Concurrent Refinement Algebra and Rely Quotients

Julian Fell and Ian Hayes and Andrius Velykis

May 4, 2022

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

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
begin

unbundle *lattice-syntax*

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*
begin

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

abbreviation
refine :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\sqsubseteq$ *50*)
where
$c \sqsubseteq d \equiv$ *less-eq c d*

abbreviation
refine-strict :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\sqsubset$ *50*)
where
$c \sqsubset d \equiv$ *less c d*

Non-deterministic choice is monotonic in both arguments

lemma *inf-mono-left*: $a \sqsubseteq b \Longrightarrow a \sqcap c \sqsubseteq b \sqcap c$
using *inf-mono* **by** *auto*

lemma *inf-mono-right*: $c \sqsubseteq d \Longrightarrow a \sqcap c \sqsubseteq a \sqcap d$
using *inf-mono* **by** *auto*

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

lemma *Inf2-inf*: $\bigsqcap \{f\,x \mid x.\ x \in \{c, d\}\} = f\,c \sqcap f\,d$
proof −
have $\{f\,x \mid x.\ x \in \{c, d\}\} = \{f\,c, f\,d\}$ **by** *blast*
then have $\bigsqcap \{f\,x \mid x.\ x \in \{c, d\}\} = \bigsqcap \{f\,c, f\,d\}$ **by** *simp*
also have $\ldots = f\,c \sqcap f\,d$ **by** *simp*
finally show *?thesis* **.**
qed

Helper lemma for choice over indexed set.

lemma *INF-Inf*: $(\bigsqcap x{\in}X.\ f\,x) = (\bigsqcap \{f\,x \mid x.\ x \in X\})$
by (*simp add*: *Setcompr-eq-image*)

lemma (**in** −) *INF-absorb-args*: $(\bigsqcap i\,j.\ (f{::}nat \Rightarrow 'c{::}complete\text{-}lattice)\ (i + j)) = (\bigsqcap k.\ f\ k)$
proof (*rule order-class.order.antisym*)
show $(\bigsqcap k.\ f\,k) \leq (\bigsqcap i\,j.\ f\ (i + j))$
by (*simp add*: *complete-lattice-class.INF-lower complete-lattice-class.le-INF-iff*)
next
have $\bigwedge k.\ \exists\, i\,j.\ f\ (i + j) \leq f\,k$
by (*metis Nat.add-0-right order-refl*)

then have $\bigwedge k.\ \exists i.\ (\prod j.\ f\ (i+j)) \leq f\ k$
by (*meson UNIV-I complete-lattice-class.INF-lower2*)
then show $(\prod i\ j.\ f\ (i+j)) \leq (\prod k.\ f\ k)$
by (*simp add*: *complete-lattice-class.INF-mono*)
qed

lemma (**in** −) *nested-Collect*: $\{f\ y\ |y.\ y \in \{g\ x\ |x.\ x \in X\}\} = \{f\ (g\ x)\ |x.\ x \in X\}$
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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$
assumes *qual*: $P\ \{d\ x\ |x.\ x \in X\}$
assumes *f-Inf-distrib*: $\bigwedge c\ D.\ P\ D \Longrightarrow f\ c\ (\prod\ D) = \prod\ \{f\ c\ d\ |\ d\ .\ d \in D\ \}$
shows $f\ c\ (\prod x{\in}X.\ d\ x) = (\prod x{\in}X.\ f\ c\ (d\ x))$
proof −
have $f\ c\ (\prod x{\in}X.\ d\ x) = f\ c\ (\prod \{d\ x\ |x.\ x \in X\})$ **by** (*simp add*: *INF-Inf*)
also have $\ldots = (\prod \{f\ c\ dx\ |dx.\ dx \in \{d\ x\ |\ x.\ x \in X\}\})$ **by** (*simp add*: *qual f-Inf-distrib*)
also have $\ldots = (\prod \{f\ c\ (d\ x)\ |x.\ x \in X\})$ **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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$
assumes *f-Inf-distrib*: $\bigwedge c\ D.\ f\ c\ (\prod\ D) = \prod\ \{f\ c\ d\ |\ d\ .\ d \in D\ \}$
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 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* ⇒ '*a*) ⇒ *bool*

where
dist-over-sup F $\equiv (\forall\ X\ .\ F\ (\bigsqcup\ X) = (\bigsqcup x{\in}X.\ F\ (x)))$

abbreviation
dist-over-inf :: $('a{::}$*refinement-lattice* $\Rightarrow\ 'a) \Rightarrow$ *bool*
where
dist-over-inf F $\equiv (\forall\ X\ .\ F\ (\bigsqcap\ X) = (\bigsqcap x{\in}X.\ F\ (x)))$

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 $\bot$ as a left annihilator.

locale *seq* =
fixes *seq* :: $'a{::}$*refinement-lattice* $\Rightarrow\ 'a \Rightarrow\ 'a$ (**infixl** ; *90*)
assumes *seq-bot* [*simp*]: $\bot\ ;\ c = \bot$

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*:
  $(c::'a::\textit{refinement-lattice});(d_0 \sqcap d_1) \sqsubseteq (c;d_0 \sqcap c;d_1)$
begin

Left distribution implies sequential composition is monotonic is its right argument

lemma *seq-mono-right*: $c_0 \sqsubseteq c_1 \Longrightarrow d\ ;\ c_0 \sqsubseteq d\ ;\ c_1$
 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*:
  $(\bigsqcap C)\ ;\ d = (\bigsqcap (c::'a::\textit{refinement-lattice}) \in C.\ c\ ;\ d)$
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 simp add*: *image-comp*)

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*: $c_0 \sqsubseteq c_1 \Longrightarrow c_0\ ;\ d \sqsubseteq c_1\ ;\ d$
 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* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixr** $;\hat{}$ *80*) **where**
seq-power-0: $a ;\hat{} 0 = nil$
| *seq-power-Suc*: $a ;\hat{} Suc\ n = a ; (a ;\hat{} n)$

notation (*latex* **output**)
seq-power ((-$\hat{}$) [*1000*] *1000*)

notation (*HTML* **output**)
seq-power ((-$\hat{}$) [*1000*] *1000*)

lemma *seq-power-front*: $(a ;\hat{} n) ; a = a ; (a ;\hat{} n)$
by (*induct n*, *simp-all add*: *seq-assoc*)

lemma *seq-power-split-less*: $i < j \Longrightarrow (b ;\hat{} j) = (b ;\hat{} i) ; (b ;\hat{} (j - i))$
proof (*induct j arbitrary*: *i type*: *nat*)
case *0*
thus *?case* **by** *simp*
next
case (*Suc j*)
have $b ;\hat{} Suc\ j = b ; (b ;\hat{} i) ; (b ;\hat{} (j - i))$
using *Suc.hyps Suc.prems less-Suc-eq seq-assoc* **by** *auto*
also have $... = (b ;\hat{} i) ; b ; (b ;\hat{} (j - i))$ **by** (*simp add*: *seq-power-front*)
also have $... = (b ;\hat{} i) ; (b ;\hat{} (Suc\ j - i))$
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*: $c_1 \sqsubseteq d_1 \Longrightarrow c_2 \sqsubseteq d_2 \Longrightarrow c_1;c_2 \sqsubseteq d_1;d_2$
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* $\Rightarrow$ $'a \Rightarrow 'a$ (**infixl** $\parallel$ *75*)
 assumes *abort-par*: $\bot \parallel c = \bot$

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*: $D \neq \{\} \Longrightarrow c \parallel (\bigsqcap D) = (\bigsqcap d{\in}D.\ c \parallel d)$

begin

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

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

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

lemma *INF-INF-par-distrib*:
$X \neq \{\} \Longrightarrow Y \neq \{\} \Longrightarrow (\bigsqcap x{\in}X.\ c\ x) \parallel (\bigsqcap y{\in}Y.\ d\ y) = (\bigsqcap x{\in}X.\ \bigsqcap y{\in}Y.\ c\ x \parallel d\ y)$
proof −
assume *nonempty-X*: $X \neq \{\}$
assume *nonempty-Y*: $Y \neq \{\}$
have $(\bigsqcap x{\in}X.\ c\ x) \parallel (\bigsqcap y{\in}Y.\ d\ y) = (\bigsqcap x{\in}X.\ c\ x \parallel (\bigsqcap y{\in}Y.\ d\ y))$
using *INF-par-distrib* **by** (*metis nonempty-X*)
also have $\ldots = (\bigsqcap x{\in}X.\ \bigsqcap y{\in}Y.\ c\ x \parallel d\ y)$ **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 $\ldots = (\bigsqcap c \in \{c_0, c_1\}.\ c \parallel d)$ **using** *Inf-par-distrib* **by** (*meson insert-not-empty*)

also have $\ldots = c_0 \parallel d \sqcap c_1 \parallel d$ **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*: $(a \sqcap b) \parallel (c \sqcap d) = (a \parallel c) \sqcap (a \parallel d) \sqcap (b \parallel c) \sqcap (b \parallel d)$
by (*simp add*: *inf-commute inf-par-distrib inf-par-distrib2 inf-sup-aci*(*3*))

lemma *par-mono*: $c_1 \sqsubseteq d_1 \Longrightarrow c_2 \sqsubseteq d_2 \Longrightarrow c_1 \parallel c_2 \sqsubseteq d_1 \parallel d_2$
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 $\Cap$ is similar to least upper bound ($\sqcup$) but is abort strict, i.e. the lattice bottom is an annihilator: $c \Cap \bot = \bot$. 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** $\Cap$ *80*)
  assumes *conj-bot-right*: $c \Cap \bot = \bot$

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*]: $\bot \Cap c = \bot$
using *conj-bot-right local.conj-commute* **by** *fastforce*

lemma *conj-not-bot*: $a \Cap b \neq \bot \Longrightarrow a \neq \bot \wedge b \neq \bot$
  using *conj-bot-right* **by** *auto*

lemma *conj-distrib1*: $c \Cap (d_0 \Cap d_1) = (c \Cap d_0) \Cap (c \Cap d_1)$
  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*: $D \neq \{\} \Longrightarrow (\bigsqcap D) \Cap c = (\bigsqcap d{\in}D.\ d \Cap c)$

begin

lemma *conj-Inf-distrib*: $D \neq \{\} \Longrightarrow c \Cap (\bigsqcap D) = (\bigsqcap d{\in}D.\ c \Cap d)$
  using *Inf-conj-distrib conj-commute* **by** *auto*

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

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

lemma *conj-mono*: $c_0 \sqsubseteq d_0 \Longrightarrow c_1 \sqsubseteq d_1 \Longrightarrow c_0 \Cap c_1 \sqsubseteq d_0 \Cap d_1$
  by (*metis inf.absorb-iff1 inf-conj-product inf-right-idem*)

lemma *conj-mono-left*: $c_0 \sqsubseteq c_1 \Longrightarrow c_0 \Cap d \sqsubseteq c_1 \Cap d$
  by (*simp add*: *conj-mono*)

lemma *conj-mono-right*: $c_0 \sqsubseteq c_1 \Longrightarrow d \Cap c_0 \sqsubseteq d \Cap c_1$
  by (*simp add*: *conj-mono*)

lemma *conj-refine*: $c_0 \sqsubseteq d \Longrightarrow c_1 \sqsubseteq d \Longrightarrow c_0 \Cap c_1 \sqsubseteq d$
  by (*metis conj-idem conj-mono*)

lemma *refine-to-conj*: $c \sqsubseteq d_0 \Longrightarrow c \sqsubseteq d_1 \Longrightarrow c \sqsubseteq d_0 \Cap d_1$
  by (*metis conj-idem conj-mono*)

lemma *conjoin-non-aborting*: $chaos \sqsubseteq c \Longrightarrow d \sqsubseteq d \Cap c$
  by (*metis conj-mono order.refl conj-chaos*)

lemma *conjunction-sup*: $c \Cap d \sqsubseteq c \sqcup d$
  by (*simp add*: *conj-refine*)

lemma *conjunction-sup-nonaborting*:
assumes $chaos \sqsubseteq c$ **and** $chaos \sqsubseteq d$
shows $c \Cap d = c \sqcup d$
proof (*rule antisym*)
show $c \sqcup d \sqsubseteq c \Cap d$ **using** *assms*(*1*) *assms*(*2*) *conjoin-non-aborting local.conj-commute*
by *fastforce*
next
show $c \Cap d \sqsubseteq c \sqcup d$ **by** (*metis conjunction-sup*)
qed

lemma *conjoin-top*: $chaos \sqsubseteq c \Longrightarrow c \Cap \top = \top$
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*: $nil \parallel nil \sqsubseteq nil$
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 conjunction and parallel.

locale *conjunction-parallel* = *conj-distrib* + *par-distrib* +
assumes *chaos-par-top*: $\top \sqsubseteq \mathit{chaos} \parallel \top$
assumes *chaos-par-chaos*: $\mathit{chaos} \sqsubseteq \mathit{chaos} \parallel \mathit{chaos}$
assumes *parallel-interchange*: $(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1)$
begin

lemma *chaos-skip*: $\mathit{chaos} \sqsubseteq \mathit{skip}$
proof −
have $\mathit{chaos} = (\mathit{chaos} \parallel \mathit{skip}) \Cap (\mathit{skip} \parallel \mathit{chaos})$ **by** *simp*
then have $\ldots \sqsubseteq (\mathit{chaos} \Cap \mathit{skip}) \parallel (\mathit{skip} \Cap \mathit{chaos})$ **using** *parallel-interchange* **by** *blast*
thus *?thesis* **by** *auto*
qed

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

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

lemma *skip-conj-top*: $\mathit{skip} \Cap \top = \top$
by (*simp add*: *chaos-skip conjoin-top*)

lemma *conj-distrib2*: $c \sqsubseteq c \parallel c \Longrightarrow c \Cap (d_0 \parallel d_1) \sqsubseteq (c \Cap d_0) \parallel (c \Cap d_1)$
proof −
assume $c \sqsubseteq c \parallel c$
then have $c \Cap (d_0 \parallel d_1) \sqsubseteq (c \parallel c) \Cap (d_0 \parallel d_1)$ **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*: $\mathit{chaos} \sqsubseteq \mathit{chaos};\mathit{chaos}$
assumes *sequential-interchange*: $(c_0;c_1) \Cap (d_0;d_1) \sqsubseteq (c_0 \Cap d_0);(c_1 \Cap d_1)$
begin

lemma *chaos-nil*: $chaos \sqsubseteq nil$
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 $chaos \sqsubseteq chaos;chaos$ **by** (*simp add*: *chaos-seq-chaos*)
next
show $chaos;chaos \sqsubseteq chaos$ **using** *chaos-nil*
using *seq-mono-left seq-nil-left* **by** *fastforce*
qed

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

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

lemma *conj-distrib3*: $c \sqsubseteq c;c \Longrightarrow c \Cap (d_0 \,;\, d_1) \sqsubseteq (c \Cap d_0);(c \Cap d_1)$
proof −
assume $c \sqsubseteq c;c$
then have $c \Cap (d_0;d_1) \sqsubseteq (c;c) \Cap (d_0;d_1)$ **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 $cra = sequential\text{-}parallel + conjunction\text{-}parallel + conjunction\text{-}sequential$

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*]: $\{F\ x\ |x.\ x = a \vee x = b\} = \{F\ a, F\ b\}$
by *auto*

locale *lower-galois-connections*
begin

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

lemma *dist-inf-mono*:
assumes *distF*: *dist-over-inf F*
shows *mono F*
proof
fix $x :: 'a$ **and** $y :: 'a$
assume $x \sqsubseteq y$
then have $F\ x = F\ (x \sqcap y)$ **by** (*simp add*: *le-iff-inf*)
also have $\ldots = F\ x \sqcap F\ y$
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* $\Longrightarrow x \sqsubseteq (F \circ F^\flat)\ x$
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 $x \sqsubseteq \bigsqcap \{F\ y \mid y.\ x \sqsubseteq F\ y\}$ **by** (*simp add*: *X-def Y-def*)
then have $x \sqsubseteq F\ (\bigsqcap \{y.\ x \sqsubseteq F\ y\})$ **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* $\Longrightarrow ((F^\flat)\ x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq F\ y)$
proof
assume $x \sqsubseteq F\ y$
then have $\bigsqcap \{y.\ x \sqsubseteq F\ y\} \sqsubseteq y$ **by** (*simp add*: *Inf-lower*)
thus $(F^\flat)\ x \sqsubseteq y$ **by** (*metis l-adjoint-def*)
next
assume *dist*: *dist-over-inf F* **then have** *monoF*: *mono F* **by** (*simp add*: *dist-inf-mono*)
assume $(F^\flat)\ x \sqsubseteq y$ **then have** *a*: $F\ ((F^\flat)\ x) \sqsubseteq F\ y$ **by** (*simp add*: *monoD monoF*)
have $x \sqsubseteq F\ ((F^\flat)\ x)$ **using** *dist l-cancellation* **by** *simp*
thus $x \sqsubseteq F\ y$ **using** *a* **by** *auto*
qed

lemma *v-simple-fusion*: *mono G* $\Longrightarrow \forall x.\ ((F \circ G)\ x \sqsubseteq (H \circ F)\ x) \Longrightarrow F\ (gfp\ G) \sqsubseteq gfp\ H$
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*: $\bigwedge x.\ ((H \circ F)\ x \sqsubseteq (F \circ G)\ x)$
shows $gfp\ H \sqsubseteq F\ (gfp\ G)$
proof −
have $(gfp\ H) \sqsubseteq (F \circ F^\flat)\ (gfp\ H)$ **using** *distribF l-cancellation* **by** *simp*
then have $H\ (gfp\ H) \sqsubseteq H\ ((F \circ F^\flat)\ (gfp\ H))$ **by** (*simp add*: *monoD monoH*)
then have $H\ (gfp\ H) \sqsubseteq F\ ((G \circ F^\flat)\ (gfp\ H))$ **using** *comp-geq* **by** (*metis comp-def refine-trans*)
then have $(F^\flat)\ (H\ (gfp\ H)) \sqsubseteq (G \circ F^\flat)\ (gfp\ H)$ **using** *distribF* **by** (*metis* (*mono-tags*) *l-galois-connection*)
then have $(F^\flat)\ (gfp\ H) \sqsubseteq (gfp\ G)$ **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*: $\bigwedge x.\ ((F \circ G)\ x = (H \circ F)\ x)$
shows *F* (*gfp G*) = *gfp H*
proof (*rule antisym*)
show *F* (*gfp G*) $\sqsubseteq$ (*gfp H*) **by** (*metis fgh-comp le-less v-simple-fusion monoG*)
next
have $\bigwedge x.\ ((H \circ F)\ x \sqsubseteq (F \circ G)\ x)$ **using** *fgh-comp* **by** *auto*
then show *gfp H* $\sqsubseteq$ *F* (*gfp G*) **using** *monoH distF fusion-gfp-geq* **by** *blast*
qed

end

7.3 Upper Galois connections

locale *upper-galois-connections*
begin

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

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 ... = *F x* $\sqcup$ *F y*
proof −
from *distF*
have *F* ($\bigsqcup \{x, y\}$) = $\bigsqcup \{F\ x, F\ y\}$ **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* $\Longrightarrow$ $(F \circ F^{\#})\ x \sqsubseteq x$
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 $\bigsqcup \{F\ y \mid y.\ F\ y \sqsubseteq x\} \sqsubseteq x$ **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 *?thesis* **by** (*metis comp-def u-adjoint-def*)
qed

lemma *u-galois-connection*: *dist-over-sup F* $\Longrightarrow$ $(F\ x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq (F^{\#})\ y)$
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*)
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* $\Longrightarrow \forall x.\ ((F \circ G)\ x \sqsubseteq (G \circ H)\ x) \Longrightarrow$ *lfp F* $\sqsubseteq$ *G* (*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*: $\bigwedge x.\ ((F \circ G)\ x \sqsubseteq (H \circ F)\ x)$
shows *F* (*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 $(G \circ F^{\#})\ (lfp\ H) \sqsubseteq (F^{\#})\ (H\ (lfp\ H))$ **using** *distribF* **by** (*metis* (*mono-tags*) *u-galois-connection*)
then have $(lfp\ G) \sqsubseteq (F^{\#})\ (lfp\ H)$ **by** (*metis comp-def def-lfp-unfold lfp-lowerbound monoH*)
thus $F\ (lfp\ G) \sqsubseteq (lfp\ H)$ **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*: $\bigwedge x.\ ((F \circ G)\ x = (H \circ F)\ x)$
shows $F\ (lfp\ G) = (lfp\ H)$
proof (*rule antisym*)
show $lfp\ H \sqsubseteq F\ (lfp\ G)$ **by** (*metis monoG fgh-comp eq-iff upper-galois-connections.u-simple-fusion*)
next
have $\bigwedge x.\ (F \circ G)\ x \sqsubseteq (H \circ F)\ x$ **using** *fgh-comp* **by** *auto*
then show $F\ (lfp\ G) \sqsubseteq (lfp\ H)$ **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
$iter :: 'a \Rightarrow 'a\ (\text{-}^{\omega}\ [103]\ 102)$
where
$c^{\omega} \equiv lfp\ (\lambda\ x.\ nil \sqcap c;x)$

lemma *iter-step-mono*: *mono* $(\lambda\, x.\ nil \sqcap c;x)$
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*: $c^\omega = nil \sqcap c;c^\omega$
using *iter-def iter-step-mono lfp-unfold* **by** *auto*

lemma *iter-induct-nil*: $nil \sqcap c;x \sqsubseteq x \Longrightarrow c^\omega \sqsubseteq x$
by (*simp add*: *iter-def lfp-lowerbound*)

lemma *iter0*: $c^\omega \sqsubseteq nil$
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 $c^\omega;c^\omega \sqsubseteq c^\omega$ **using** *iter0 seq-mono-right* **by** *fastforce*
next
have *a*: $nil \sqcap c;c^\omega;c^\omega \sqsubseteq nil \sqcap c;c^\omega \sqcap c;c^\omega;c^\omega$
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*: $\ldots = (nil \sqcap c;c^\omega);c^\omega$ **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*: $c \sqsubseteq d \Longrightarrow c^\omega \sqsubseteq d^\omega$
proof −
assume $c \sqsubseteq d$
then have $nil \sqcap c;d^\omega \sqsubseteq d;d^\omega$ **by** (*metis inf.absorb-iff2 inf-left-commute inf-seq-distrib*)
then have $nil \sqcap c;d^\omega \sqsubseteq d^\omega$ **by** (*metis inf.bounded-iff inf-sup-ord*(*1*) *iter-unfold*)
thus *?thesis* **by** (*simp add*: *iter-induct-nil*)
qed

lemma *iter-abort*: $\bot = nil^\omega$
by (*simp add*: *antisym iter-induct-nil*)

lemma *nil-iter*: $\top^\omega = nil$
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
fiter :: $'a \Rightarrow 'a$ (-* [*101*] *100*)
where
$c^\star \equiv gfp\ (\lambda\ x.\ nil \sqcap c;x)$

lemma *fin-iter-step-mono*: *mono* $(\lambda\ x.\ nil \sqcap c;x)$
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*: $c^\star = nil \sqcap c;c^\star$
using *fiter-def gfp-unfold fin-iter-step-mono* **by** *auto*

lemma *fiter-induct-nil*: $x \sqsubseteq nil \sqcap c;x \Longrightarrow x \sqsubseteq c^\star$
by (*simp add*: *fiter-def gfp-upperbound*)

lemma *fiter0*: $c^\star \sqsubseteq nil$
by (*metis fiter-unfold inf.cobounded1*)

lemma *fiter1*: $c^\star \sqsubseteq c$
by (*metis fiter0 fiter-unfold inf-le2 order.trans seq-mono-right seq-nil-right*)

lemma *fiter-induct-eq*: $c^\star;d = gfp\ (\lambda\ x.\ c;x \sqcap d)$
proof −
define F **where** $F = (\lambda\ x.\ x;d)$
define G **where** $G = (\lambda\ x.\ nil \sqcap c;x)$
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 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 $(gfp\ (\lambda\ x.\ nil \sqcap c;x));d = gfp\ (\lambda\ x.\ c;x \sqcap d)$ **using** *F-def G-def H-def inf-commute* **by** *simp*
thus *?thesis* **by** (*metis fiter-def*)
qed

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

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

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

lemma *fiter-mono*: $c \sqsubseteq d \Longrightarrow c^\star \sqsubseteq d^\star$
proof −
assume $c \sqsubseteq d$
then have $c^\star \sqsubseteq nil \sqcap d;c^\star$ **by** (*metis fiter0 fiter1 fiter2 inf*.*bounded-iff refine-trans seq-mono-left*)
thus *?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 :: $'a \Rightarrow 'a$ (-$^\infty$ [105] 106)
where
$c^\infty \equiv lfp\ (\lambda\, x.\ c;x)$

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*: $c;x \sqsubseteq x \Longrightarrow c^\infty \sqsubseteq x$
by (*simp add*: *infiter-def lfp-lowerbound*)

theorem *infiter-unfold-any*: $c^\infty = (c\ ;\hat{}\ i)\ ;\ c^\infty$
proof (*induct i*)
case *0*
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 $\forall\, a.\ (\bot::'a) \sqsubseteq a$
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*: $c^\omega \sqsubseteq c^\star$
by (*metis seq-nil-right order.refl iter-unfold fiter-induct*)

lemma *iter-absorption* [*simp*]: $(c^\omega)^\star = c^\omega$
proof (*rule antisym*)
show $(c^\omega)^\star \sqsubseteq c^\omega$ **by** (*metis fiter1*)
next
show $c^\omega \sqsubseteq (c^\omega)^\star$ **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 $c \ ; (c^\omega \ ; \top) = nil \ ; \top \sqcap c \ ; c^\omega \ ; \top$
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.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 $c^\infty \sqsubseteq c^\star \ ; \top$
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

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*: $c;(d_0 \sqcap d_1) = c;d_0 \sqcap c;d_1$
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* = *seq-distrib-right* +
 assumes *seq-Inf-distrib*: $D \neq \{\} \Longrightarrow c \,;\, \bigsqcap D = (\bigsqcap d{\in}D.\ c \,;\, d)$
begin

sublocale *seq-distrib*
proof *unfold-locales*
 fix $c{::}'a$ **and** $d_0{::}'a$ **and** $d_1{::}'a$
 have $\{d_0, d_1\} \neq \{\}$ **by** *simp*
 then have $c \,;\, \bigsqcap \{d_0, d_1\} = \bigsqcap \{c \,;\, d \mid d.\ d \in \{d_0, d_1\}\}$ **using** *seq-Inf-distrib*
 proof −
 have $\bigsqcap\ ((;)\ c\ `\ \{d_0, d_1\}) = \bigsqcap \{c \,;\, a \mid a.\ a \in \{d_0, d_1\}\}$
 using *INF-Inf* **by** *blast*
 then show *?thesis*
 using ‹$\bigwedge(c{::}'a{::}$*refinement-lattice*$)\ D{::}'a{::}$*refinement-lattice set*. $D \neq \{\} \Longrightarrow c \,;\, \bigsqcap D = (\bigsqcap d{::}'a{::}$*refinement-lattice*$\in D.\ c \,;\, d)$› ‹$\{d_0{::}'a{::}$*refinement-lattice*$, d_1{::}'a{::}$*refinement-lattice*$\} \neq \{\}$› **by** *presburger*
 qed
 also have $\ldots = c \,;\, d_0 \sqcap c \,;\, d_1$ **by** (*simp only*: *Inf2-inf*)
 finally show $c \,;\, (d_0 \sqcap d_1) \sqsubseteq c \,;\, d_0 \sqcap c \,;\, d_1$ **by** *simp*
qed

lemma *seq-INF-distrib*: $X \neq \{\} \Longrightarrow c \,;\, (\bigsqcap x{\in}X.\ d\ x) = (\bigsqcap x{\in}X.\ c \,;\, d\ x)$
proof −
 assume *xne*: $X \neq \{\}$
 have *a*: $c \,;\, (\bigsqcap x{\in}X.\ d\ x) = c \,;\, \bigsqcap (d\ `\ X)$ **by** *auto*
 also have *b*: $\ldots = (\bigsqcap d{\in}(d\ `\ X).\ c \,;\, d)$ **by** (*meson image-is-empty seq-Inf-distrib xne*)
 also have *c*: $\ldots = (\bigsqcap x{\in}X.\ c \,;\, d\ x)$ **by** (*simp add*: *image-comp*)
 finally show *?thesis* **by** (*simp add*: *b image-comp*)

qed

lemma *seq-INF-distrib-UNIV*: $c \,;\, (\bigsqcap x.\ d\ x) = (\bigsqcap x.\ c \,;\, d\ x)$
by (*simp add*: *seq-INF-distrib*)

lemma *INF-INF-seq-distrib*: $Y \neq \{\} \Longrightarrow (\bigsqcap x{\in}X.\ c\ x) \,;\, (\bigsqcap y{\in}Y.\ d\ y) = (\bigsqcap x{\in}X.\ \bigsqcap y{\in}Y.\ c\ x \,;\, d\ y)$
by (*simp add*: *INF-seq-distrib seq-INF-distrib*)

lemma *INF-INF-seq-distrib-UNIV*: $(\bigsqcap x.\ c\ x) \,;\, (\bigsqcap y.\ d\ y) = (\bigsqcap x.\ \bigsqcap y.\ c\ x \,;\, d\ y)$
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 $(\bigsqcap j.\ f\ i\ j) =$
$(\bigsqcap j{\in}\{j.\ i = j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ i < j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ j < i\}.\ f\ i\ j)$
proof −
have *univ-part*: $UNIV = \{j.\ i = j\} \cup \{j.\ i < j\} \cup \{j.\ j < i\}$ **by** *auto*
have $(\bigsqcap j \in \{j.\ i = j\} \cup \{j.\ i < j\} \cup \{j.\ j < i\}.\ f\ i\ j) =$
$(\bigsqcap j{\in}\{j.\ i = j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ i < j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ j < i\}.\ f\ i\ j)$ **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*

shows $(\bigsqcap i.\ \bigsqcap j.\ f\ i\ j) =$
$(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ i = j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ i < j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ i\ j)$
proof −
have $(\bigsqcap i.\ \bigsqcap j.\ f\ i\ j) = (\bigsqcap i.\ ((\bigsqcap j{\in}\{j.\ i = j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ i < j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap j{\in}\{j.\ j < i\}.\ f\ i\ j)))$
by (*simp add*: *INF-partition-nat3*)
also have $\ldots = (\bigsqcap i.\ \bigsqcap j{\in}\{j.\ i = j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ i < j\}.\ f\ i\ j) \sqcap$
$(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ i\ j)$
by (*simp add*: *INF-inf-distrib*)
finally show *?thesis* **.**
qed

lemma *INF-nat-shift*: $(\bigsqcap i{\in}\{i.\ 0 < i\}.\ f\ i) = (\bigsqcap i.\ f\ (Suc\ i))$
by (*metis greaterThan-0 greaterThan-def range-composition*)

lemma *INF-nat-minus*:
fixes $f :: nat \Rightarrow {'a}$::*refinement-lattice*
shows $(\bigsqcap j{\in}\{j.\ i < j\}.\ f\ (j - i)) = (\bigsqcap k{\in}\{k.\ 0 < k\}.\ f\ k)$
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 $f :: nat \Rightarrow nat \Rightarrow {'a}$::*refinement-lattice*
shows $(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ j\ (i - j)) = (\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j < i\}.\ f\ j\ (i - j))$
proof (*rule antisym*)
have $\bigwedge jj\ ii.\ jj < ii \Longrightarrow \exists i.\ \exists j{<}i.\ f\ j\ (i - j) \sqsubseteq f\ jj\ (ii - jj)$
by *blast*
then have $\bigwedge jj\ ii.\ jj < ii \Longrightarrow \exists i.\ (\bigsqcap j{\in}\{j.\ j < i\}.\ f\ j\ (i - j)) \sqsubseteq f\ jj\ (ii - jj)$
by (*meson INF-lower mem-Collect-eq*)
then have $\bigwedge jj\ ii.\ jj < ii \Longrightarrow (\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ j\ (i - j)) \sqsubseteq f\ jj\ (ii - jj)$
by (*meson UNIV-I INF-lower dual-order.trans*)
then have $\bigwedge jj.\ (\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ j\ (i - j)) \sqsubseteq (\bigsqcap ii{\in}\{ii.\ jj < ii\}.\ f\ jj\ (ii - jj))$
by (*metis* (*mono-tags*, *lifting*) *INF-greatest mem-Collect-eq*)
then have $(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j < i\}.\ f\ j\ (i - j)) \sqsubseteq (\bigsqcap jj.\ \bigsqcap ii{\in}\{ii.\ jj < ii\}.\ f\ jj\ (ii - jj))$
by (*simp add*: *INF-greatest*)

then show $(\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq (\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j))$
by *simp*
next
have $\bigwedge ii\ jj.\ jj<ii \Longrightarrow \exists j.\ \exists i{>}j.\ f\,j\ (i-j) \sqsubseteq f\,jj\ (ii-jj)$
by *blast*
then have $\bigwedge ii\ jj.\ jj<ii \Longrightarrow \exists j.\ (\bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq f\,jj\ (ii-jj)$
by (*meson INF-lower mem-Collect-eq*)
then have $\bigwedge ii\ jj.\ jj<ii \Longrightarrow (\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq f\,jj\ (ii-jj)$
by (*meson UNIV-I INF-lower dual-order.trans*)
then have $\bigwedge ii.\ (\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq (\bigsqcap jj{\in}\{jj.\ jj<ii\}.\ f\,jj\ (ii-jj))$
by (*metis* (*mono-tags*, *lifting*) *INF-greatest mem-Collect-eq*)
then have $(\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq (\bigsqcap ii.\ \bigsqcap jj{\in}\{jj.\ jj<ii\}.\ f\,jj\ (ii-jj))$
by (*simp add*: *INF-greatest*)
then show $(\bigsqcap j.\ \bigsqcap i{\in}\{i.\ j<i\}.\ f\,j\ (i-j)) \sqsubseteq (\bigsqcap i.\ \bigsqcap j{\in}\{j.\ j<i\}.\ f\,j\ (i-j))$
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*: $c^\omega = c^\star \sqcap c^\infty$
proof −
define F **where** $F = (\lambda\ x.\ c^\star \sqcap x)$
define G **where** $G = (\lambda\ x.\ c;x)$
define H **where** $H = (\lambda\ x.\ nil \sqcap c;x)$

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

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

have *adjoint*: *dist-over-sup F* **by** (*simp add*: *F-def inf-Sup*)
have *monoH*: *mono H* **by** (*metis H-def inf-mono monoI order-refl seq-mono-right*)
have *monoG*: *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 monoH monoG fusion-lfp-eq* **by** *blast*
then have $c^\star \sqcap lfp\ (\lambda\, x.\ c;x) = lfp\ (\lambda\, x.\ nil \sqcap c;x)$
using *F-def G-def H-def* **by** *blast*
thus *?thesis* **by** (*simp add*: *infiter-def iter-def*)
qed

lemma *iter-induct-isolate*: $c^\star;d \sqcap c^\infty = lfp\ (\lambda\, x.\ d \sqcap c;x)$
proof −
define *F* **where** $F = (\lambda\, x.\ c^\star;d \sqcap x)$
define *G* **where** $G = (\lambda\, x.\ c;x)$
define *H* **where** $H = (\lambda\, x.\ d \sqcap c;x)$

have *FG*: $F \circ G = (\lambda\, x.\ c^\star;d \sqcap c;x)$ **using** *F-def G-def* **by** *auto*
have *HF*: $H \circ F = (\lambda\, x.\ d \sqcap c;c^\star;d \sqcap c;x)$ **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*: $c^\star;d = (nil \sqcap c;c^\star);d$ **using** *fiter-unfold* **by** *auto*
have *distribute*: $c^\star;d = d \sqcap c;c^\star;d$ **by** (*simp add*: *unroll inf-seq-distrib*)
have *FGx*: $(F \circ G)\ x = d \sqcap c;c^\star;d \sqcap c;x$ **using** *FG distribute* **by** *simp*

have *adjoint*: *dist-over-sup F* **by** (*simp add*: *F-def inf-Sup*)
have *monoH*: *mono H* **by** (*metis H-def inf-mono monoI order-refl seq-mono-right*)
have *monoG*: *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 monoH monoG fusion-lfp-eq* **by** *blast*
then have $c^\star;d \sqcap lfp\ (\lambda\, x.\ c;x) = lfp\ (\lambda\, x.\ d \sqcap c;x)$
using *F-def G-def H-def* **by** *blast*
thus *?thesis* **by** (*simp add*: *infiter-def*)
qed

lemma *iter-induct-eq*: $c^\omega;d = lfp\ (\lambda\, x.\ d \sqcap c;x)$
proof −
have $c^\omega;d = c^\star;d \sqcap c^\infty;d$ **by** (*simp add*: *isolation inf-seq-distrib*)
then have $c^\star;d \sqcap c^\infty;d = c^\star;d \sqcap c^\infty$ **by** (*simp add*: *infiter-annil*)
then have $c^\star;d \sqcap c^\infty = lfp\ (\lambda\, x.\ d \sqcap c;x)$ **by** (*simp add*: *iter-induct-isolate*)

thus *?thesis*
by (*simp add*: ‹c^ω ; $d = c^\star$; $d \sqcap c^\infty$; d› ‹$c^\star$; $d \sqcap c^\infty$; $d = c^\star$; $d \sqcap c^\infty$›)
qed

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

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

lemma *iter-isolate2*: $c;c^\star;d \sqcap c^\infty = c;c^\omega;d$
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 $c;c^\omega;(d;c^\omega)^\omega \sqcap (d;c^\omega)^\omega \sqsubseteq c^\omega;(d;c^\omega)^\omega$ **by** (*metis inf-commute order.refl inf-seq-distrib seq-nil-left iter-unfold*)
thus $(c \sqcap d)^\omega \sqsubseteq c^\omega;(d;c^\omega)^\omega$ **by** (*metis inf.left-commute iter-induct-nil iter-unfold seq-assoc inf-seq-distrib*)
next
have $(c;(c \sqcap d)^\omega \sqcap d;(c \sqcap d)^\omega) \sqcap nil \sqsubseteq (c \sqcap d)^\omega$ **by** (*metis inf-commute order.refl inf-seq-distrib iter-unfold*)
then have *a*: $c^\omega;(d;(c \sqcap d)^\omega \sqcap nil) \sqsubseteq (c \sqcap d)^\omega$
proof −
have $nil \sqcap d;(c \sqcap d)^\omega \sqcap c;(c \sqcap d)^\omega \sqsubseteq (c \sqcap d)^\omega$
by (*metis eq-iff inf.semigroup-axioms inf-commute inf-seq-distrib iter-unfold semigroup.assoc*)
thus *?thesis* **using** *iter-induct-eq* **by** (*metis inf-sup-aci*(*1*) *iter-induct*)
qed
then have $d;c^\omega;(d;(c \sqcap d)^\omega \sqcap nil) \sqcap nil \sqsubseteq d;(c \sqcap d)^\omega \sqcap nil$ **by** (*metis inf-mono order.refl seq-assoc seq-mono*)
then have $(d;c^\omega)^\omega \sqsubseteq d;(c \sqcap d)^\omega \sqcap nil$ **by** (*metis inf-commute iter-induct-nil*)
then have $c^\omega;(d;c^\omega)^\omega \sqsubseteq c^\omega;(d;(c \sqcap d)^\omega \sqcap nil)$ **by** (*metis order.refl seq-mono*)
thus $c^\omega;(d;c^\omega)^\omega \sqsubseteq (c \sqcap d)^\omega$ **using** *a refine-trans* **by** *blast*
qed

lemma *iter-leapfrog-var*: $(c;d)^\omega;c \sqsubseteq c;(d;c)^\omega$
proof −
have $c \sqcap c;d;c;(d;c)^\omega \sqsubseteq c;(d;c)^\omega$
by (*metis iter-unfold order-refl seq-assoc seq-inf-distrib seq-nil-right*)
thus *?thesis* **using** *iter-induct-eq* **by** (*metis iter-induct seq-assoc*)
qed

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 $(d;c)^\omega \sqsubseteq ((d;c)^\omega;d);c \sqcap nil$ **by** (*metis inf.bounded-iff order.refl seq-assoc seq-mono iter-unfold iter1 iter2*)
 then have $(d;c)^\omega \sqsubseteq (d;(c;d)^\omega);c \sqcap nil$ **by** (*metis inf.absorb-iff2 inf.boundedE inf-assoc iter-leapfrog-var inf-seq-distrib*)
 then have $c;(d;c)^\omega \sqsubseteq c;d;(c;d)^\omega;c \sqcap nil;c$ **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*: $c;(d;c)^\star = (c;d)^\star;c$
proof −
 have *lr*: $c;(d;c)^\star \sqsubseteq (c;d)^\star;c$
 proof −
  have $(d\ ;\ c)^\star = nil \sqcap d\ ;\ c\ ;\ (d\ ;\ c)^\star$
   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*: $(c;d)^\star;c \sqsubseteq c;(d;c)^\star$
 proof −
  have *a1*: $(c;d)^\star;c = c \sqcap c;d;(c;d)^\star;c$
   by (*metis finite-iteration.fiter-unfold finite-iteration-axioms inf-seq-distrib seq-nil-left*)
  have *a2*: $(c;d)^\star;c \sqsubseteq c;(d;c)^\star \longleftrightarrow c \sqcap c;d;(c;d)^\star;c \sqsubseteq c;(d;c)^\star$ **by** (*simp add*: *a1*)
  then have *a3*: $... \longleftrightarrow c;(\ nil \sqcap d;(c;d)^\star;c) \sqsubseteq c;(d;c)^\star$
   by (*metis a1 eq-iff fiter-unfold lr seq-assoc seq-inf-distrib seq-nil-right*)
  have *a4*: $(nil \sqcap d;(c;d)^\star;c) \sqsubseteq (d;c)^\star \Longrightarrow c;(\ nil \sqcap d;(c;d)^\star;c) \sqsubseteq c;(d;c)^\star$
   using *seq-mono-right* **by** *blast*
  have *a5*: $(nil \sqcap d;(c;d)^\star;c) \sqsubseteq (d;c)^\star$
   proof −
    have *f1*: $d\ ;\ (c\ ;\ d)^\star\ ;\ c \sqcap nil = d\ ;\ ((c\ ;\ d)^\star\ ;\ c) \sqcap nil \sqcap nil$
     by (*simp add*: *seq-assoc*)
    have $d\ ;\ c\ ;\ (d\ ;\ (c\ ;\ d)^\star\ ;\ c \sqcap nil) = d\ ;\ ((c\ ;\ d)^\star\ ;\ c)$
     by (*metis* (*no-types*) *a1 inf-sup-aci*(*1*) *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*

inf.cobounded1 inf-sup-aci(1) seq-nil-right)
  qed
 thus *?thesis* **using** *a2 a3 a4* **by** *blast*
qed
thus *?thesis* **by** (*simp add*: *eq-iff lr*)
qed

end

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

begin

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

lemma *fiter-seq-choice-nonempty*: $c\ ;\ c^\star = (\bigsqcap i{\in}\{i.\ 0 < i\}.\ c\ ;\!\hat{}\ i)$
proof −
 have $(\bigsqcap i{\in}\{i.\ 0 < i\}.\ c\ ;\!\hat{}\ i) = (\bigsqcap i.\ c\ ;\!\hat{}\ (Suc\ i))$ **by** (*simp add*: *INF-nat-shift*)
 also have $\ldots = (\bigsqcap i.\ c\ ;\ (c\ ;\!\hat{}\ i))$ **by** *simp*

also have ... $= c$; $(\bigsqcap i.\ c\ ;\hat{}\ i)$ **by** (*simp add*: *seq-INF-distrib-UNIV*)
also have ... $= c$; $c^\star$ **by** (*simp add*: *fiter-seq-choice*)
finally show *?thesis* **by** *simp*
qed

end

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

begin

lemma *conj-distrib4*: $c^\star \sqinter d^\star \sqsubseteq (c \sqinter d)^\star$
proof −
have $c^\star \sqinter d^\star = (nil \sqcap (c;c^\star)) \sqinter d^\star$ **by** (*metis fiter-unfold*)
then have $c^\star \sqinter d^\star = (nil \sqinter d^\star) \sqcap ((c;c^\star) \sqinter d^\star)$ **by** (*simp add*: *inf-conj-distrib*)
then have $c^\star \sqinter d^\star \sqsubseteq nil \sqcap ((c;c^\star) \sqinter (d;d^\star))$ **by** (*metis conj-idem fiter0 fiter-unfold inf.bounded-iff inf-le2 local.conj-mono*)
then have $c^\star \sqinter d^\star \sqsubseteq nil \sqcap ((c \sqinter d);(c^\star \sqinter d^\star))$ **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
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 :: *'a* ⇒ *'a* ⇒ *'a* (**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* ⊑ *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 $\bigsqcap\ C \sqsubseteq (\bigsqcap\ D)$ **by** (*simp add*: *Inf-mono*)
then have $c \sqsubseteq \bigsqcap \{\ d \parallel i \mid d.\ (c \sqsubseteq d \parallel i)\}$ **by** (*simp add*: *C-def D-def*)
also have $\ldots = \bigsqcap \{\ d \parallel i \mid d.\ d \in \{d.\ (c \sqsubseteq d \parallel i)\}\}$ **by** *simp*
also have $\ldots = (\bigsqcap d \in \{d.\ (c \sqsubseteq d \parallel i)\}.\ d \parallel i)$ **by** (*simp add*: *INF-Inf*)
also have $\ldots = \bigsqcap \{\ d \mid d.\ (c \sqsubseteq d \parallel i)\} \parallel i$
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*: $\{\ d \mid d.\ (c \sqsubseteq d \parallel i)\} \neq \{\}$

have *b*: $\{d.\ (c \sqsubseteq d \parallel i)\} \neq \{\}$ **using** *a* **by** *blast*
then have $(\bigsqcap d \in \{d.\ (c \sqsubseteq d \parallel i)\}.\ d \parallel i) = \bigsqcap \{\ d.\ (c \sqsubseteq d \parallel i)\} \parallel i$
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 $c\ //\ i \sqsubseteq d \longleftrightarrow c \sqsubseteq d \parallel i$
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 $\bigsqcap \{\ d.\ (c \sqsubseteq d \parallel i)\} \sqsubseteq d$ **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 $\bigwedge f.\ ((d \sqsubseteq f \parallel i) \Longrightarrow \exists\ e.\ (c \sqsubseteq e \parallel i) \wedge (e \sqsubseteq f))$
using *c-refsto-d order.trans* **by** *blast*
then have *b*: $\bigsqcap \{\ e.\ (c \sqsubseteq e \parallel i)\} \sqsubseteq \bigsqcap \{f.\ (d \sqsubseteq f \parallel i)\}$
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 $(c \mathbin{//} j) \sqsubseteq (c \mathbin{//} i)$
proof −
have $\bigwedge f.\ ((c \sqsubseteq f \parallel i) \Longrightarrow \exists\ e.\ (c \sqsubseteq e \parallel j) \wedge (e \sqsubseteq f))$
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*: $\bigsqcap\{\ e.\ (c \sqsubseteq e \parallel j)\} \sqsubseteq \bigsqcap\{f.\ (c \sqsubseteq f \parallel i)\}$
by (*metis Inf-mono mem-Collect-eq*)
show *?thesis* **using** *rely-quotient-def b* **by** *simp*
qed

lemma *par-nonabort*:
assumes *nonabort-i*: *chaos* $\sqsubseteq i$
assumes *nonabort-j*: *chaos* $\sqsubseteq j$
shows *chaos* $\sqsubseteq i \parallel j$
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* $\sqsubseteq j$
shows $((c \mathbin{//} j) \mathbin{//} i) = c \mathbin{//} (i \parallel j)$
proof (*rule antisym*)
show $((c \mathbin{//} j) \mathbin{//} i) \sqsubseteq c \mathbin{//} (i \parallel j)$
proof −
have $\bigwedge f.\ ((c \sqsubseteq f \parallel i \parallel j) \Longrightarrow \exists\ e.\ (c \sqsubseteq e \parallel j) \wedge (e \sqsubseteq f \parallel i))$ **by** *blast*
then have $\bigsqcap\{\ d.\ (\bigsqcap\{\ e.\ (c \sqsubseteq e \parallel j)\} \sqsubseteq d \parallel i)\} \sqsubseteq \bigsqcap\{f.\ (c \sqsubseteq f \parallel i \parallel j)\}$
by (*simp add*: *Collect-mono Inf-lower Inf-superset-mono*)
thus *?thesis* **using** *local.rely-quotient-def par-assoc* **by** *auto*
qed
next
show $c \mathbin{//} (i \parallel j) \sqsubseteq ((c \mathbin{//} j) \mathbin{//} i)$
proof −
have $c \sqsubseteq \bigsqcap\{\ e.\ (c \sqsubseteq e \parallel j)\} \parallel j$
using *j-nonabort local.rely-quotient-def rely-quotient* **by** *auto*
then have $\bigwedge d.\ \bigsqcap\{\ e.\ (c \sqsubseteq e \parallel j)\} \sqsubseteq d \parallel i \Longrightarrow (c \sqsubseteq d \parallel i \parallel j)$
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*
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* $\sqsubseteq$ *i*
assumes *nonabort-j*: *chaos* $\sqsubseteq$ *j*
shows $c \Cap d \sqsubseteq (j \Cap (c \mathbin{//} i)) \parallel (i \Cap (d \mathbin{//} j))$
proof −
have *a*: $c \sqsubseteq (c \mathbin{//} i) \parallel i$ **using** *nonabort-i nonabort-j rely-quotient* **by** *auto*
have *b*: $d \sqsubseteq j \parallel (d \mathbin{//} j)$ **using** *rely-quotient par-commute*
by (*simp add*: *nonabort-j*)
have $c \Cap d \sqsubseteq ((c \mathbin{//} i) \parallel i) \Cap (j \parallel (d \mathbin{//} j))$ **using** *a b* **by** (*metis conj-mono*)
also have *interchange*: $c \Cap d \sqsubseteq ((c \mathbin{//} i) \Cap j) \parallel (i \Cap (d \mathbin{//} j))$
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* $\sqsubseteq$ *i*
shows $(c \Cap d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i) \Cap (d \mathbin{//} i)$
proof −
have $c \Cap d \sqsubseteq ((c \mathbin{//} i) \parallel i) \Cap ((d \mathbin{//} i) \parallel i)$ **using** *conj-mono rely-quotient*
by (*simp add*: *nonabort-i*)
then have $c \Cap d \sqsubseteq ((c \mathbin{//} i) \Cap (d \mathbin{//} i)) \parallel (i \Cap i)$
by (*metis parallel-interchange refine-trans*)
then have $c \Cap d \sqsubseteq ((c \mathbin{//} i) \Cap (d \mathbin{//} i)) \parallel i$ **by** (*metis conj-idem*)
thus *?thesis* **using** *rely-refinement* **by** (*simp add*: *nonabort-i*)
qed

lemma *distribute-rely-choice*:
assumes *nonabort-i*: *chaos* $\sqsubseteq$ *i*
shows $(c \sqcap d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i) \sqcap (d \mathbin{//} i)$
proof −
have $c \sqcap d \sqsubseteq ((c \mathbin{//} i) \parallel i) \sqcap ((d \mathbin{//} i) \parallel i)$
by (*metis nonabort-i inf-mono rely-quotient*)
then have $c \sqcap d \sqsubseteq ((c \mathbin{//} i) \sqcap (d \mathbin{//} i)) \parallel i$ **by** (*metis inf-par-distrib*)
thus *?thesis* **by** (*metis nonabort-i rely-refinement*)

qed

lemma *distribute-rely-parallel1*:
assumes *nonabort-i*: $chaos \sqsubseteq i$
assumes *nonabort-j*: $chaos \sqsubseteq j$
shows $(c \parallel d) \mathbin{//} (i \parallel j) \sqsubseteq (c \mathbin{//} i) \parallel (d \mathbin{//} j)$
proof −
have $(c \parallel d) \sqsubseteq ((c \mathbin{//} i) \parallel i) \parallel ((d \mathbin{//} j) \parallel j)$
using *par-mono rely-quotient nonabort-i nonabort-j* **by** *simp*
then have $(c \parallel d) \sqsubseteq (c \mathbin{//} i) \parallel (d \mathbin{//} j) \parallel j \parallel i$ **by** (*metis par-assoc par-commute*)
thus *?thesis* **using** *par-assoc par-commute rely-refinement*
by (*metis nonabort-i nonabort-j par-nonabort*)
qed

lemma *distribute-rely-parallel2*:
assumes *nonabort-i*: $chaos \sqsubseteq i$
assumes *i-par-i*: $i \parallel i \sqsubseteq i$
shows $(c \parallel d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i) \parallel (d \mathbin{//} i)$
proof −
have $(c \parallel d) \mathbin{//} i \sqsubseteq ((c \parallel d) \mathbin{//} (i \parallel i))$ **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*: $chaos \sqsubseteq i$
assumes $(\forall\ c.\ (\forall\ d.\ ((c \parallel i);(d \parallel i) \sqsubseteq (c;d) \parallel i)))$
shows $(c;d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i);(d \mathbin{//} i)$
proof −
have $c;d \sqsubseteq ((c \mathbin{//} i) \parallel i);((d \mathbin{//} i) \parallel i)$
by (*metis rely-quotient nonabort-i seq-mono*)
then have $c;d \sqsubseteq (c \mathbin{//} i)\ ;\ (d \mathbin{//} i) \parallel i$ **using** *assms*(*2*) **by** (*metis refine-trans*)
thus *?thesis* **by** (*metis rely-refinement nonabort-i*)
qed

lemma *distribute-rely-sequential-event*:
assumes *nonabort-i*: $chaos \sqsubseteq i$
assumes *nonabort-j*: $chaos \sqsubseteq j$
assumes *nonabort-e*: $chaos \sqsubseteq e$
assumes $(\forall\ c.\ (\forall\ d.\ ((c \parallel i);e;(d \parallel j) \sqsubseteq (c;e;d) \parallel (i;e;j))))$
shows $(c;e;d) \mathbin{//} (i;e;j) \sqsubseteq (c \mathbin{//} i);e;(d \mathbin{//} j)$
proof −
have $c;e;d \sqsubseteq ((c \mathbin{//} i) \parallel i);e;((d \mathbin{//} j) \parallel j)$

by (*metis order.refl rely-quotient nonabort-i nonabort-j seq-mono*)
then have $c;e;d \sqsubseteq ((c \mathbin{//} i);e;(d \mathbin{//} j)) \parallel (i;e;j)$ **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*: $chaos \sqsubseteq i$
assumes *nonabort-j0*: $chaos \sqsubseteq j_0$
assumes *nonabort-j1*: $chaos \sqsubseteq j_1$
shows $(c \Cap d) \mathbin{//} i \sqsubseteq (j_1 \Cap (c \mathbin{//} (j_0 \parallel i))) \parallel (j_0 \Cap (d \mathbin{//} (j_1 \parallel i)))$
proof −
have $(c \Cap d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i) \Cap (d \mathbin{//} i)$
by (*metis distribute-rely-conjunction nonabort-i*)
then have $(c \Cap d) \mathbin{//} i \sqsubseteq (j_1 \Cap ((c \mathbin{//} i) \mathbin{//} j_0)) \parallel (j_0 \Cap ((d \mathbin{//} i) \mathbin{//} j_1))$
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*: $chaos \sqsubseteq i$
assumes *nonabort-j0*: $chaos \sqsubseteq j_0$
assumes *nonabort-j1*: $chaos \sqsubseteq j_1$
shows $(j_1 \parallel j_0) \Cap (c \Cap d) \mathbin{//} i \sqsubseteq (j_1 \Cap (c \mathbin{//} (j_0 \parallel i))) \parallel (j_0 \Cap (d \mathbin{//} (j_1 \parallel i)))$
proof −
have $(j_1 \parallel j_0) \Cap (c \Cap d) \mathbin{//} i \sqsubseteq (j_1 \parallel j_0) \Cap ((j_1 \Cap (c \mathbin{//} (j_0 \parallel i))) \parallel (j_0 \Cap (d \mathbin{//} (j_1 \parallel i))))$
by (*metis introduce-parallel-with-rely nonabort-i nonabort-j0 nonabort-j1*
conj-mono order.refl)
also have $\ldots \sqsubseteq (j_1 \Cap j_1 \Cap (c \mathbin{//} (j_0 \parallel i))) \parallel (j_0 \Cap j_0 \Cap (d \mathbin{//} (j_1 \parallel i)))$
by (*metis conj-assoc parallel-interchange*)
finally show *?thesis* **by** (*metis conj-idem*)
qed

lemma *wrap-rely-guar*:
assumes *nonabort-rg*: $chaos \sqsubseteq rg$
and *skippable*: $rg \sqsubseteq skip$
shows $c \sqsubseteq rg \Cap c \mathbin{//} rg$
proof −
have $c = c \mathbin{//} skip$ **by** (*simp add*: *quotient-identity*)
also have $\ldots \sqsubseteq c \mathbin{//} rg$ **by** (*simp add*: *skippable weaken-rely nonabort-rg*)
also have $\ldots \sqsubseteq rg \Cap c \mathbin{//} rg$ **using** *conjoin-non-aborting conj-commute nonabort-rg*
by *auto*

finally show $c \sqsubseteq rg \Cap c \mathbin{//} rg$ **.**
qed

end

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

begin

lemma *distribute-rely-iteration*:
assumes *nonabort-i*: *chaos* $\sqsubseteq i$
assumes $(\forall\ c.\ (\forall\ d.\ ((c \parallel i);(d \parallel i) \sqsubseteq (c;d) \parallel i)))$
shows $(c^\omega;d) \mathbin{//} i \sqsubseteq (c \mathbin{//} i)^\omega;(d \mathbin{//} i)$
proof −
have $d \sqcap c\ ;\ ((c \mathbin{//} i)^\omega;(d \mathbin{//} i) \parallel i) \sqsubseteq ((d \mathbin{//} i) \parallel i) \sqcap ((c \mathbin{//} i) \parallel i);((c \mathbin{//} i)^\omega;(d \mathbin{//} i) \parallel i)$
by (*metis inf-mono order.refl rely-quotient nonabort-i seq-mono*)
also have $\ldots \sqsubseteq ((d \mathbin{//} i) \parallel i) \sqcap ((c \mathbin{//} i);(c \mathbin{//} i)^\omega;(d \mathbin{//} i) \parallel i\ )$
using *assms inf-mono-right seq-assoc* **by** *fastforce*
also have $\ldots \sqsubseteq ((d \mathbin{//} i) \sqcap (c \mathbin{//} i);(c \mathbin{//} i)^\omega;(d \mathbin{//} i)) \parallel i$
by (*simp add*: *inf-par-distrib*)
also have $\ldots = ((c \mathbin{//} i)^\omega;(d \mathbin{//} i)) \parallel i$
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 Council 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 $(\bigsqcap C) \parallel d = (\bigsqcap c \in C.c \parallel d)$ 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 $c \Cap (\bigsqcup D) = (\bigsqcup d \in D.c \Cap d)$. 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 $c \Cap \bot = \bot$.

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 $\top \sqsubseteq chaos \parallel \top$.

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 $D \neq \{\} \Rightarrow c \,;\, \bigsqcap D = (\bigsqcap d \in D.c \,;\, d)$. 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

[1] C. Aarts, R. Backhouse, E. Boiten, H. Doombos, N. van Gasteren, R. van Geldrop, P. Hoogendijk, E. Voermans, and J. van der Woude. Fixed-point calculus. *Information Processing Letters*, 53:131–136, 1995. Mathematics of Program Construction Group.

[2] R. J. Colvin, I. J. Hayes, and L. A. Meinicke. Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, pages 1–22, 2016. Accepted 28 November 2016.

[3] I. J. Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, November 2016.

[4] I. J. Hayes, R. J. Colvin, L. A. Meinicke, K. Winter, and A. Velykis. An algebra of synchronous atomic steps. In J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 352–369, Cham, November 2016. Springer International Publishing.

[5] C. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, Oct. 1983.

[6] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25.