

Concurrent Refinement Algebra and Rely Quotients

Julian Fell and Ian Hayes and Andrius Velykis

May 26, 2024

Abstract

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

Contents

1	Overview	4
2	Refinement Lattice	4
3	Sequential Operator	7
3.1	Basic sequential	7
3.2	Distributed sequential	8
4	Parallel Operator	10
4.1	Basic parallel operator	10
4.2	Distributed parallel	10
5	Weak Conjunction Operator	12
5.1	Distributed weak conjunction	13
6	Concurrent Refinement Algebra	14
7	Galois Connections and Fusion Theorems	16
7.1	Lower Galois connections	17
7.2	Greatest fixpoint fusion theorems	18
7.3	Upper Galois connections	19
7.4	Least fixpoint fusion theorems	20
8	Iteration	21
8.1	Possibly infinite iteration	21
8.2	Finite iteration	23
8.3	Infinite iteration	24
8.4	Combined iteration	25
9	Sequential composition for conjunctive models	26
10	Infimum nat lemmas	28
11	Iteration for conjunctive models	30
12	Rely Quotient Operator	35
12.1	Basic rely quotient	35
12.2	Distributed rely quotient	39
13	Conclusions	42

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 ⇒ 'a ⇒ bool (**infix** ⊑ 50)

where

$c \sqsubseteq d \equiv \text{less-eq } c \ d$

abbreviation

refine-strict :: 'a ⇒ 'a ⇒ bool (**infix** ⊐ 50)

where

$c \sqsubset d \equiv \text{less } c \ d$

Non-deterministic choice is monotonic in both arguments

lemma *inf-mono-left*: $a \sqsubseteq b \implies a \sqcap c \sqsubseteq b \sqcap c$

using *inf-mono* **by** *auto*

lemma *inf-mono-right*: $c \sqsubseteq d \implies a \sqcap c \sqsubseteq a \sqcap d$

using *inf-mono* **by** *auto*

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

lemma *Inf2-inf*: $\sqcap \{f x \mid x. x \in \{c, d\}\} = f c \sqcap f d$

proof –

have $\{f x \mid x. x \in \{c, d\}\} = \{f c, f d\}$ **by** *blast*

then have $\sqcap \{f x \mid x. x \in \{c, d\}\} = \sqcap \{f c, f d\}$ **by** *simp*

also have ... = $f c \sqcap f d$ **by** *simp*

finally show *?thesis* .

qed

Helper lemma for choice over indexed set.

lemma *INF-Inf*: $(\sqcap_{x \in X}. f x) = (\sqcap \{f x \mid x. x \in X\})$

by (*simp add: Setcompr-eq-image*)

lemma (**in** –) *INF-absorb-args*: $(\sqcap i j. (f :: \text{nat} \Rightarrow 'c :: \text{complete-lattice}) (i + j)) = (\sqcap k. f k)$

proof (*rule order-class.order.antisym*)

show $(\sqcap k. f k) \leq (\sqcap i j. f (i + j))$

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

lemma (*in -*) *nested-Collect*: $\{f y \mid y. y \in \{g x \mid x. x \in X\}\} = \{f (g x) \mid x. x \in X\}$
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*: $P \{d x \mid x. x \in X\}$

assumes *f-Inf-distrib*: $\bigwedge c D. P D \Longrightarrow f c (\bigcap D) = \bigcap \{f c d \mid d. d \in D\}$

shows $f c (\bigcap x \in X. d x) = (\bigcap x \in X. f c (d x))$

proof –

have $f c (\bigcap x \in X. d x) = f c (\bigcap \{d x \mid x. x \in X\})$ **by** (*simp add: INF-Inf*)

also have $\dots = (\bigcap \{f c dx \mid dx. dx \in \{d x \mid x. x \in X\}\})$ **by** (*simp add: qual f-Inf-distrib*)

also have $\dots = (\bigcap \{f c (d x) \mid x. x \in X\})$ **by** (*simp only: nested-Collect*)

also have $\dots = (\bigcap x \in X. f c (d x))$ **by** (*simp add: INF-Inf*)

finally show *?thesis* .

qed

lemma *Inf-distrib-INF*:

fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$

assumes *f-Inf-distrib*: $\bigwedge c D. f c (\bigcap D) = \bigcap \{f c d \mid d. d \in D\}$

shows $f c (\bigcap x \in X. d x) = (\bigcap x \in X. f c (d x))$

by (*simp add: Setcompr-eq-image f-Inf-distrib image-comp*)

end

lemmas *refine-trans = order.trans*

More transitivity rules to make calculational reasoning smoother

declare *ord-eq-le-trans*[*trans*]

declare *ord-le-eq-trans*[*trans*]

declare *dual-order.trans*[*trans*]

abbreviation

dist-over-sup :: $('a :: \text{refinement-lattice} \Rightarrow 'a) \Rightarrow \text{bool}$

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 \perp as a left annihilator.

locale *seq* =

fixes $seq :: 'a :: refinement-lattice \Rightarrow 'a \Rightarrow 'a$ (**infixl** ; 90)

assumes $seq-bot [simp]: \perp ; c = \perp$

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

Left distribution implies sequential composition is monotonic in 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*]: $c ; \perp \sqsubseteq c$
by (*metis bot.extremum seq.right-neutral seq-mono-right*)

end

locale *seq-distrib-right* = *sequential* +
assumes *Inf-seq-distrib*:
 $(\sqcap C) ; d = (\sqcap (c::'a::\text{refinement-lattice}) \in C. c ; d)$
begin

lemma *INF-seq-distrib*: $(\sqcap c \in C. f c) ; d = (\sqcap c \in C. f c ; d)$
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 $(c_0 \sqcap c_1) ; d = (\sqcap \{c_0, c_1\}) ; d$ **by** *simp*
also have $\dots = (\sqcap c \in \{c_0, c_1\}. c) ; d$ **by** (*fact Inf-seq-distrib*)
also have $\dots = (c_0 ; d) \sqcap (c_1 ; d)$ **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 $\top ; c = (\sqcap a \in \{\}. a) ; c$

by (*metis Inf-empty Inf-seq-distrib*)
thus *?thesis*
by *simp*
qed

primrec *seq-power* :: '*a* \Rightarrow *nat* \Rightarrow '*a* (**infixr** $^$ 80) **where**
seq-power-0: $a ^ 0 = \text{nil}$
| *seq-power-Suc*: $a ^ \text{Suc } n = a ; (a ^ n)$

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*: $i < j \Longrightarrow (b ^ j) = (b ^ i) ; (b ^ (j - i))$

proof (*induct j arbitrary: i type: nat*)

case 0

thus *?case by simp*

next

case (*Suc j*)

have $b ^ \text{Suc } j = b ; (b ^ i) ; (b ^ (j - i))$

using *Suc.hyps Suc.prem1 less-Suc-eq seq-assoc by auto*

also have $\dots = (b ^ i) ; b ; (b ^ (j - i))$ **by** (*simp add: seq-power-front*)

also have $\dots = (b ^ i) ; (b ^ (\text{Suc } j - i))$

using *Suc.prem1 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 || 75)
  assumes abort-par:  $\perp$  || c =  $\perp$ 
```

```
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 || ( $\bigcap$  D) = ( $\bigcap$  d $\in$ D. c || d)
```

```
begin
```

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

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

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

lemma *INF-INF-par-distrib*:

$X \neq \{\} \implies Y \neq \{\} \implies (\prod_{x \in X} c x) \parallel (\prod_{y \in Y} d y) = (\prod_{x \in X} \prod_{y \in Y} c x \parallel d y)$

proof –

assume *nonempty-X*: $X \neq \{\}$

assume *nonempty-Y*: $Y \neq \{\}$

have $(\prod_{x \in X} c x) \parallel (\prod_{y \in Y} d y) = (\prod_{x \in X} c x \parallel (\prod_{y \in Y} d y))$

using *INF-par-distrib* **by** (*metis nonempty-X*)

also have $\dots = (\prod_{x \in X} \prod_{y \in Y} c x \parallel d y)$ **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 $(c_0 \sqcap c_1) \parallel d = (\prod \{c_0, c_1\}) \parallel d$ **by** *simp*

also have $\dots = (\prod c \in \{c_0, c_1\}. c \parallel d)$ **using** *Inf-par-distrib* **by** (*meson insert-not-empty*)

also have $\dots = c_0 \parallel d \sqcap c_1 \parallel d$ **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*: $c_1 \sqsubseteq d_1 \implies c_2 \sqsubseteq d_2 \implies c_1 \parallel c_2 \sqsubseteq d_1 \parallel d_2$

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 \mathbb{m} is similar to least upper bound (\sqcup) but is abort strict, i.e. the lattice bottom is an annihilator: $c \mathbb{m} \perp = \perp$. 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  $\mathbb{m}$  80)
  assumes conj-bot-right:  $c \mathbb{m} \perp = \perp$ 
```

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]:  $\perp \mathbb{m} c = \perp$ 
using conj-bot-right local.conj-commute by fastforce
```

```
lemma conj-not-bot:  $a \mathbb{m} b \neq \perp \Longrightarrow a \neq \perp \wedge b \neq \perp$ 
using conj-bot-right by auto
```

```
lemma conj-distrib1:  $c \mathbb{m} (d_0 \mathbb{m} d_1) = (c \mathbb{m} d_0) \mathbb{m} (c \mathbb{m} d_1)$ 
by (metis conj-assoc conj-commute conj-idem)
```

```
end
```

5.1 Distributed weak conjunction

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

locale *conj-distrib* = *conjunction* +

assumes *Inf-conj-distrib*: $D \neq \{\}$ $\implies (\bigcap D) \mathbin{\&} c = (\bigcap_{d \in D} d \mathbin{\&} c)$

begin

lemma *conj-Inf-distrib*: $D \neq \{\}$ $\implies c \mathbin{\&} (\bigcap D) = (\bigcap_{d \in D} c \mathbin{\&} d)$

using *Inf-conj-distrib conj-commute* **by** *auto*

lemma *inf-conj-distrib*: $(c_0 \sqcap c_1) \mathbin{\&} d = (c_0 \mathbin{\&} d) \sqcap (c_1 \mathbin{\&} d)$

proof –

have $(c_0 \sqcap c_1) \mathbin{\&} d = (\bigcap \{c_0, c_1\}) \mathbin{\&} d$ **by** *simp*

also have $\dots = (\bigcap c \in \{c_0, c_1\}. c \mathbin{\&} d)$ **by** (*rule Inf-conj-distrib, simp*)

also have $\dots = (c_0 \mathbin{\&} d) \sqcap (c_1 \mathbin{\&} d)$ **by** *simp*

finally show *?thesis* .

qed

lemma *inf-conj-product*: $(a \sqcap b) \mathbin{\&} (c \sqcap d) = (a \mathbin{\&} c) \sqcap (a \mathbin{\&} d) \sqcap (b \mathbin{\&} c) \sqcap (b \mathbin{\&} d)$

by (*metis inf-conj-distrib conj-commute inf-assoc*)

lemma *conj-mono*: $c_0 \sqsubseteq d_0 \implies c_1 \sqsubseteq d_1 \implies c_0 \mathbin{\&} c_1 \sqsubseteq d_0 \mathbin{\&} d_1$

by (*metis inf.absorb-iff1 inf-conj-product inf-right-idem*)

lemma *conj-mono-left*: $c_0 \sqsubseteq c_1 \implies c_0 \mathbin{\&} d \sqsubseteq c_1 \mathbin{\&} d$

by (*simp add: conj-mono*)

lemma *conj-mono-right*: $c_0 \sqsubseteq c_1 \implies d \mathbin{\&} c_0 \sqsubseteq d \mathbin{\&} c_1$

by (*simp add: conj-mono*)

lemma *conj-refine*: $c_0 \sqsubseteq d \implies c_1 \sqsubseteq d \implies c_0 \mathbin{\&} c_1 \sqsubseteq d$

by (*metis conj-idem conj-mono*)

lemma *refine-to-conj*: $c \sqsubseteq d_0 \implies c \sqsubseteq d_1 \implies c \sqsubseteq d_0 \mathbin{\&} d_1$

by (*metis conj-idem conj-mono*)

lemma *conjoin-non-aborting*: *chaos* $\sqsubseteq c \implies d \sqsubseteq d \mathbin{\&} c$

by (*metis conj-mono order.refl conj-chaos*)

lemma *conjunction-sup*: $c \mathbin{\&} d \sqsubseteq c \sqcup d$

by (*simp add: conj-refine*)

lemma *conjunction-sup-nonaborting*:
assumes $chaos \sqsubseteq c$ **and** $chaos \sqsubseteq d$
shows $c \sqcap d = c \sqcup d$
proof (*rule antisym*)
show $c \sqcup d \sqsubseteq c \sqcap d$ **using** *assms(1) assms(2) 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*: $chaos \sqsubseteq c \implies c \sqcap \top = \top$
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*: $\top \sqsubseteq \text{chaos} \parallel \top$
 assumes *chaos-par-chaos*: $\text{chaos} \sqsubseteq \text{chaos} \parallel \text{chaos}$
 assumes *parallel-interchange*: $(c_0 \parallel c_1) \pitchfork (d_0 \parallel d_1) \sqsubseteq (c_0 \pitchfork d_0) \parallel (c_1 \pitchfork d_1)$
begin

lemma *chaos-skip*: $\text{chaos} \sqsubseteq \text{skip}$

proof –

have $\text{chaos} = (\text{chaos} \parallel \text{skip}) \pitchfork (\text{skip} \parallel \text{chaos})$ **by** *simp*
 then have $\dots \sqsubseteq (\text{chaos} \pitchfork \text{skip}) \parallel (\text{skip} \pitchfork \text{chaos})$ **using** *parallel-interchange* **by** *blast*
 thus ?thesis **by** *auto*

qed

lemma *chaos-par-chaos-eq*: $\text{chaos} = \text{chaos} \parallel \text{chaos}$

by (*metis antisym chaos-par-chaos chaos-skip order-refl par-mono par-skip*)

lemma *nonabort-par-top*: $\text{chaos} \sqsubseteq c \implies c \parallel \top = \top$

by (*metis chaos-par-top par-mono top.extremum-uniqueI*)

lemma *skip-conj-top*: $\text{skip} \pitchfork \top = \top$

by (*simp add: chaos-skip conjoin-top*)

lemma *conj-distrib2*: $c \sqsubseteq c \parallel c \implies c \pitchfork (d_0 \parallel d_1) \sqsubseteq (c \pitchfork d_0) \parallel (c \pitchfork d_1)$

proof –

assume $c \sqsubseteq c \parallel c$
 then have $c \pitchfork (d_0 \parallel d_1) \sqsubseteq (c \parallel c) \pitchfork (d_0 \parallel d_1)$ **by** (*metis conj-mono order.refl*)
 thus ?thesis **by** (*metis parallel-interchange refine-trans*)

qed

end

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

locale *conjunction-sequential* = *conj-distrib* + *seq-distrib* +
 assumes *chaos-seq-chaos*: $\text{chaos} \sqsubseteq \text{chaos};\text{chaos}$
 assumes *sequential-interchange*: $(c_0;c_1) \pitchfork (d_0;d_1) \sqsubseteq (c_0 \pitchfork d_0);(c_1 \pitchfork d_1)$
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:  $c;\perp \sqcap d \sqsubseteq (c \sqcap d);\perp$ 
  by (metis (no-types) conj-bot-left seq-nil-right sequential-interchange)

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

lemma conj-distrib3:  $c \sqsubseteq c;c \implies c \sqcap (d_0 ; d_1) \sqsubseteq (c \sqcap d_0);(c \sqcap d_1)$ 
proof –
  assume  $c \sqsubseteq c;c$ 
  then have  $c \sqcap (d_0;d_1) \sqsubseteq (c;c) \sqcap (d_0;d_1)$  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-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*]: $\{F x \mid x. x = a \vee x = b\} = \{F a, F b\}$
by *auto*

locale *lower-galois-connections*
begin

definition

l-adjoint :: (*'a*::*refinement-lattice* \Rightarrow *'a*) \Rightarrow (*'a* \Rightarrow *'a*) (^b [201] 200)

where

(*F*^b) *x* $\equiv \sqcap \{y. x \sqsubseteq F y\}$

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 $\dots = F x \sqcap F y$

proof –

from *distF*

have $F (\sqcap \{x, y\}) = \sqcap \{F x, F y\}$ **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* $\Longrightarrow x \sqsubseteq (F \circ F^b) x$

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** $X\text{-def } Y\text{-def CollectD singletonI}$ **by** *auto*

then have $\sqcap X \sqsubseteq \sqcap Y$ **by** (*simp add: Inf-mono*)

then have $x \sqsubseteq \sqcap \{F y \mid y. x \sqsubseteq F y\}$ **by** (*simp add: X-def Y-def*)

then have $x \sqsubseteq F (\sqcap \{y. x \sqsubseteq F y\})$ **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 $\implies ((F^b) x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq F y)$*

proof

assume $x \sqsubseteq F y$

then have $\sqcap \{y. x \sqsubseteq F y\} \sqsubseteq y$ **by** (*simp add: Inf-lower*)

thus $(F^b) x \sqsubseteq y$ **by** (*metis l-adjoint-def*)

next

assume *dist: dist-over-inf F* **then have** *monoF: mono F* **by** (*simp add: dist-inf-mono*)

assume $(F^b) x \sqsubseteq y$ **then have** $a: F ((F^b) x) \sqsubseteq F y$ **by** (*simp add: monoD monoF*)

have $x \sqsubseteq F ((F^b) x)$ **using** *dist l-cancellation* **by** *simp*

thus $x \sqsubseteq F y$ **using** a **by** *auto*

qed

lemma *v-simple-fusion: mono G $\implies \forall x. ((F \circ G) x \sqsubseteq (H \circ F) x) \implies F (\text{gfp } G) \sqsubseteq \text{gfp } H$*

by (*metis comp-eq-dest-lhs gfp-unfold gfp-upperbound*)

7.2 Greatest fixpoint fusion theorems

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

theorem *fusion-gfp-geq:*

assumes *monoH: mono H*

and *distribF: dist-over-inf F*

and *comp-geq: $\bigwedge x. ((H \circ F) x \sqsubseteq (F \circ G) x)$*

shows $\text{gfp } H \sqsubseteq F (\text{gfp } G)$

proof –

have $(\text{gfp } H) \sqsubseteq (F \circ F^b) (\text{gfp } H)$ **using** *distribF l-cancellation* **by** *simp*

then have $H (\text{gfp } H) \sqsubseteq H ((F \circ F^b) (\text{gfp } H))$ **by** (*simp add: monoD monoH*)

then have $H (\text{gfp } H) \sqsubseteq F ((G \circ F^b) (\text{gfp } H))$ **using** *comp-geq* **by** (*metis comp-def refine-trans*)

then have $(F^b) (H (\text{gfp } H)) \sqsubseteq (G \circ F^b) (\text{gfp } H)$ **using** *distribF* **by** (*metis (mono-tags) l-galois-connection*)

then have $(F^b) (\text{gfp } H) \sqsubseteq (\text{gfp } G)$ **by** (*metis comp-apply gfp-unfold gfp-upperbound monoH*)

thus $\text{gfp } H \sqsubseteq F (\text{gfp } G)$ **using** $\text{distrib}F$ **by** $(\text{metis } (\text{mono-tags}) \text{ l-galois-connection})$
qed

theorem fusion-gfp-eq :

assumes $\text{mono}H$: $\text{mono } H$ **and** $\text{mono}G$: $\text{mono } G$

and $\text{dist}F$: $\text{dist-over-inf } F$

and fgh-comp : $\bigwedge x. ((F \circ G) x = (H \circ F) x)$

shows $F (\text{gfp } G) = \text{gfp } H$

proof (rule antisym)

show $F (\text{gfp } G) \sqsubseteq (\text{gfp } H)$ **by** $(\text{metis } \text{fgh-comp } \text{le-less } \text{v-simple-fusion } \text{mono}G)$

next

have $\bigwedge x. ((H \circ F) x \sqsubseteq (F \circ G) x)$ **using** fgh-comp **by** auto

then show $\text{gfp } H \sqsubseteq F (\text{gfp } G)$ **using** $\text{mono}H$ $\text{dist}F$ fusion-gfp-geq **by** blast

qed

end

7.3 Upper Galois connections

locale $\text{upper-galois-connections}$

begin

definition

$\text{u-adjoint} :: ('a::\text{refinement-lattice} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) (-\# [201] 200)$

where

$(F\#) x \equiv \bigsqcup \{y. F y \sqsubseteq x\}$

lemma dist-sup-mono :

assumes $\text{dist}F$: $\text{dist-over-sup } F$

shows $\text{mono } F$

proof

fix $x :: 'a$ **and** $y :: 'a$

assume $x \sqsubseteq y$

then have $F y = F (x \sqcup y)$ **by** $(\text{simp add: } \text{le-iff-sup})$

also have $\dots = F x \sqcup F y$

proof $-$

from $\text{dist}F$

have $F (\bigsqcup \{x, y\}) = \bigsqcup \{F x, F y\}$ **by** $(\text{drule-tac } x = \{x, y\} \text{ in spec, simp})$

then show $F (x \sqcup y) = F x \sqcup F y$ **by** simp

qed

finally show $F x \sqsubseteq F y$ **by** $(\text{metis } \text{le-iff-sup})$

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 $\sqcup Y \sqsubseteq \sqcup X$ **by** (*simp add: Sup-mono*)

then have $\sqcup \{F y \mid y. F y \sqsubseteq x\} \sqsubseteq x$ **by** (*simp add: X-def Y-def*)

then have $F (\sqcup \{y. F y \sqsubseteq x\}) \sqsubseteq x$ **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* $\implies (F x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq (F^\#) y)$

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 $x \sqsubseteq \sqcup \{x. F x \sqsubseteq y\}$ **by** (*simp add: Sup-upper*)

thus $x \sqsubseteq (F^\#) y$ **by** (*metis u-adjoint-def*)

qed

lemma *u-simple-fusion: mono H* $\implies \forall x. ((F \circ G) x \sqsubseteq (G \circ H) x) \implies \text{lfp } F \sqsubseteq G (\text{lfp } H)$

by (*metis comp-def lfp-lowerbound lfp-unfold*)

7.4 Least fixpoint fusion theorems

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

theorem *fusion-lfp-leq:*

assumes *monoH: mono H*

and *distribF: dist-over-sup F*

and *comp-leq: $\bigwedge x. ((F \circ G) x \sqsubseteq (H \circ F) x)$*

shows $F (\text{lfp } G) \sqsubseteq (\text{lfp } H)$

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 $(G \circ F^\#) (lfp\ H) \sqsubseteq (F^\#) (H (lfp\ H))$ **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 :: $'a \Rightarrow 'a$ ($-\omega$ [103] 102)

where

$c^\omega \equiv lfp (\lambda x. nil \sqcap c;x)$

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

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

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

lemma iter-induct-nil: $\text{nil} \sqcap c; x \sqsubseteq x \implies c^\omega \sqsubseteq x$
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 $c^\omega; c^\omega \sqsubseteq c^\omega$ **using** *iter0 seq-mono-right* **by** *fastforce*
next
have $a: \text{nil} \sqcap c; c^\omega; c^\omega \sqsubseteq \text{nil} \sqcap c; c^\omega \sqcap c; c^\omega; c^\omega$
by (*metis inf-greatest inf-le2 inf-mono iter0 order-refl seq-distrib-left.seq-mono-right seq-distrib-left-axioms seq-nil-right*)
then have $b: \dots = c^\omega \sqcap c; c^\omega; c^\omega$ **using** *iter-unfold* **by** *auto*
then have $c: \dots = (\text{nil} \sqcap c; c^\omega); c^\omega$ **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: $\perp = \text{nil}^\omega$
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

fiter :: 'a \Rightarrow 'a (-* [101] 100)

where

$c^* \equiv \text{gfp } (\lambda x. \text{nil} \sqcap c;x)$

lemma *fin-iter-step-mono*: $\text{mono } (\lambda x. \text{nil} \sqcap c;x)$

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*: $c^* = \text{nil} \sqcap c;c^*$

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

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

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

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

by (*metis fiter-unfold inf.cobounded1*)

lemma *fiter1*: $c^* \sqsubseteq c$

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

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

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 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*: $x \sqsubseteq d \sqcap c;x \implies x \sqsubseteq c^*;d$

proof –

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

qed

lemma *fiter2* [*simp*]: $c^*;c^* = c^*$

proof –

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

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 –

assume $c \sqsubseteq d$
then have $c^* \sqsubseteq nil \sqcap d;c^*$ **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

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

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

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

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

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

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

lemma *infiter-induct*: $c;x \sqsubseteq x \Longrightarrow c^\infty \sqsubseteq x$

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

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

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 $\forall a. (\perp :: 'a) \sqsubseteq a$

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

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

lemma infiter-inf-top:  $c^\infty = c^\omega ; \top$ 
proof –
  have lr:  $c^\infty \sqsubseteq c^\omega ; \top$ 
  proof –
    have c ;  $(c^\omega ; \top) = \text{nil} ; \top \sqcap c ; c^\omega ; \top$ 
    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.inf-seq-distrib seq-distrib-right-axioms)
  qed
  have rl:  $c^\omega ; \top \sqsubseteq c^\infty$ 
  by (metis inf-le2 infiter-annil infiter-unfold iter-induct-nil seq-mono-left)
  thus ?thesis using antisym-conv lr by blast
qed

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

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

end

end

```

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*: $D \neq \{\} \implies c; \sqcap D = (\sqcap d \in D. c; d)$

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 $c; \sqcap \{d_0, d_1\} = \sqcap \{c; d \mid d. d \in \{d_0, d_1\}\}$ **using** *seq-Inf-distrib*

proof –

have $\sqcap ((: c \text{ ' } \{d_0, d_1\})) = \sqcap \{c; a \mid a. a \in \{d_0, d_1\}\}$

using *INF-Inf* **by** *blast*

then show *?thesis*

using $\langle \wedge (c::'a::\text{refinement-lattice}) D::'a::\text{refinement-lattice set. } D \neq \{\} \implies c; \sqcap D = (\sqcap d::'a::\text{refinement-lattice} \in D. c; d) \rangle \langle \{d_0::'a::\text{refinement-lattice}, d_1::'a::\text{refinement-lattice}\} \neq \{\} \rangle$ **by** *presburger*

qed

also have $\dots = c; d_0 \sqcap c; d_1$ **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*: $X \neq \{\} \implies c; (\sqcap x \in X. d x) = (\sqcap x \in X. c; d x)$

proof –

assume $xne: X \neq \{\}$

have $a: c; (\sqcap x \in X. d x) = c; \sqcap (d \text{ ' } X)$ **by** *auto*

also have $b: \dots = (\sqcap d \in (d \text{ ' } X). c; d)$ **by** (*meson image-is-empty seq-Inf-distrib xne*)

also have $c: \dots = (\sqcap x \in X. c; d x)$ **by** (*simp add: image-comp*)

finally show *?thesis* **by** (*simp add: b image-comp*)

qed

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

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

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

end

end

10 Infimum nat lemmas

theory *Infimum-Nat*

imports

Refinement-Lattice

begin

locale *infimum-nat*

begin

lemma *INF-partition-nat3*:

fixes $f :: nat \Rightarrow nat \Rightarrow 'a::refinement-lattice$

shows $(\prod j. f i j) =$

$(\prod j \in \{j. i = j\}. f i j) \sqcap$

$(\prod j \in \{j. i < j\}. f i j) \sqcap$

$(\prod j \in \{j. j < i\}. f i j)$

proof –

have *univ-part*: $UNIV = \{j. i = j\} \cup \{j. i < j\} \cup \{j. j < i\}$ **by** *auto*

have $(\prod j \in \{j. i = j\} \cup \{j. i < j\} \cup \{j. j < i\}. f i j) =$

$(\prod j \in \{j. i = j\}. f i j) \sqcap$

$(\prod j \in \{j. i < j\}. f i j) \sqcap$

$(\prod j \in \{j. j < i\}. f i j)$ **by** (*metis INF-union*)

with *univ-part* **show** *?thesis* **by** *simp*

qed

lemma *INF-INF-partition-nat3*:

fixes $f :: nat \Rightarrow nat \Rightarrow 'a::refinement-lattice$

shows $(\prod i. \prod j. f i j) =$
 $(\prod i. \prod j \in \{j. i = j\}. f i j) \sqcap$
 $(\prod i. \prod j \in \{j. i < j\}. f i j) \sqcap$
 $(\prod i. \prod j \in \{j. j < i\}. f i j)$

proof –

have $(\prod i. \prod j. f i j) = (\prod i. ((\prod j \in \{j. i = j\}. f i j) \sqcap$
 $(\prod j \in \{j. i < j\}. f i j) \sqcap$
 $(\prod j \in \{j. j < i\}. f i j)))$

by (*simp add: INF-partition-nat3*)

also have $\dots = (\prod i. \prod j \in \{j. i = j\}. f i j) \sqcap$
 $(\prod i. \prod j \in \{j. i < j\}. f i j) \sqcap$
 $(\prod i. \prod j \in \{j. j < i\}. f i j)$

by (*simp add: INF-inf-distrib*)

finally show *?thesis* .

qed

lemma *INF-nat-shift*: $(\prod i \in \{i. 0 < i\}. f i) = (\prod i. f (Suc i))$

by (*metis greaterThan-0 greaterThan-def range-composition*)

lemma *INF-nat-minus*:

fixes $f :: nat \Rightarrow 'a::refinement-lattice$

shows $(\prod j \in \{j. i < j\}. f (j - i)) = (\prod k \in \{k. 0 < k\}. f k)$

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 $(\prod i. \prod j \in \{j. j < i\}. f j (i - j)) = (\prod j. \prod i \in \{i. j < i\}. f j (i - j))$

proof (*rule antisym*)

have $\bigwedge j \ ii. jj < ii \implies \exists i. \exists j < i. f j (i - j) \sqsubseteq f jj (ii - jj)$

by *blast*

then have $\bigwedge j \ ii. jj < ii \implies \exists i. (\prod j \in \{j. j < i\}. f j (i - j)) \sqsubseteq f jj (ii - jj)$

by (*meson INF-lower mem-Collect-eq*)

then have $\bigwedge j \ ii. jj < ii \implies (\prod i. \prod j \in \{j. j < i\}. f j (i - j)) \sqsubseteq f jj (ii - jj)$

by (*meson UNIV-I INF-lower dual-order.trans*)

then have $\bigwedge j. (\prod i. \prod j \in \{j. j < i\}. f j (i - j)) \sqsubseteq (\prod ii \in \{ii. jj < ii\}. f jj (ii - jj))$

by (*metis (mono-tags, lifting) INF-greatest mem-Collect-eq*)

then have $(\prod i. \prod j \in \{j. j < i\}. f j (i - j)) \sqsubseteq (\prod jj. \prod ii \in \{ii. jj < ii\}. f jj (ii - jj))$

by (*simp add: INF-greatest*)

```

then show  $(\prod i. \prod j \in \{j. j < i\}. fj (i - j)) \sqsubseteq (\prod j. \prod i \in \{i. j < i\}. fj (i - j))$ 
  by simp
next
have  $\bigwedge ii\ jj. jj < ii \implies \exists j. \exists i > j. fj (i - j) \sqsubseteq fjj (ii - jj)$ 
  by blast
then have  $\bigwedge ii\ jj. jj < ii \implies \exists j. (\prod i \in \{i. j < i\}. fj (i - j)) \sqsubseteq fjj (ii - jj)$ 
  by (meson INF-lower mem-Collect-eq)
then have  $\bigwedge ii\ jj. jj < ii \implies (\prod j. \prod i \in \{i. j < i\}. fj (i - j)) \sqsubseteq fjj (ii - jj)$ 
  by (meson UNIV-I INF-lower dual-order.trans)
then have  $\bigwedge ii. (\prod j. \prod i \in \{i. j < i\}. fj (i - j)) \sqsubseteq (\prod jj \in \{jj. jj < ii\}. fjj (ii - jj))$ 
  by (metis (mono-tags, lifting) INF-greatest mem-Collect-eq)
then have  $(\prod j. \prod i \in \{i. j < i\}. fj (i - j)) \sqsubseteq (\prod ii. \prod jj \in \{jj. jj < ii\}. fjj (ii - jj))$ 
  by (simp add: INF-greatest)
then show  $(\prod j. \prod i \in \{i. j < i\}. fj (i - j)) \sqsubseteq (\prod i. \prod j \in \{j. j < i\}. fj (i - j))$ 
  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*: $c^\omega = c^* \sqcap c^\infty$

proof —

define *F* **where** $F = (\lambda x. c^* \sqcap x)$

define *G* **where** $G = (\lambda x. c;x)$

define *H* **where** $H = (\lambda x. nil \sqcap c;x)$

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

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

have $adjoint: dist\text{-over-sup } F$ **by** $(simp\ add: F\text{-def } inf\text{-Sup})$

have $monoH: mono\ H$ **by** $(metis\ H\text{-def } inf\text{-mono } monoI\ order\ refl\ seq\ mono\ right)$

have $monoG: mono\ G$ **by** $(metis\ G\text{-def } inf.\text{absorb-iff2 } monoI\ seq\ inf\text{-distrib})$

have $\forall x. ((F \circ G) x = (H \circ F) x)$ **using** $FG\ HF$

by $(metis\ fiter\ unfold\ inf\text{-sup-aci}(2)\ seq\ inf\text{-distrib})$

then have $F (lfp\ G) = lfp\ H$ **using** $adjoint\ monoH\ monoG\ fusion\ lfp\ eq$ **by** $blast$

then have $c^* \sqcap lfp (\lambda x. c; x) = lfp (\lambda x. nil \sqcap c; x)$

using $F\text{-def } G\text{-def } H\text{-def}$ **by** $blast$

thus $?thesis$ **by** $(simp\ add: infiter\text{-def } iter\text{-def})$

qed

lemma $iter\text{-induct-isolate}: c^*; d \sqcap c^\infty = lfp (\lambda x. d \sqcap c; x)$

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** $F\text{-def } G\text{-def}$ **by** $auto$

have $HF: H \circ F = (\lambda x. d \sqcap c; c^*; d \sqcap c; x)$ **using** $F\text{-def } H\text{-def } weak\text{-seq}\text{-inf}\text{-distrib}$

by $(metis\ comp\ apply\ inf.\text{commute } inf.\text{left-commute } seq\text{-assoc } seq\text{-inf}\text{-distrib})$

have $unroll: c^*; d = (nil \sqcap c; c^*); d$ **using** $fiter\text{-unfold}$ **by** $auto$

have $distribute: c^*; d = d \sqcap c; c^*; d$ **by** $(simp\ add: unroll\ inf\text{-seq}\text{-distrib})$

have $FGx: (F \circ G) x = d \sqcap c; c^*; d \sqcap c; x$ **using** $FG\ distribute$ **by** $simp$

have $adjoint: dist\text{-over-sup } F$ **by** $(simp\ add: F\text{-def } inf\text{-Sup})$

have $monoH: mono\ H$ **by** $(metis\ H\text{-def } inf\text{-mono } monoI\ order\ refl\ seq\ mono\ right)$

have $monoG: mono\ G$ **by** $(metis\ G\text{-def } inf.\text{absorb-iff2 } monoI\ seq\ inf\text{-distrib})$

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 $c^*; d \sqcap lfp (\lambda x. c; x) = lfp (\lambda x. d \sqcap c; x)$

using $F\text{-def } G\text{-def } H\text{-def}$ **by** $blast$

thus $?thesis$ **by** $(simp\ add: infiter\text{-def})$

qed

lemma $iter\text{-induct-eq}: c^\omega; d = lfp (\lambda x. d \sqcap c; x)$

proof –

have $c^\omega; d = c^*; d \sqcap c^\infty; d$ **by** $(simp\ add: isolation\ inf\text{-seq}\text{-distrib})$

then have $c^*; d \sqcap c^\infty; d = c^*; d \sqcap c^\infty$ **by** $(simp\ add: infiter\text{-annil})$

then have $c^*; d \sqcap c^\infty = lfp (\lambda x. d \sqcap c; x)$ **by** $(simp\ add: iter\text{-induct-isolate})$

thus *?thesis*

by (*simp add: <c^ω ; d = c^{*} ; d ⊓ c[∞] ; d> <c^{*} ; d ⊓ c[∞] ; d = c^{*} ; d ⊓ c[∞]>*)

qed

lemma *iter-induct*: $d \sqcap c; x \sqsubseteq x \implies c^\omega; d \sqsubseteq x$

by (*simp add: iter-induct-eq lfp-lowerbound*)

lemma *iter-isolate*: $c^*; d \sqcap c^\infty = c^\omega; d$

by (*simp add: iter-induct-eq iter-induct-isolate*)

lemma *iter-isolate2*: $c; c^*; d \sqcap c^\infty = c; c^\omega; d$

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*: $c;(d;c)^* = (c;d)^*;c$

proof –

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

proof –

have $(d;c)^* = nil \sqcap d;c;(d;c)^*$

by (*meson finite-iteration.fiter-unfold finite-iteration-axioms*)

then show *?thesis*

by (*metis fiter-induct seq-assoc seq-distrib-left.weak-seq-inf-distrib seq-distrib-left-axioms seq-nil-right*)

qed

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

proof –

have *a1*: $(c;d)^*;c = c \sqcap c;d;(c;d)^*;c$

by (*metis finite-iteration.fiter-unfold finite-iteration-axioms inf-seq-distrib seq-nil-left*)

have *a2*: $(c;d)^*;c \sqsubseteq c;(d;c)^* \longleftrightarrow c \sqcap c;d;(c;d)^*;c \sqsubseteq c;(d;c)^*$ **by** (*simp add: a1*)

then have *a3*: $\dots \longleftrightarrow c;(nil \sqcap d;(c;d)^*;c) \sqsubseteq c;(d;c)^*$

by (*metis a1 eq-iff fiter-unfold lr seq-assoc seq-inf-distrib seq-nil-right*)

have *a4*: $(nil \sqcap d;(c;d)^*;c) \sqsubseteq (d;c)^* \implies c;(nil \sqcap d;(c;d)^*;c) \sqsubseteq c;(d;c)^*$

using *seq-mono-right* **by** *blast*

have *a5*: $(nil \sqcap d;(c;d)^*;c) \sqsubseteq (d;c)^*$

proof –

have *f1*: $d;(c;d)^*;c \sqcap nil = d;((c;d)^*;c) \sqcap nil \sqcap nil$

by (*simp add: seq-assoc*)

have $d;c;(d;(c;d)^*;c \sqcap nil) = d;((c;d)^*;c)$

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

lemma fiter-seq-choice-nonempty:  $c ; c^* = (\prod i \in \{i. 0 < i\}. c \text{ :}^{\wedge} i)$ 
proof –
  have  $(\prod i \in \{i. 0 < i\}. c \text{ :}^{\wedge} i) = (\prod i. c \text{ :}^{\wedge} (\text{Suc } i))$  by (simp add: INF-nat-shift)
  also have  $\dots = (\prod i. c ; (c \text{ :}^{\wedge} i))$  by simp

```

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

end

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

begin

lemma *conj-distrib4*: $c^* \pitchfork d^* \sqsubseteq (c \pitchfork d)^*$

proof –

have $c^* \pitchfork d^* = (nil \sqcap (c; c^*)) \pitchfork d^*$ **by** (*metis fiter-unfold*)

then have $c^* \pitchfork d^* = (nil \pitchfork d^*) \sqcap ((c; c^*) \pitchfork d^*)$ **by** (*simp add: inf-conj-distrib*)

then have $c^* \pitchfork d^* \sqsubseteq nil \sqcap ((c; c^*) \pitchfork (d; d^*))$ **by** (*metis conj-idem fiter0 fiter-unfold inf.bounded-iff inf-le2 local.conj-mono*)

then have $c^* \pitchfork d^* \sqsubseteq nil \sqcap ((c \pitchfork d); (c^* \pitchfork d^*))$ **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

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 \Rightarrow 'a \Rightarrow 'a (**infixl** '//' 85)

where

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

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

lemma *quotient-identity*: $c // \text{skip} = c$

proof –

have $c // \text{skip} = \sqcap \{ d. (c \sqsubseteq d \parallel \text{skip}) \}$ **by** (*metis rely-quotient-def*)

then have $c // \text{skip} = \sqcap \{ d. (c \sqsubseteq d) \}$ **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 $(c // \perp) \parallel \perp$ equals \perp , it does not refine c in general.

theorem *rely-quotient*:

assumes *nonabort-i*: $\text{chaos} \sqsubseteq i$

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

proof –

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

define C **where** $C = \{ c \}$

have $(\forall d \in D. (\exists c \in C. c \sqsubseteq d))$ **using** *D-def C-def CollectD singletonI* **by** *auto*

then have $\sqcap C \sqsubseteq (\sqcap D)$ **by** (*simp add: Inf-mono*)

then have $c \sqsubseteq \sqcap \{ d \parallel i \mid d. (c \sqsubseteq d \parallel i) \}$ **by** (*simp add: C-def D-def*)

also have $\dots = \sqcap \{ d \parallel i \mid d. d \in \{ d. (c \sqsubseteq d \parallel i) \} \}$ **by** *simp*

also have $\dots = (\sqcap d \in \{ d. (c \sqsubseteq d \parallel i) \}. d \parallel i)$ **by** (*simp add: INF-Inf*)

also have $\dots = \sqcap \{ d \mid d. (c \sqsubseteq d \parallel i) \} \parallel i$

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

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

then show $(\sqcap d \in \{ d. (c \sqsubseteq d \parallel i) \}. d \parallel i) = \sqcap \{ d \mid d. (c \sqsubseteq d \parallel i) \} \parallel i$

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 $(\prod d \in \{d. (c \sqsubseteq d \parallel i)\}. d \parallel i) = \prod \{d. (c \sqsubseteq d \parallel i)\} \parallel i$
using $Inf\text{-}par\text{-}distrib$ **by** $simp$
then show $?thesis$ **by** $auto$
qed
also have $\dots = (c // i) \parallel i$ **by** $(metis\ rely\text{-}quotient\text{-}def)$
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 $rely\text{-}refinement$:

assumes $nonabort\text{-}i: chaos \sqsubseteq i$
shows $c // i \sqsubseteq d \iff c \sqsubseteq d \parallel i$

proof

assume $a: c // i \sqsubseteq d$
have $c \sqsubseteq (c // i) \parallel i$ **using** $rely\text{-}quotient\ nonabort\text{-}i$ **by** $simp$
thus $c \sqsubseteq d \parallel i$ **using** $par\text{-}mono\ a$
by $(metis\ inf.\text{absorb}\text{-}iff2\ inf\text{-}commute\ le\text{-}infI2\ order\text{-}refl)$

next

assume $b: c \sqsubseteq d \parallel i$
then have $\prod \{d. (c \sqsubseteq d \parallel i)\} \sqsubseteq d$ **by** $(simp\ add: Inf\text{-}lower)$
thus $c // i \sqsubseteq d$ **by** $(metis\ rely\text{-}quotient\text{-}def)$

qed

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

lemma $rely\text{-}mono$:

assumes $c\text{-}refsto\text{-}d: c \sqsubseteq d$
shows $(c // i) \sqsubseteq (d // i)$

proof –

have $\bigwedge f. ((d \sqsubseteq f \parallel i) \implies \exists e. (c \sqsubseteq e \parallel i) \wedge (e \sqsubseteq f))$
using $c\text{-}refsto\text{-}d\ order.\text{trans}$ **by** $blast$
then have $b: \prod \{e. (c \sqsubseteq e \parallel i)\} \sqsubseteq \prod \{f. (d \sqsubseteq f \parallel i)\}$
by $(metis\ Inf\text{-}mono\ mem\text{-}Collect\text{-}eq)$
show $?thesis$ **using** $rely\text{-}quotient\text{-}def\ b$ **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 $weaken\text{-}rely$:

assumes $i\text{-}refsto\text{-}j: i \sqsubseteq j$

shows $(c // j) \sqsubseteq (c // i)$
proof –
have $\bigwedge f. ((c \sqsubseteq f // i) \implies \exists e. (c \sqsubseteq e // j) \wedge (e \sqsubseteq f))$
using *i-refsto-j order.trans*
by (*metis inf.absorb-iff2 inf-le1 inf-par-distrib inf-sup-ord(2) par-commute*)
then have $b: \prod \{ e. (c \sqsubseteq e // j) \} \sqsubseteq \prod \{ f. (c \sqsubseteq f // i) \}$
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* $\sqsubseteq i // j$
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 $((c // j) // i) = c // (i // j)$
proof (*rule antisym*)
show $((c // j) // i) \sqsubseteq c // (i // j)$
proof –
have $\bigwedge f. ((c \sqsubseteq f // i // j) \implies \exists e. (c \sqsubseteq e // j) \wedge (e \sqsubseteq f // i))$ **by** *blast*
then have $\prod \{ d. (\prod \{ e. (c \sqsubseteq e // j) \} \sqsubseteq d // i) \} \sqsubseteq \prod \{ f. (c \sqsubseteq f // i // j) \}$
by (*simp add: Collect-mono Inf-lower Inf-superset-mono*)
thus *?thesis using local.rely-quotient-def par-assoc by auto*
qed

next

show $c // (i // j) \sqsubseteq ((c // j) // i)$
proof –
have $c \sqsubseteq \prod \{ e. (c \sqsubseteq e // j) \} // j$
using *j-nonabort local.rely-quotient-def rely-quotient by auto*
then have $\bigwedge d. \prod \{ e. (c \sqsubseteq e // j) \} \sqsubseteq d // i \implies (c \sqsubseteq d // i // j)$
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

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*: $\text{chaos} \sqsubseteq i$
assumes *nonabort-j*: $\text{chaos} \sqsubseteq j$
shows $c \text{ m } d \sqsubseteq (j \text{ m } (c // i)) \parallel (i \text{ m } (d // j))$

proof –

have $a: c \sqsubseteq (c // i) \parallel i$ **using** *nonabort-i nonabort-j rely-quotient* **by** *auto*
have $b: d \sqsubseteq j \parallel (d // j)$ **using** *rely-quotient par-commute*
by (*simp add: nonabort-j*)
have $c \text{ m } d \sqsubseteq ((c // i) \parallel i) \text{ m } (j \parallel (d // j))$ **using** $a \ b$ **by** (*metis conj-mono*)
also have *interchange*: $c \text{ m } d \sqsubseteq ((c // i) \text{ m } j) \parallel (i \text{ m } (d // j))$
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*: $\text{chaos} \sqsubseteq i$
shows $(c \text{ m } d) // i \sqsubseteq (c // i) \text{ m } (d // i)$

proof –

have $c \text{ m } d \sqsubseteq ((c // i) \parallel i) \text{ m } ((d // i) \parallel i)$ **using** *conj-mono rely-quotient*
by (*simp add: nonabort-i*)
then have $c \text{ m } d \sqsubseteq ((c // i) \text{ m } (d // i)) \parallel (i \text{ m } i)$
by (*metis parallel-interchange refine-trans*)
then have $c \text{ m } d \sqsubseteq ((c // i) \text{ m } (d // i)) \parallel i$ **by** (*metis conj-idem*)
thus *?thesis* **using** *rely-refinement* **by** (*simp add: nonabort-i*)

qed

lemma *distribute-rely-choice*:

assumes *nonabort-i*: $\text{chaos} \sqsubseteq i$
shows $(c \sqcap d) // i \sqsubseteq (c // i) \sqcap (d // i)$

proof –

have $c \sqcap d \sqsubseteq ((c // i) \parallel i) \sqcap ((d // i) \parallel i)$
by (*metis nonabort-i inf-mono rely-quotient*)
then have $c \sqcap d \sqsubseteq ((c // i) \sqcap (d // i)) \parallel i$ **by** (*metis inf-par-distrib*)
thus *?thesis* **by** (*metis nonabort-i rely-refinement*)

qed

lemma *distribute-rely-parallel1*:

assumes *nonabort-i*: $\text{chaos} \sqsubseteq i$

assumes *nonabort-j*: $\text{chaos} \sqsubseteq j$

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

proof –

have $(c \parallel d) \sqsubseteq ((c // i) \parallel i) \parallel ((d // j) \parallel j)$

using *par-mono rely-quotient nonabort-i nonabort-j* **by** *simp*

then have $(c \parallel d) \sqsubseteq (c // i) \parallel (d // j) \parallel j \parallel i$ **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*: $\text{chaos} \sqsubseteq i$

assumes *i-par-i*: $i \parallel i \sqsubseteq i$

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

proof –

have $(c \parallel d) // i \sqsubseteq ((c \parallel d) // (i \parallel i))$ **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*: $\text{chaos} \sqsubseteq i$

assumes $(\forall c. (\forall d. ((c \parallel i);(d \parallel i) \sqsubseteq (c;d) \parallel i)))$

shows $(c;d) // i \sqsubseteq (c // i);(d // i)$

proof –

have $c;d \sqsubseteq ((c // i) \parallel i);((d // i) \parallel i)$

by (*metis rely-quotient nonabort-i seq-mono*)

then have $c;d \sqsubseteq (c // i);(d // i) \parallel i$ **using** *assms(2)* **by** (*metis refine-trans*)

thus *?thesis* **by** (*metis rely-refinement nonabort-i*)

qed

lemma *distribute-rely-sequential-event*:

assumes *nonabort-i*: $\text{chaos} \sqsubseteq i$

assumes *nonabort-j*: $\text{chaos} \sqsubseteq j$

assumes *nonabort-e*: $\text{chaos} \sqsubseteq e$

assumes $(\forall c. (\forall d. ((c \parallel i);e;(d \parallel j) \sqsubseteq (c;e;d) \parallel (i;e;j))))$

shows $(c;e;d) // (i;e;j) \sqsubseteq (c // i);e;(d // j)$

proof –

have $c;e;d \sqsubseteq ((c // i) \parallel i);e;((d // j) \parallel j)$

by (*metis order.refl rely-quotient nonabort-i nonabort-j seq-mono*)
then have $c;e;d \sqsubseteq ((c // i);e;(d // j)) \parallel (i;e;j)$ **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 $(c \pitchfork d) // i \sqsubseteq (j_1 \pitchfork (c // (j_0 \parallel i))) \parallel (j_0 \pitchfork (d // (j_1 \parallel i)))$
proof –
have $(c \pitchfork d) // i \sqsubseteq (c // i) \pitchfork (d // i)$
by (*metis distribute-rely-conjunction nonabort-i*)
then have $(c \pitchfork d) // i \sqsubseteq (j_1 \pitchfork ((c // i) // j_0)) \parallel (j_0 \pitchfork ((d // i) // j_1))$
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 $(j_1 \parallel j_0) \pitchfork (c \pitchfork d) // i \sqsubseteq (j_1 \pitchfork (c // (j_0 \parallel i))) \parallel (j_0 \pitchfork (d // (j_1 \parallel i)))$
proof –
have $(j_1 \parallel j_0) \pitchfork (c \pitchfork d) // i \sqsubseteq (j_1 \parallel j_0) \pitchfork ((j_1 \pitchfork (c // (j_0 \parallel i))) \parallel (j_0 \pitchfork (d // (j_1 \parallel i))))$
by (*metis introduce-parallel-with-rely nonabort-i nonabort-j0 nonabort-j1 conj-mono order.refl*)
also have $\dots \sqsubseteq (j_1 \pitchfork j_1 \pitchfork (c // (j_0 \parallel i))) \parallel (j_0 \pitchfork j_0 \pitchfork (d // (j_1 \parallel i)))$
by (*metis conj-assoc parallel-interchange*)
finally show *?thesis* **by** (*metis conj-idem*)
qed

lemma *wrap-rely-guar:*

assumes *nonabort-rg: chaos* $\sqsubseteq rg$
and *skippable: rg* $\sqsubseteq skip$
shows $c \sqsubseteq rg \pitchfork c // rg$
proof –
have $c = c // skip$ **by** (*simp add: quotient-identity*)
also have $\dots \sqsubseteq c // rg$ **by** (*simp add: skippable weaken-rely nonabort-rg*)
also have $\dots \sqsubseteq rg \pitchfork c // rg$ **using** *conjoin-non-aborting conj-commute nonabort-rg*
by *auto*

finally show $c \sqsubseteq rg \sqcap c // rg$.

qed

end

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

begin

lemma *distribute-rely-iteration*:

assumes *nonabort-i*: $chaos \sqsubseteq i$

assumes $(\forall c. (\forall d. ((c // i);(d // i) \sqsubseteq (c;d // i)))$

shows $(c^\omega;d) // i \sqsubseteq (c // i)^\omega;(d // i)$

proof –

have $d \sqcap c ; ((c // i)^\omega;(d // i) // i) \sqsubseteq ((d // i) // i) \sqcap ((c // i) // i);((c // i)^\omega;(d // i) // i)$

by (*metis inf-mono order.refl rely-quotient nonabort-i seq-mono*)

also have $\dots \sqsubseteq ((d // i) // i) \sqcap ((c // i);(c // i)^\omega;(d // i) // i)$

using *assms inf-mono-right seq-assoc* **by** *fastforce*

also have $\dots \sqsubseteq ((d // i) \sqcap (c // i);(c // i)^\omega;(d // i) // i)$

by (*simp add: inf-par-distrib*)

also have $\dots = ((c // i)^\omega;(d // i) // i)$

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 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 $(\sqcap C) \parallel d = (\sqcap c \in C. c \parallel d)$ but in Section 4 we assume this only for non-empty C and furthermore assume that parallel is abort strict, i.e. $\perp \parallel c = c$.
3. The earlier paper assumes $c \pitchfork (\sqcup D) = (\sqcup d \in D. c \pitchfork d)$. 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 $c \pitchfork \perp = \perp$.
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 $D \neq \{\} \Rightarrow c ; \sqcap D = (\sqcap d \in D. c ; d)$. 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.