf{also have} \(\ldots = (\bigcap i. \bigcap j | j = i \cdot f i j) \cap \bigcap i. \bigcap j | j < i \cdot f i j) \cap \bigcap i. \bigcap j | j < i \cdot f i j)\)
\item \textbf{by} (simp add: INF-inf-distrib)
\item \textbf{finally show} \(?thesis .
\item \textbf{qed}
\end{enumerate}
\end{enumerate}

\textbf{lemma} \(\text{INF-nat-shift}: (\bigcap i | i < i \cdot f i) = (\bigcap i | f (\text{Suc} i))\)
\item \textbf{by} (metis greaterThan-0 greaterThan-def range-composition)

\textbf{lemma} \(\text{INF-nat-minus}:
\item \textbf{fixes} \(f :: \text{nat} \Rightarrow 'a::\text{refinement-lattice}
\item \textbf{shows} \((\bigcap j | j < i \cdot f (j - i)) = (\bigcap k | k < k \cdot f k)\)
\item \textbf{apply} (rule antisym)
\item \textbf{apply} (rule INF-mono, simp)
\item \textbf{apply} (metis add.right-neutral add-diff-cancel-left' add-less-cancel-left order-refl)
\item \textbf{apply} (rule INF-mono, simp)
\item \textbf{by} (meson order-refl zero-less-diff)

\textbf{lemma} \(\text{INF-INF-guarded-switch}:
\item \textbf{fixes} \(f :: \text{nat} \Rightarrow \text{nat} \Rightarrow 'a::\text{refinement-lattice}
\item \textbf{shows} \((\bigcap i. \bigcap j | j < i \cdot f j (i - j)) = (\bigcap j | i \in \{j. j < i\} \cdot f j (i - j))\)
\item \textbf{proof} (rule antisym)
\item \textbf{have} \(\bigwedge j. i j < ii \implies \exists i. \exists j<i. f j (i - j) \subseteq f j j (ii - jj)\)
\item \textbf{by} blast
\item \textbf{then have} \(\bigwedge j. i j < ii \implies \exists i. \bigcap j | j < i \cdot f j (i - j) \subseteq f j j (ii - jj)\)
\item \textbf{by} (meson INF-lower mem-Collect-eq)
\item \textbf{then have} \(\bigwedge j. i j < ii \implies \bigcap i. \bigcap j | j < i \cdot f j (i - j) \subseteq f j j (ii - jj)\)
\item \textbf{by} (meson UNIV-I INF-lower dual-order-trans)
\item \textbf{then have} \(\bigwedge j j. \bigcap i. \bigcap j | j < i \cdot f j (i - j) \subseteq (\bigcap j j. \bigcap ii | ii \in \{ii. jj < ii\} \cdot f jj (ii - jj))\)
\item \textbf{by} (meson (mono-tags, lifting) INF-greatest mem-Collect-eq)
\item \textbf{then have} \(\bigwedge j j. \bigcap i. \bigcap j | j < i \cdot f j (i - j) \subseteq (\bigcap j j. \bigcap ii | ii \in \{ii. jj < ii\} \cdot f jj (ii - jj))\)
\item \textbf{by} (simp add: INF-greatest)
\item \textbf{then show} \(\bigcap i. \bigcap j | j < i \cdot f j (i - j) \subseteq (\bigcap j j. \bigcap i | i < i \cdot f j (i - j))\)
\item \textbf{by} simp
\end{enumerate}
have \( \land ii \land jj < ii \implies \exists j. \exists i > j. f j (i - j) \subseteq f jj (ii - jj) \)
by blast
then have \( \land ii \land jj < ii \implies \exists j. \{ i. j < i \}. f j (i - j) \subseteq f jj (ii - jj) \)
by (meson INF-lower mem-Collect-eq)
then have \( \land ii \land jj < ii \implies (\bigcap j. \bigcap i \in \{ i. j < i \}. f j (i - j)) \subseteq f jj (ii - jj) \)
by (meson UNIV-I INF-lower dual-order.trans)
then have \( (\bigcap j. \bigcap i \in \{ i. j < i \}. f j (i - j)) \subseteq (\bigcap i. \bigcap j \in \{ j. jj < ii \}. f jj (ii - jj)) \)
by (simp add: INF-greatest)
then show \( (\bigcap j. \bigcap i \in \{ i. j < i \}. f j (i - j)) \subseteq (\bigcap i. \bigcap j \in \{ j. j < i \}. f j (i - j)) \)
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 \( H = (\lambda x. \text{nil} \cap c;x \cap x) \)

have \( FG: F \circ G = (\lambda x. c^* \cap c;x) \) using F-def G-def by auto
have \( HF: H \circ F = (\lambda x. \text{nil} \cap c;(c^* \cap x)) \) 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 \( F \ (lfp \ G) = lfp \ H \) using adjoint mono\ H monoG fusion-lfp-eq by blast
then have \( c^* \sqcap lfp (\lambda x. c;x) = lfp (\lambda x. \text{nil} \sqcap c;x) \)
  using F-def G-def H-def by blast
thus \( ?\text{thesis} \) by (simp add: infiter-def iter-def)
qed

lemma iter-induct-isolate: \( c^*:d \sqcap c^\omega = lfp (\lambda x. d \sqcap c;x) \)
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: \( c^*:d = (\text{nil} \sqcap c;c^*)\cdot d \) 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 \( F \ (lfp \ G) = lfp \ H \) using adjoint mono\ H monoG fusion-lfp-eq by blast
then have \( c^* \sqcap lfp (\lambda x. c;x) = lfp (\lambda x. d \sqcap c;x) \)
  using F-def G-def H-def by blast
thus \( ?\text{thesis} \) by (simp add: infiter-def iter-def)
qed

lemma iter-induct-eq: \( c^\omega;d = lfp (\lambda x. d \sqcap c;x) \)
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 \( c^*:d \sqcap c^\omega = lfp (\lambda x. d \sqcap c;x) \) by (simp add: iter-induct-isolate)
  thus \( ?\text{thesis} \)
    by (simp add: \( c^\omega;d = c^*;d \sqcap c^\omega;d \))
qed
lemma iter-induct: $d \sqsubseteq c : x \subseteq x \implies c^\omega : d \sqsubseteq x$
by (simp add: iter-induct-eq lfp-lowerbound)

lemma iter-isolate: $c^\omega : d \sqcap c^{\infty} = c^{\omega} : d$
by (simp add: iter-induct-eq iter-induct-isolate)

lemma iter-isolate2: $c : c^{\omega} : d \sqcap c^{\infty} = c : c^{\omega} : d$
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
  seq-seq-distrib)

next
  have $(c ; (c \sqcap d)^{\omega} \sqcap d ; (c \sqcap d)^{\omega}) \sqcap \text{nil} \subseteq (c \sqcap d)^{\omega}$
by (metis inf-commute order.refl
  inf-seq-distrib iter-unfold)
  then have $a : c^{\omega} ; (d ; (c \sqcap d)^{\omega} \sqcap \text{nil}) \subseteq (c \sqcap d)^{\omega}$

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.semgroup-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 $d ; c^{\omega} ; (d ; (c \sqcap d)^{\omega} \sqcap \text{nil}) \sqcap \text{nil} \subseteq d ; (c \sqcap d)^{\omega} \sqcap \text{nil}$
by (metis inf-mono order.iter-refl
  seq-assoc seq-mono)
  then have $(d ; c^{\omega})^{\omega} \subseteq (d ; (c \sqcap d)^{\omega} \sqcap \text{nil})$
by (metis inf-commute iter-induct-nil)
  then have $c^{\omega} ; (d ; c^{\omega})^{\omega} \subseteq c^{\omega} ; (d ; (c \sqcap d)^{\omega} \sqcap \text{nil})$
by (metis order.refl seq-mono)
  thus $c^{\omega} ; (d ; c^{\omega})^{\omega} \subseteq (c \sqcap d)^{\omega}$
using a refine-trans by blast

qed

lemma iter-leapfrog-var: $(c ; d)^{\omega} ; c \subseteq c ; (d ; c)^{\omega}$

proof
  have $c \sqcap c ; d ; c^{\omega} \subseteq c ; (d ; c)^{\omega}$

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

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

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

lemma fiter - leapfrog: $c;(d;c)^* = (c;d)^*;c$
proof
  have lr: $c;(d;c)^* \subseteq (c;d)^*;c$
  proof -
  have $(d : c)^* = nil \cap d : c ; (d : c)^*$ by (meson finite - iteration . fiter - unfold finite - iteration - axioms)
  then show ?thesis by (metis fiter - induct seq - assoc seq - distrub - left . weak seq - inf - distrub seq - distrub - left - axioms seq - nil - right)
qed

have rl: $(c;d)^*;c \subseteq c;(d;c)^*$
proof -
  have a1: $(c;d)^*;c = c \cap c;d;(c;d)^*;c$
  by (metis finite - iteration . fiter - unfold finite - iteration - axioms inf - seq - distrub seq - nil - left)
  have a2: $(c;d)^*;c \subseteq c;(d;c)^* \iff c \cap c;d;(c;d)^*;c \subseteq c;(d;c)^*$ by (simp add : a1)
  then have a3: $(c;d)^*;c \subseteq c;(d;c)^*;c$
  by (metis a1 eq - iff fiter - unfold lr seq - assoc seq - inf - distrub seq - nil - right)
  have a4: $(nil \cap d;(c;d)^*;c) \subseteq (d;c)^* \Longrightarrow c;(nil \cap d;(c;d)^*;c) \subseteq c;(d;c)^*$
  using seq - mono - right by blast
  have a5: $(nil \cap d;(c;d)^*;c) \subseteq (d;c)^*$
  proof -
  have f1: $d : (c ; d)^* ; c \cap nil = d ; ((c ; d)^* ; c) \cap nil \cap nil$
  by (simp add : seq - assoc)
  have $d : c ; (d : c)^* ; c \cap nil = d ; ((c ; d)^* ; c)$ by (metis (no - types) a1 inf - sup - aci(1) seq - assoc seq - finite - conjunctive . seq - inf - distrub seq - finite - conjunctive - axioms seq - nil - right)
  then show ?thesis by (metis (no - types) finite - iteration . fiter - induct finite - iteration - axioms inf . coboundedI inf - sup - aci(1) seq - nil - right)
qed

thus ?thesis using a2 a3 a4 by blast
thus ?thesis by (simp add: eq-iff lr)

end

locale iteration-infinite-conjunctive = seq-infinite-conjunctive + iteration + infimum-nat

begin

lemma fiter-seq-choice: c* = (∏ i::nat. c i
proof (rule antisym)
  show c* ⊑ (∏ i. c i)
  proof (rule INF-greatest)
    fix i
    show c* ⊑ c i
    proof (induct i type: nat)
      case 0
      show c* ⊑ c 0 by (simp add: fiter0)
      next
      case (Suc n)
      have c* ⊑ c i by (metis fiter-unfold inf-le2)
      also have ... = c i Suc n by simp
      finally show c* ⊑ c i Suc n.
    qed
  qed
  next
  have (∏ i. c i) ⊑ (c 0) ∩ (∏ i. c Suc i)
  by (meson INF-greatest INF-lower UNIV-I le-inf-iff)
  also have ... = nil ∩ (∏ i. c i) by simp
  also have ... = nil ∩ c by (simp add: seq-INF-distrib)
  finally show (∏ i. c i) ⊑ c* using fiter-induct by fastforce
  qed

lemma fiter-seq-choice-nonempty: c : c* = (∏ i∈{i. 0 < i}. c i
proof —
  have (∏ i∈{i. 0 < i}. c i) = (∏ i. c Suc i) by (simp add: INF-nat-shift)
  also have ... = (∏ i. c Suc i) by simp
  also have ... = c by (simp add: seq-INF-distrib-UNIV)
  also have ... = c* by (simp add: fiter-seq-choice)
  finally show ?thesis by simp


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

begin

lemma conj-distrib4: \( c^* \otimes d^* \subseteq (c \otimes d)^* \)
proof
  have \( c^* \otimes d^* = (\text{nil} \otimes (c;c^*)) \otimes d^* \) by (metis fiter-unfold)
  then have \( c^* \otimes d^* = (\text{nil} \otimes d^*) \otimes ((c;c^*) \otimes d^*) \) by (simp add: inf-conj-distrib)
  then have \( c^* \otimes d^* \subseteq \text{nil} \otimes ((c;c^*) \otimes (d;d^*)) \) by (metis conj-idem fiter0 fiter-unfold inf.bounded-iff inf-le2 local.conj-mono)
  then have \( c^* \otimes d^* \subseteq \text{nil} \otimes ((c \otimes d);(c^* \otimes d^*)) \) 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 \( i \) representing interference from the environment.

theory Rely-Quotient
imports CRA
  Conjunction-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 :: 'a ⇒ 'a ⇒ 'a (infixl '/'/ 85)

where
  c // i ≡ \{ d. (c ⊑ d || i)\}

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

lemma quotient-identity: c // skip = c
proof −
  have c // skip = \{ d. (c ⊑ d || skip) \} by (metis rely-quotient-def)
  then have c // skip = \{ d. (c ⊑ 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 // ⊥) || ⊥ equals ⊥, it does not refine
C in general.

theorem rely-quotient:
  assumes nonabort-i: chaos ⊑ i
  shows c ⊑ (c // i) || i
proof −
  define D where D = \{ d || i \} d ⊑ d || i\}
  define C where C = \{ c \}
  have (\forall d ∈ D. (\exists c ∈ C. c ⊑ d)) using D-def C-def CollectD singletonI by auto
  then have \{ d || i \} d ⊑ d || i\} by simp add: Inf-mono
  also have ... = \{ d || i \} d ∈ \{ d. (c ⊑ d || i)\} by simp
  also have ... = (\{ d ∈ \{ d. (c ⊑ d || i)\}. d || i \} by (simp add: INF-Inf)
  also have ... = \{ d | d. (c ⊑ d || i)\} || i
proof (cases \{ d | d. (c ⊑ d || i)\} = {\})
  assume \{ d | d. (c ⊑ d || i)\} = {\}
  then show (\{ d ∈ \{ d. (c ⊑ d || i)\}. d || i \} = \{ d | d. (c ⊑ d || i)\} || i
  using nonabort-i Collect-empty-eq top-greatest
  nonabort-par-top par-commute by fastforce
next
  assume a: \{ d | d. (c ⊑ d || i)\} ≠ {\}
  have b: \{ d. (c ⊑ d || i)\} ≠ {\} using a by blast
  then have \{ d ∈ \{ d. (c ⊑ d || i)\}. d || i \} = \{ d. (c ⊑ d || i)\} || i
  using Inf-par-distrib by simp
then show \( \text{thesis by auto} \)
qed
also have \( \cdots = (c / \parallel i) \parallel i \) by (metis rely-quotient-def)
finally show \( \text{thesis} \).
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** \( c \parallel i \sube d \iff c \sqsubseteq d \parallel i \)

**proof**

assume \( a: c \parallel i \sube d \)

have \( i \sube (c \parallel i) \parallel i \) 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 \( \bigsqcup \{ d. (c \sqsubseteq d \parallel i) \} \sube d \) by (simp add: Inf-lower)

thus \( c \parallel i \sube d \) 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** \( (c \parallel i) \sube (d \parallel i) \)

**proof**

have \( \forall f. \ ((d \sqsubseteq f \parallel i) \implies \exists e. (c \sqsubseteq e \parallel i) \land (e \sqsubseteq f)) \)

using c-refsto-d order.trans by blast

then have \( \bigsqcup \{ e. (c \sqsubseteq e \parallel i) \} \subseteq \bigsqcup \{ f. (d \sqsubseteq f \parallel i) \} \)

by (metis Inf-mono mem-Collect-eq)

show \( \text{thesis} \) 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** \( (c \parallel j) \sube (c \parallel i) \)

**proof**

have \( \forall f. \ ((c \sqsubseteq f \parallel i) \implies \exists e. (c \sqsubseteq e \parallel j) \land (e \sqsubseteq f)) \)
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

lemma 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 \( \bigwedge f. ((c \subseteq f \parallel i \parallel j) \implies \exists e. (c \subseteq e \parallel j) \land (e \subseteq f \parallel i)) \) 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 \( \bigwedge d. (\bigcap \{ e. (c \subseteq e \parallel j) \} \subseteq d \parallel i \implies (c \subseteq d \parallel i \parallel j)) \)
    by (meson j-nonabort order-trans rely-refinement)
    thus ?thesis
    by (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

38
The following is a fundamental law for introducing a parallel composition of pro-
cess 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) ∥ (d // j) using a b by (metis conj-mono)
  also have interchange: c ∩ d ⊑ (c // i) ∩ (d // j) using parallel-interchange refine-trans calculation by blast
  show ?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) ∩ (d // i)) || (i ∩ i) by (metis inf-par-distrib)
  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 \((c || d) /\!\!\!/ (i || j) \subseteq (c /\!\!\!/ i) || (d /\!\!\!/ j)\)
proof
  have \((c || d) \subseteq ((c /\!\!\!/ i) || i) || ((d /\!\!\!/ j) || j)\)
  using par-mono rely-quotient nonabort-i nonabort-j by simp
  then have \((c || d) \subseteq ((c /\!\!\!/ i) \otimes (d /\!\!\!/ j)) || (i \otimes j)\)
  by (metis par-assoc par-commute)
thus \(?thesis\)
  by (metis par-assoc par-commute rely-refinement nonabort-i nonabort-j par-nonabort)
qed

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

lemma distribute-rely-sequential:
assumes nonabort-i: chaos ⊆ i
assumes \(\forall c. \forall d. ((c || d) \otimes (i || i) \subseteq (c || d) || (i || i))\)
shows \((c || e || d) /\!\!\!/ i \subseteq (c || e || i) \otimes (d || e || j)\)
proof
  have \(c || d \subseteq ((c || i) || i) \otimes (d || i || i)\)
  by (metis rely-quotient nonabort-i seq-mono)
  then have \(c || d \subseteq (c || i) || (d || i) \otimes (i || i)\)
  using assms(2) 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 \(\forall c. \forall d. ((c || i) || e || (d || j) \subseteq (c || e || d) || (i || e || j))\)
shows \((c || e || d) /\!\!\!/ (i || e || j) \subseteq (c || e || i) || (d || e || j)\)
proof
  have \(c || e || d \subseteq ((c || i) || i) \otimes (d || j) \otimes (i || e || j)\)
  by (metis order_refl rely-quotient nonabort-i nonabort-j seq-mono)
  then have \(c || e || d \subseteq (c || i) || (d || j) \otimes (i || e || j)\)
  using assms

40
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 \((c \cap d) /\ i \subseteq (j_1 \cap (c /\ (j_0 \parallel i))) \parallel (j_0 \cap (d /\ (j_1 \parallel i)))\)
proof
  have \((c \cap d) /\ i \subseteq (c /\ i) \cap (d /\ i)\)
    by (metis distribute-rely-conjunction nonabort-i)
  then have \((c \cap d) /\ i \subseteq (j_1 \cap (c /\ (j_0 \parallel i))) \parallel (j_0 \cap (d /\ (j_1 \parallel i)))\)
    by (metis introduce-parallel nonabort-j0 nonabort-j1 inf-assoc inf.absorb-iff1)
  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 \((j_1 \parallel j_0) \cap (c \cap d) /\ i \subseteq (j_1 \cap (c /\ (j_0 \parallel i))) \parallel (j_0 \cap (d /\ (j_1 \parallel i)))\)
proof
  have \((j_1 \parallel j_0) \cap (c \cap d) /\ i \subseteq (j_1 \parallel j_0) \cap (j_1 \cap (c /\ (j_0 \parallel i))) \parallel (j_0 \cap (d /\ (j_1 \parallel i)))\)
    by (metis introduce-parallel-with-rely nonabort-i nonabort-j0 nonabort-j1 conj-mono order.refl)
  also have ... \subseteq (j_1 \parallel j_0) \cap (c /\ (j_0 \parallel i)) \parallel (j_0 \cap (d /\ (j_1 \parallel 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 \(c \subseteq rg \cap c /\ rg\)
proof
  have \(c = c /\ skip\) by (simp add: quotient-identity)
  also have \(c \subseteq rg\) by (simp add: skippable weaken-rely nonabort-rg)
  also have \(rg \cap c /\ rg\) using conjoin-non-aborting conj-commute nonabort-rg
    by auto
  finally show \(c \subseteq rg \cap c /\ rg\)
qed
begin

locale rely-distrib-iteration = rely-distrib + iteration-finite-conjunctive

begin

lemma distribute-rely-iteration:
  assumes nonabort-i: chaos ⊑ i
  assumes (∀ c. (∀ d. ((c || i);(d || i) ⊑ (c;d) || i)))
  shows (cω;d) // i ⊑ (c /// iω;(d // i))
proof −
  have d △ c ; ((c /// iω;(d /// i) || i) ⊑ ((d /// i) || i) △ ((c /// i) || i):((c /// iω; (d /// i)) || i)
    by (metis inf-mono order.refl rely-quotient nonabort-i seq-mono)
  also have ... ⊑ ((d // i) || i) △ ((c /// i):(c /// iω;(d /// i)) || i)
    using assms inf-mono-right seq-associative by fastforce
  also have ... ⊑ ((d // i) △ (c // i):(c // iω;(d // i)) || i)
    by (simp add: inf-par-distrib)
  also have ... = ((c // iω;(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

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 \( (\bigcap C) \parallel d = (\bigcap c \in C. c \parallel d) \) but in Section 4 we assume this only for non-empty \( C \) and furthermore assume that parallel is abort strict, i.e. \( \bot \parallel c = c \).

3. The earlier paper assumes \( c \sqcap (\bigcup D) = (\bigcup d \in D. c \sqcap d) \). 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 \( nil \sqsubseteq nil \parallel nil \) to locale sequential-parallel.

5. In Section 6 we add the assumption \( T \sqsubseteq chaos \parallel T \).

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

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

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

9. Section 9 assumes \( D \neq \{\} \Rightarrow c ; \bigcap D = (\bigcap d \in D. c ; d) \). 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


