

Given Clause Loops

Jasmin Blanchette

Qi Qiu

Sophie Tourret

April 18, 2024

Abstract

This Isabelle/HOL formalization extends the `Saturation_Framework` and `Saturation_Framework_Extensions` entries of the *Archive of Formal Proofs* with the specification and verification of four semiabstract given clause procedures, or “loops”: the DISCOUNT, Otter, iProver, and Zipperposition loops. For each loop, (dynamic) refutational completeness is proved under the assumption that the underlying calculus is (statically) refutationally complete and that the used queue data structures are fair.

The formalization is inspired by the proof sketches found in the article “A comprehensive framework for saturation theorem proving” by Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette (*Journal of Automated Reasoning* **66**(4): 499–539, 2022). A paper titled “Verified given clause procedures” about the present formalization is in the works.

Contents

1	Utilities for Given Clause Loops	1
2	More Lemmas about Given Clause Architectures	5
2.1	Inference System	5
2.2	Given Clause Procedure Basis	5
2.3	Given Clause Procedure	7
2.4	Lazy Given Clause Procedure	9
3	DISCOUNT Loop	10
3.1	Locale	11
3.2	Basic Definitions and Lemmas	12
3.3	Refinement	13
3.4	Completeness	17
4	Prover Queues and Fairness	18
4.1	Basic Lemmas	18
4.2	More on Relational Chains over Lazy Lists	18
4.3	Locales	19
4.4	Instantiation with FIFO Queue	20
5	Fair DISCOUNT Loop	25
5.1	Locale	25
5.2	Basic Definitions and Lemmas	26
5.3	Initial State and Invariant	29
5.4	Final State	30

5.5	Refinement	32
5.6	Completeness	33
5.7	Specialization with FIFO Queue	41
6	Otter Loop	42
6.1	Basic Definitions and Lemmas	43
6.2	Refinement	44
6.3	Completeness	50
7	Definition of Fair Otter Loop	51
7.1	Locale	51
7.2	Basic Definitions and Lemmas	52
7.3	Initial State and Invariant	53
7.4	Final State	54
7.5	Refinement	56
8	iProver Loop	58
8.1	Definition	58
8.2	Refinement	58
8.3	Completeness	59
9	Fair iProver Loop	60
9.1	Locale	60
9.2	Basic Definition	60
9.3	Initial State and Invariant	60
9.4	Final State	61
9.5	Refinement	61
9.6	Completeness	62
10	Completeness of Fair Otter Loop	76
10.1	Completeness	76
10.2	Specialization with FIFO Queue	76
11	Zipperposition Loop with Ghost State	77
11.1	Basic Definitions and Lemmas	78
11.2	Refinement	78
11.3	Completeness	83
12	Prover Lazy List Queues and Fairness	83
12.1	Basic Lemmas	83
12.2	Locales	84
12.3	Instantiation with FIFO Queue	87
13	Fair Zipperposition Loop with Ghosts	97
13.1	Locale	98
13.2	Basic Definitions and Lemmas	98
13.3	Initial State and Invariant	100
13.4	Final State	102
13.5	Refinement	104
13.6	Completeness	106

14 Fair Zipperposition Loop without Ghosts	117
14.1 Locale	117
14.2 Basic Definitions and Lemmas	118
14.3 Initial States and Invariants	119
14.4 Abstract Nonsense for Ghost–Ghostless Conversion	119
14.5 Ghost–Ghostless Conversions, the Concrete Version	121
14.6 Completeness	125
14.7 Specialization with FIFO Queue	127
15 Given Clause Loops	128

1 Utilities for Given Clause Loops

This section contains various lemmas used by the rest of the formalization of given clause procedures.

```

theory Given-Clause-Loops-Util
imports
  HOL-Library.FSet
  HOL-Library.Multiset
  Ordered-Resolution-Prover.Lazy-List-Chain
  Weighted-Path-Order.Multiset-Extension-Pair
  Lambda-Free-RPOs.Lambda-Free-Util
begin

hide-const (open) Seq.chain

hide-fact (open) Abstract-Rewriting.chain-mono

declare fset-of-list.rep-eq [simp]

instance bool :: wellorder
proof
  fix P and b :: bool
  assume  $(\bigwedge y. y < b \implies P y) \implies P b$  for b :: bool
  hence  $\bigwedge q. q \leq b \implies P q$ 
    using less-bool-def by presburger
  then show P b
    by auto
qed

lemma finite-imp-set-eq:
  assumes fin: finite A
  shows  $\exists xs. \text{set } xs = A$ 
  using fin
proof (induct A rule: finite-induct)
  case empty
  then show ?case
    by auto
next
  case (insert x B)
  then obtain xs :: 'a list where
    set xs = B
  by blast

```

then have $set (x \# xs) = insert\ x\ B$
by *auto*
then show *?case*
by *blast*
qed

lemma *Union-Setcompr-member-mset-mono*:
assumes $sub: P \subseteq\# Q$
shows $\bigcup \{f\ x \mid x. x \in\# P\} \subseteq \bigcup \{f\ x \mid x. x \in\# Q\}$
proof –
have $\{f\ x \mid x. x \in\# P\} \subseteq \{f\ x \mid x. x \in\# Q\}$
by (*rule Collect-mono*) (*metis sub mset-subset-eqD*)
thus *?thesis*
by (*simp add: Sup-subset-mono*)
qed

lemma *singletons-in-mult1*: $(x, y) \in R \implies (\{x\}, \{y\}) \in mult1\ R$
by (*metis add-mset-add-single insert-DiffM mult1I single-eq-add-mset*)

lemma *singletons-in-mult*: $(x, y) \in R \implies (\{x\}, \{y\}) \in mult\ R$
by (*simp add: mult-def singletons-in-mult1 trancl.intros(1)*)

lemma *multiset-union-diff-assoc*:
fixes $A\ B\ C :: 'a\ multiset$
assumes $A \cap\# C = \{\#\}$
shows $A + B - C = A + (B - C)$
by (*metis assms multiset-union-diff-commute union-commute*)

lemma *Liminf-llist-subset*:
assumes
 $llength\ Xs = llength\ Ys$ **and**
 $\forall i < llength\ Xs. lnth\ Xs\ i \subseteq lnth\ Ys\ i$
shows $Liminf\text{-}llist\ Xs \subseteq Liminf\text{-}llist\ Ys$
unfolding *Liminf-llist-def* **using** *assms*
by (*smt INT-iff SUP-mono mem-Collect-eq subsetD subsetI*)

lemma *countable-imp-lset*:
assumes $count: countable\ A$
shows $\exists as. lset\ as = A$
proof (*cases finite A*)
case *fin: True*
have $\exists as. set\ as = A$
by (*simp add: fin finite-imp-set-eq*)
thus *?thesis*
by (*meson lset-llist-of*)
next
case *inf: False*

let $?as = inf\text{-}llist\ (from\text{-}nat\text{-}into\ A)$

have $lset\ ?as = A$
by (*simp add: inf infinite-imp-nonempty count*)
thus *?thesis*
by *blast*
qed

lemma *distinct-imp-notin-set-drop-Suc*:

assumes

distinct xs

i < length xs

xs ! i = x

shows $x \notin \text{set } (\text{drop } (\text{Suc } i) \text{ } xs)$

by (*metis Cons-nth-drop-Suc assms distinct.simps(2) distinct-drop*)

lemma *distinct-set-drop-removeAll-hd*:

assumes

distinct xs

xs \neq []

shows $\text{set } (\text{drop } n \text{ } (\text{removeAll } (\text{hd } xs) \text{ } xs)) = \text{set } (\text{drop } (\text{Suc } n) \text{ } xs)$

using *assms*

by (*metis distinct.simps(2) drop-Suc list.exhaust-sel removeAll.simps(2) removeAll-id*)

lemma *set-drop-removeAll*: $\text{set } (\text{drop } n \text{ } (\text{removeAll } y \text{ } xs)) \subseteq \text{set } (\text{drop } n \text{ } xs)$

proof (*induct n arbitrary: xs*)

case 0

then show *?case*

by *auto*

next

case (*Suc n*)

then show *?case*

proof (*cases xs*)

case *Nil*

then show *?thesis*

by *auto*

next

case (*Cons x xs'*)

then show *?thesis*

by (*metis Suc Suc-n-not-le-n drop-Suc-Cons nat-le-linear removeAll.simps(2)*)

set-drop-subset-set-drop subset-code(1))

qed

qed

lemma *set-drop-fold-removeAll*: $\text{set } (\text{drop } k \text{ } (\text{fold } \text{removeAll } ys \text{ } xs)) \subseteq \text{set } (\text{drop } k \text{ } xs)$

proof (*induct ys arbitrary: xs*)

case (*Cons y ys*)

note *ih = this(1)*

have $\text{set } (\text{drop } k \text{ } (\text{fold } \text{removeAll } ys \text{ } (\text{removeAll } y \text{ } xs))) \subseteq \text{set } (\text{drop } k \text{ } (\text{removeAll } y \text{ } xs))$

using *ih[of removeAll y xs]* .

also have $\dots \subseteq \text{set } (\text{drop } k \text{ } xs)$

by (*meson set-drop-removeAll*)

finally show *?case*

by *simp*

qed *simp*

lemma *set-drop-append-subseteq*: $\text{set } (\text{drop } n \text{ } (xs @ ys)) \subseteq \text{set } (\text{drop } n \text{ } xs) \cup \text{set } ys$

by (*metis drop-append set-append set-drop-subset sup.idem sup.orderI sup-mono*)

lemma *distinct-fold-removeAll*:

assumes *dist: distinct xs*

```

shows distinct (fold removeAll ys xs)
using dist
proof (induct ys arbitrary: xs)
  case Nil
  then show ?case
    using dist by simp
next
  case (Cons y ys)
  note ih = this(1) and dist-xs = this(2)

  have dist-ys: distinct (removeAll y xs)
    using dist-xs by (simp add: distinct-removeAll)

  show ?case
    by simp (rule ih[OF dist-ys])
qed

lemma set-drop-append-cons: set (drop n (xs @ ys)) ⊆ set (drop n (xs @ y # ys))
proof (induct n arbitrary: xs)
  case 0
  then show ?case
    by auto
next
  case (Suc n)
  note ih = this(1)

  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis
      using set-drop-subset-set-drop[of n Suc n] by force
  next
    case (Cons x xs')
    note xs = this(1)

    have set (drop n (xs' @ ys)) ⊆ set (drop n (xs' @ y # ys))
      using ih .
    thus ?thesis
      unfolding xs by auto
  qed
qed

lemma chain-ltl: chain R sts ⇒ ¬ lnull (ltl sts) ⇒ chain R (ltl sts)
  by (metis chain.simps eq-LConsD lnull-def)

end

```

2 More Lemmas about Given Clause Architectures

This section proves lemmas about Tourret's formalization of the abstract given clause procedures *GC* and *LGC*.

```

theory More-Given-Clause-Architectures
  imports Saturation-Framework.Given-Clause-Architectures
begin

```

2.1 Inference System

context *inference-system*
begin

lemma *Inf-from-empty*: $\text{Inf-from } \{\} = \{\iota \in \text{Inf}. \text{prems-of } \iota = \{\}\}$
using *Inf-from-def* **by** *auto*

end

2.2 Given Clause Procedure Basis

context *given-clause-basis*
begin

lemma *no-labels-entails-mono-left*: $M \subseteq N \implies M \models_{\text{NG}} P \implies N \models_{\text{NG}} P$
using *no-labels.entails-trans* *no-labels.subset-entailed* **by** *blast*

lemma *no-labels-Red-F-imp-Red-F*:
assumes $C \in \text{no-labels.Red-F } (\text{fst } \mathcal{N})$
shows $(C, l) \in \text{Red-F } \mathcal{N}$

proof –

let $?N = \text{fst } \mathcal{N}$

have *c-in-red-f-g-q*: $\forall q \in Q. C \in \text{no-labels.Red-F-G-q } q ?N$

using *no-labels.Red-F-def* **assms** **by** *auto*

moreover **have** *redfgq-eq-redfeq*:

$\forall q \in Q. \text{no-labels.Red-F-G-q } q ?N = \text{no-labels.Red-F-G-empty-q } q ?N$

using *no-labels.Red-F-G-empty-q-def* *no-labels.Red-F-G-q-def* **by** *auto*

ultimately **have** $\forall q \in Q. C \in \text{no-labels.Red-F-G-empty-q } q ?N$

by *simp*

then **have** $\forall q \in Q. \mathcal{G}\text{-F-q } q C \subseteq \text{Red-F-q } q (\text{no-labels.G-Fset-q } q ?N)$

using *redfgq-eq-redfeq* *no-labels.Red-F-G-q-def* **by** *auto*

moreover **have** $\forall q \in Q. \mathcal{G}\text{-F-L-q } q (C, l) = \mathcal{G}\text{-F-q } q C$

by *simp*

moreover **have** $\forall q \in Q. \text{no-labels.G-Fset-q } q ?N = \mathcal{G}\text{-Fset-q } q \mathcal{N}$

by *auto*

ultimately **have** $\forall q \in Q. \mathcal{G}\text{-F-L-q } q (C, l) \subseteq \text{Red-F-q } q (\mathcal{G}\text{-Fset-L-q } q \mathcal{N})$

by *auto*

then **have** $\forall q \in Q. (C, l) \in \text{Red-F-G-q } q \mathcal{N}$

using *c-in-red-f-g-q* *Red-F-G-q-def* **by** *force*

then **show** $(C, l) \in \text{Red-F } \mathcal{N}$

using *Red-F-def* **by** *simp*

qed

lemma *succ-F-imp-Red-F*:

assumes

$C' \in \text{fst } \mathcal{N}$ **and**

$C' \prec \cdot C$

shows $(C, l) \in \text{Red-F } \mathcal{N}$

proof –

have $\exists l'. (C', l') \in \mathcal{N}$

using *assms* **by** *auto*

then **obtain** l' **where**

c'-l'-in: $(C', l') \in \mathcal{N}$

by *auto*

then **have** *c'-l'-ls-c-l*: $(C', l') \sqsubset (C, l)$

using *assms Prec-FL-def* **by simp**
 moreover have *g-f-q-included*: $\forall q \in Q. \mathcal{G}\text{-F-q } q \ C \subseteq \mathcal{G}\text{-F-q } q \ C'$
 using *assms prec-F-grounding* **by simp**
 ultimately have $\forall q \in Q. \mathcal{G}\text{-F-L-q } q \ (C, l) \subseteq \mathcal{G}\text{-F-L-q } q \ (C, l)$
 by *auto*
 then have $\forall q \in Q. (C, l) \in \text{Red-F-}\mathcal{G}\text{-q } q \ \mathcal{N}$
 using *c'-l'-in c'-l'-ls-c-l g-f-q-included Red-F-}\mathcal{G}\text{-q-def* **by fastforce**
 thus $(C, l) \in \text{Red-F } \mathcal{N}$
 using *Red-F-def* **by auto**
qed

lemma succ-L-imp-Red-F:

assumes
 $(C', l') \in \mathcal{N}$ **and**
 $C' \preceq C$ **and**
 $l' \sqsubset_L l$

shows $(C, l) \in \text{Red-F } \mathcal{N}$

proof –

have *c'-l'-ls-c-l*: $(C', l') \sqsubset (C, l)$
 using *Prec-FL-def assms* **by auto**

have *c'-le-c*: $C' \preceq C$

using *assms* **by simp**

then show $(C, l) \in \text{Red-F } \mathcal{N}$

proof

assume *c'-ls-c*: $C' \prec C$

have $C' \in \text{fst } \mathcal{N}$

by (*metis assms(1) eq-fst-iff rev-image-eqI*)

then show *?thesis*

using *c'-ls-c succ-F-imp-Red-F* **by blast**

next

assume *c'-eq-c*: $C' \doteq C$

have *c-eq-c'*: $C \doteq C'$

using *c'-eq-c equiv-equiv-F equivp-symp* **by force**

have $\forall q \in Q. \mathcal{G}\text{-F-q } q \ C' = \mathcal{G}\text{-F-q } q \ C$

using *c'-eq-c c-eq-c' equiv-F-grounding subset-antisym* **by auto**

then have $\forall q \in Q. \mathcal{G}\text{-F-L-q } q \ (C, l) = \mathcal{G}\text{-F-L-q } q \ (C', l')$ **by auto**

then have $\forall q \in Q. (C, l) \in \text{Red-F-}\mathcal{G}\text{-q } q \ \mathcal{N}$

using *assms(1) c'-l'-ls-c-l Red-F-}\mathcal{G}\text{-q-def* **by auto**

then show *?thesis*

using *Red-F-def* **by auto**

qed

qed

lemma prj-fl-set-to-f-set-distr-union [*simp*]: $\text{fst } \mathcal{C} \ (M \cup N) = \text{fst } \mathcal{C} \ M \cup \text{fst } \mathcal{C} \ N$
 by (*rule Set.image-Un*)

lemma prj-labeledN-eq-N [*simp*]: $\text{fst } \mathcal{C} \ \{(C, l) \mid C. C \in N\} = N$

proof –

let $?N = \{(C, l) \mid C. C \in N\}$

have $\text{fst } \mathcal{C} \ ?N = N$

proof

show $\text{fst } \mathcal{C} \ ?N \subseteq N$

by *fastforce*

next

show $\text{fst } \mathcal{C} \ ?N \supseteq N$


```

proof
  fix  $x$ 
  assume  $x \in N$ 
  then have  $(x, l) \in ?\mathcal{N}$ 
    by auto
  then show  $x \in \text{fst}' \ ?\mathcal{N}$ 
    by force
  qed
qed
then show  $\text{fst}' \ ?\mathcal{N} = N$ 
  by simp
qed

end

```

2.3 Given Clause Procedure

```

context given-clause
begin

```

```

lemma remove-redundant:
  assumes  $(C, l) \in \text{Red-F } \mathcal{N}$ 
  shows  $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$ 
proof –
  have  $\{(C, l)\} \subseteq \text{Red-F } (\mathcal{N} \cup \{\})$ 
    using assms by simp
  moreover have active-subset  $\{\} = \{\}$ 
    using active-subset-def by simp
  ultimately show  $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$ 
    by (metis process sup-bot-right)
qed

```

```

lemma remove-redundant-no-label:
  assumes  $C \in \text{no-labels.Red-F } (\text{fst}' \ \mathcal{N})$ 
  shows  $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$ 
proof –
  have  $(C, l) \in \text{Red-F } \mathcal{N}$ 
    using no-labels.Red-F-imp-Red-F assms by simp
  then show ?thesis
    using remove-redundant by auto
qed

```

```

lemma add-inactive:
  assumes  $l \neq \text{active}$ 
  shows  $\mathcal{N} \rightsquigarrow_{GC} \mathcal{N} \cup \{(C, l)\}$ 
proof –
  have active-subset-C-l: active-subset  $\{(C, l)\} = \{\}$ 
    using active-subset-def assms by simp
  also have  $\{\} \subseteq \text{Red-F } (\mathcal{N} \cup \{(C, l)\})$ 
    by simp
  finally show  $\mathcal{N} \rightsquigarrow_{GC} \mathcal{N} \cup \{(C, l)\}$ 
    by (metis active-subset-C-l process sup-bot.right-neutral)
qed

```

```

lemma remove-succ-F:
  assumes

```

$(C', l') \in \mathcal{N}$ and
 $C' \prec \cdot C$
shows $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$
proof –
have $C' \in \text{fst } \cdot \mathcal{N}$
by (*metis assms(1) fst-conv rev-image-eqI*)
then have $\{(C, l)\} \subseteq \text{Red-F } (\mathcal{N})$
using *assms succ-F-imp-Red-F* **by** *auto*
then show *?thesis*
using *remove-redundant* **by** *simp*
qed

lemma *remove-succ-L*:
assumes
 $(C', l') \in \mathcal{N}$ and
 $C' \preceq \cdot C$ and
 $l' \sqsubset_L l$
shows $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$
proof –
have $(C, l) \in \text{Red-F } \mathcal{N}$
using *assms succ-L-imp-Red-F* **by** *auto*
then show $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N}$
using *remove-redundant* **by** *auto*
qed

lemma *relabel-inactive*:
assumes
 $l' \sqsubset_L l$ and
 $l' \neq \text{active}$
shows $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N} \cup \{(C, l')\}$
proof –
have *active-subset-c-l'*: *active-subset* $\{(C, l')\} = \{\}$
using *active-subset-def assms* **by** *auto*

have $C \doteq C$
by (*simp add: equiv-equiv-F equivp-reflp*)
moreover have $(C, l') \in \mathcal{N} \cup \{(C, l')\}$
by *auto*
ultimately have $(C, l) \in \text{Red-F } (\mathcal{N} \cup \{(C, l')\})$
using *assms succ-L-imp-Red-F[of - - $\mathcal{N} \cup \{(C, l')\}$]* **by** *auto*
then have $\{(C, l)\} \subseteq \text{Red-F } (\mathcal{N} \cup \{(C, l')\})$
by *auto*

then show $\mathcal{N} \cup \{(C, l)\} \rightsquigarrow_{GC} \mathcal{N} \cup \{(C, l')\}$
using *active-subset-c-l' process[of - - $\{(C, l)\} - \{(C, l')\}$]* **by** *auto*
qed

end

2.4 Lazy Given Clause Procedure

context *lazy-given-clause*
begin

lemma *remove-redundant*:
assumes $(C, l) \in \text{Red-F } \mathcal{N}$

shows $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
proof –
have $\{(C, l)\} \subseteq Red-F \mathcal{N}$
using *assms* **by** *simp*
moreover have *active-subset* $\{\} = \{\}$
using *active-subset-def* **by** *simp*
ultimately show $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
by (*metis process sup-bot-right*)
qed

lemma *remove-redundant-no-label*:
assumes $C \in no-labels.Red-F (fst \text{ ' } \mathcal{N})$
shows $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
proof –
have $(C, l) \in Red-F \mathcal{N}$
using *no-labels-Red-F-imp-Red-F assms* **by** *simp*
then show $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
using *remove-redundant* **by** *auto*
qed

lemma *add-inactive*:
assumes $l \neq active$
shows $(T, \mathcal{N}) \rightsquigarrow LGC (T, \mathcal{N} \cup \{(C, l)\})$
proof –
have *active-subset-C-l*: *active-subset* $\{(C, l)\} = \{\}$
using *active-subset-def assms* **by** *simp*
also have $\{\} \subseteq Red-F (\mathcal{N} \cup \{(C, l)\})$
by *simp*
finally show $(T, \mathcal{N}) \rightsquigarrow LGC (T, \mathcal{N} \cup \{(C, l)\})$
by (*metis active-subset-C-l process sup-bot.right-neutral*)
qed

lemma *remove-succ-F*:
assumes
 $(C', l') \in \mathcal{N}$ **and**
 $C' \prec \cdot C$
shows $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
proof –
have $C' \in fst \text{ ' } \mathcal{N}$
by (*metis assms(1) fst-conv rev-image-eqI*)
then have $\{(C, l)\} \subseteq Red-F (\mathcal{N})$
using *assms succ-F-imp-Red-F* **by** *auto*
then show *?thesis*
using *remove-redundant* **by** *simp*
qed

lemma *remove-succ-L*:
assumes
 $(C', l') \in \mathcal{N}$ **and**
 $C' \preceq \cdot C$ **and**
 $l' \sqsubset_L l$
shows $(T, \mathcal{N} \cup \{(C, l)\}) \rightsquigarrow LGC (T, \mathcal{N})$
proof –
have $(C, l) \in Red-F \mathcal{N}$
using *assms succ-L-imp-Red-F* **by** *auto*

```

then show  $(T, \mathcal{N} \cup \{(C, l)\}) \sim LGC (T, \mathcal{N})$ 
  using remove-redundant by auto
qed

lemma relabel-inactive:
  assumes
     $l' \sqsubseteq_L l$  and
     $l' \neq \text{active}$ 
  shows  $(T, \mathcal{N} \cup \{(C, l)\}) \sim LGC (T, \mathcal{N} \cup \{(C, l')\})$ 
proof -
  have active-subset-c-l': active-subset  $\{(C, l')\} = \{\}$ 
    using active-subset-def assms by auto

  have  $C \doteq C$ 
    by (simp add: equiv-equiv-F equivp-reflp)
  moreover have  $(C, l') \in \mathcal{N} \cup \{(C, l')\}$ 
    by auto
  ultimately have  $(C, l) \in \text{Red-F } (\mathcal{N} \cup \{(C, l')\})$ 
    using assms succ-L-imp-Red-F[of - -  $\mathcal{N} \cup \{(C, l')\}$ ] by auto
  then have  $\{(C, l)\} \subseteq \text{Red-F } (\mathcal{N} \cup \{(C, l')\})$ 
    by auto

  then show  $(T, \mathcal{N} \cup \{(C, l)\}) \sim LGC (T, \mathcal{N} \cup \{(C, l')\})$ 
    using active-subset-c-l' process[of - -  $\{(C, l)\} - \{(C, l')\}$ ] by auto
qed

end

end

```

3 DISCOUNT Loop

The DISCOUNT loop is one of the two best-known given clause procedures. It is formalized as an instance of the abstract procedure *LGC*.

```

theory DISCOUNT-Loop
  imports
    Given-Clause-Loops-Util
    More-Given-Clause-Architectures
begin

```

3.1 Locale

```

datatype DL-label =
  Passive | YY | Active

```

```

primrec nat-of-DL-label :: DL-label  $\Rightarrow$  nat where
  nat-of-DL-label Passive = 2
| nat-of-DL-label YY = 1
| nat-of-DL-label Active = 0

```

```

definition DL-Prec-L :: DL-label  $\Rightarrow$  DL-label  $\Rightarrow$  bool (infix  $\sqsubseteq_L$  50) where
  DL-Prec-L  $l\ l' \iff \text{nat-of-DL-label } l < \text{nat-of-DL-label } l'$ 

```

```

locale discount-loop = labeled-lifting-intersection Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q

```

```

Red-F-q  $\mathcal{G}$ -F-q  $\mathcal{G}$ -I-q
{ $\iota_{FL} :: ('f \times 'l)$  inference. Infer (map fst (prems-of  $\iota_{FL}$ )) (fst (concl-of  $\iota_{FL}$ ))  $\in$  Inf-F}
for
  Bot-F :: 'f set
  and Inf-F :: 'f inference set
  and Bot-G :: 'g set
  and Q :: 'q set
  and entails-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set  $\Rightarrow$  bool
  and Inf-G-q :: 'q  $\Rightarrow$  'g inference set
  and Red-I-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g inference set
  and Red-F-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set
  and  $\mathcal{G}$ -F-q :: 'q  $\Rightarrow$  'f  $\Rightarrow$  'g set
  and  $\mathcal{G}$ -I-q :: 'q  $\Rightarrow$  'f inference  $\Rightarrow$  'g inference set option
+ fixes
  Equiv-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix  $\doteq$  50) and
  Prec-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix  $\prec$  50)
assumes
  equiv-equiv-F: equivp ( $\doteq$ ) and
  wf-prec-F: minimal-element ( $\prec$ ) UNIV and
  compat-equiv-prec:  $C1 \doteq D1 \Longrightarrow C2 \doteq D2 \Longrightarrow C1 \prec C2 \Longrightarrow D1 \prec D2$  and
  equiv-F-grounding:  $q \in Q \Longrightarrow C1 \doteq C2 \Longrightarrow \mathcal{G}\text{-F-q } q \ C1 \subseteq \mathcal{G}\text{-F-q } q \ C2$  and
  prec-F-grounding:  $q \in Q \Longrightarrow C2 \prec C1 \Longrightarrow \mathcal{G}\text{-F-q } q \ C1 \subseteq \mathcal{G}\text{-F-q } q \ C2$  and
  static-ref-comp: statically-complete-calculus Bot-F Inf-F ( $\models \cap \mathcal{G}$ )
  no-labels.Red-I- $\mathcal{G}$  no-labels.Red-F- $\mathcal{G}$ -empty and
  inf-have-prems:  $\iota F \in \text{Inf-F} \Longrightarrow \text{prems-of } \iota F \neq []$ 
begin

lemma po-on-DL-Prec-L: po-on ( $\sqsubset L$ ) UNIV
  by (metis (mono-tags, lifting) DL-Prec-L-def irreflp-onI less-imp-neq order.strict-trans po-on-def
      transp-onI)

lemma wfp-on-DL-Prec-L: wfp-on ( $\sqsubset L$ ) UNIV
  unfolding wfp-on-UNIV DL-Prec-L-def by (simp add: wfP-app)

lemma Active-minimal:  $l2 \neq \text{Active} \Longrightarrow \text{Active} \sqsubset L \ l2$ 
  by (cases l2) (auto simp: DL-Prec-L-def)

lemma at-least-two-labels:  $\exists l2. \text{Active} \sqsubset L \ l2$ 
  using Active-minimal by blast

sublocale lgc?: lazy-given-clause Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q  $\mathcal{G}$ -F-q  $\mathcal{G}$ -I-q
  Equiv-F Prec-F DL-Prec-L Active
  apply unfold-locales
    apply simp
    apply simp
    apply (rule equiv-equiv-F)
    apply (simp add: minimal-element.po wf-prec-F)
    using minimal-element.wf wf-prec-F apply blast
    apply (rule po-on-DL-Prec-L)
    apply (rule wfp-on-DL-Prec-L)
    apply (fact compat-equiv-prec)
    apply (fact equiv-F-grounding)
    apply (fact prec-F-grounding)
    apply (fact Active-minimal)
    apply (rule at-least-two-labels)

```

using *static-ref-comp* *statically-complete-calculus*.*statically-complete* **apply** *fastforce*
done

notation *lgc.step* (**infix** \rightsquigarrow *LGC 50*)

3.2 Basic Definitions and Lemmas

abbreviation *c-dot-succ* $:: 'f \Rightarrow 'f \Rightarrow \text{bool}$ (**infix** $\cdot \succ$ 50) **where**

$C \cdot \succ C' \equiv C' \prec \cdot C$

abbreviation *sqsupset* $:: DL\text{-label} \Rightarrow DL\text{-label} \Rightarrow \text{bool}$ (**infix** $\sqsupset L$ 50) **where**

$l \sqsupset L l' \equiv l' \sqsubset L l$

fun *labeled-formulas-of* $:: 'f \text{ set} \times 'f \text{ set} \times 'f \text{ set} \Rightarrow ('f \times DL\text{-label}) \text{ set}$ **where**

$\text{labeled-formulas-of } (P, Y, A) = \{(C, \text{Passive}) \mid C. C \in P\} \cup \{(C, YY) \mid C. C \in Y\} \cup \{(C, \text{Active}) \mid C. C \in A\}$

lemma *labeled-formulas-of-alt-def*:

$\text{labeled-formulas-of } (P, Y, A) =$

$(\lambda C. (C, \text{Passive})) \cdot P \cup (\lambda C. (C, YY)) \cdot Y \cup (\lambda C. (C, \text{Active})) \cdot A$

by *auto*

fun

$\text{state} :: 'f \text{ inference set} \times 'f \text{ set} \times 'f \text{ set} \times 'f \text{ set} \Rightarrow 'f \text{ inference set} \times ('f \times DL\text{-label}) \text{ set}$

where

$\text{state } (T, P, Y, A) = (T, \text{labeled-formulas-of } (P, Y, A))$

lemma *state-alt-def*:

$\text{state } (T, P, Y, A) = (T, (\lambda C. (C, \text{Passive})) \cdot P \cup (\lambda C. (C, YY)) \cdot Y \cup (\lambda C. (C, \text{Active})) \cdot A)$

by *auto*

inductive

$DL :: 'f \text{ inference set} \times ('f \times DL\text{-label}) \text{ set} \Rightarrow 'f \text{ inference set} \times ('f \times DL\text{-label}) \text{ set} \Rightarrow \text{bool}$
(infix \rightsquigarrow *DL 50*)

where

$\text{compute-infer}: \iota \in \text{no-labels.Red-I } (A \cup \{C\}) \Longrightarrow$

$\text{state } (T \cup \{\iota\}, P, \{\}, A) \rightsquigarrow DL \text{ state } (T, P, \{C\}, A)$

| $\text{choose-p}: \text{state } (T, P \cup \{C\}, \{\}, A) \rightsquigarrow DL \text{ state } (T, P, \{C\}, A)$

| $\text{delete-fwd}: C \in \text{no-labels.Red-F } A \vee (\exists C' \in A. C' \preceq \cdot C) \Longrightarrow$

$\text{state } (T, P, \{C\}, A) \rightsquigarrow DL \text{ state } (T, P, \{\}, A)$

| $\text{simplify-fwd}: C \in \text{no-labels.Red-F } (A \cup \{C'\}) \Longrightarrow$

$\text{state } (T, P, \{C\}, A) \rightsquigarrow DL \text{ state } (T, P, \{C'\}, A)$

| $\text{delete-bwd}: C' \in \text{no-labels.Red-F } \{C\} \vee C' \cdot \succ C \Longrightarrow$

$\text{state } (T, P, \{C\}, A \cup \{C'\}) \rightsquigarrow DL \text{ state } (T, P, \{C\}, A)$

| $\text{simplify-bwd}: C' \in \text{no-labels.Red-F } \{C, C''\} \Longrightarrow$

$\text{state } (T, P, \{C\}, A \cup \{C'\}) \rightsquigarrow DL \text{ state } (T, P \cup \{C''\}, \{C\}, A)$

| $\text{schedule-infer}: T' = \text{no-labels.Inf-between } A \{C\} \Longrightarrow$

$\text{state } (T, P, \{C\}, A) \rightsquigarrow DL \text{ state } (T \cup T', P, \{\}, A \cup \{C\})$

| $\text{delete-orphan-infers}: T' \cap \text{no-labels.Inf-from } A = \{\} \Longrightarrow$

$\text{state } (T \cup T', P, Y, A) \rightsquigarrow DL \text{ state } (T, P, Y, A)$

lemma *If-f-in-A-then-ft-in-PYA*: $C' \in A \Longrightarrow (C', \text{Active}) \in \text{labeled-formulas-of } (P, Y, A)$

by *auto*

lemma *PYA-add-passive-formula[simp]*:

$\text{labeled-formulas-of } (P, Y, A) \cup \{(C, \text{Passive})\} = \text{labeled-formulas-of } (P \cup \{C\}, Y, A)$

by *auto*

lemma *P0A-add-y-formula[simp]*:

labeled-formulas-of $(P, \{\}, A) \cup \{(C, YY)\} = \textit{labeled-formulas-of}$ $(P, \{C\}, A)$

by *auto*

lemma *PYA-add-active-formula[simp]*:

labeled-formulas-of $(P, Y, A) \cup \{(C', \textit{Active})\} = \textit{labeled-formulas-of}$ $(P, Y, A \cup \{C'\})$

by *auto*

lemma *prj-active-subset-of-state*: $\textit{fst} \textit{' active-subset}$ $(\textit{labeled-formulas-of}$ $(P, Y, A)) = A$

proof –

have *active-subset* $\{(C, YY) \mid C. C \in Y\} = \{\}$ **and**

active-subset $\{(C, \textit{Passive}) \mid C. C \in P\} = \{\}$

using *active-subset-def* **by** *auto*

moreover have *active-subset* $\{(C, \textit{Active}) \mid C. C \in A\} = \{(C, \textit{Active}) \mid C. C \in A\}$

using *active-subset-def* **by** *fastforce*

ultimately have $\textit{fst} \textit{' active-subset}$ $(\textit{labeled-formulas-of}$ $(P, Y, A)) =$

$\textit{fst} \textit{'}$ $\{(C, \textit{Active}) \mid C. C \in A\}$

by *simp*

then show *?thesis*

by *simp*

qed

lemma *active-subset-of-setOfFormulasWithLabelDiffActive*:

$l \neq \textit{Active} \implies \textit{active-subset}$ $\{(C', l)\} = \{\}$

using *active-subset-def* **by** *auto*

3.3 Refinement

lemma *dl-compute-infer-in-lgc*:

assumes $\iota \in \textit{no-labels.Red-I-G}$ $(A \cup \{C\})$

shows \textit{state} $(T \cup \{\iota\}, P, \{\}, A) \rightsquigarrow \textit{LGC state}$ $(T, P, \{C\}, A)$

proof –

let $\textit{?N} = \textit{labeled-formulas-of}$ $(P, \{\}, A)$

and $\textit{?M} = \{(C, YY)\}$

have $A \cup \{C\} \subseteq \textit{fst} \textit{'}$ $(\textit{labeled-formulas-of}$ $(P, \{\}, A) \cup \{(C, YY)\})$

by *auto*

then have $\iota \in \textit{no-labels.Red-I-G}$ $(\textit{fst} \textit{'}$ $(\textit{?N} \cup \textit{?M}))$

by $(\textit{meson assms no-labels.empty-ord.Red-I-of-subset subsetD})$

also have *active-subset* $\textit{?M} = \{\}$

using *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *auto*

then have $(T \cup \{\iota\}, \textit{?N}) \rightsquigarrow \textit{LGC}$ $(T, \textit{?N} \cup \textit{?M})$

using *calculation lgc.step.compute-infer* **by** *blast*

moreover have $\textit{?N} \cup \textit{?M} = \textit{labeled-formulas-of}$ $(P, \{C\}, A)$

by *simp*

ultimately show *?thesis*

by *auto*

qed

lemma *dl-choose-p-in-lgc*: \textit{state} $(T, P \cup \{C\}, \{\}, A) \rightsquigarrow \textit{LGC state}$ $(T, P, \{C\}, A)$

proof –

let $\textit{?N} = \textit{labeled-formulas-of}$ $(P, \{\}, A)$

have *Passive* $\sqsupseteq L$ *YY*

by $(\textit{simp add: DL-Prec-L-def})$

then have $(T, \textit{?N} \cup \{(C, \textit{Passive})\}) \rightsquigarrow \textit{LGC}$ $(T, \textit{?N} \cup \{(C, YY)\})$

using *relabel-inactive* **by** *blast*

then have $(T, \text{labeled-formulas-of } (P \cup \{C\}, \{\}, A)) \rightsquigarrow LGC (T, \text{labeled-formulas-of } (P, \{C\}, A))$
by $(\text{metis PYA-add-passive-formula P0A-add-y-formula})$
then show *?thesis*
by *auto*
qed

lemma *dl-delete-fwd-in-lgc:*

assumes $(C \in \text{no-labels.Red-F } A) \vee (\exists C' \in A. C' \preceq C)$
shows state $(T, P, \{C\}, A) \rightsquigarrow LGC \text{ state } (T, P, \{\}, A)$
using *assms*

proof

assume *c-in*: $C \in \text{no-labels.Red-F } A$
then have $A \subseteq \text{fst}' (\text{labeled-formulas-of } (P, \{\}, A))$
by *simp*
then have $C \in \text{no-labels.Red-F } (\text{fst}' (\text{labeled-formulas-of } (P, \{\}, A)))$
by $(\text{metis } (\text{no-types, lifting}) \text{ c-in in-mono no-labels.Red-F-of-subset})$
then show *?thesis*
using *remove-redundant-no-label* **by** *auto*

next

assume $\exists C' \in A. C' \preceq C$
then obtain C' **where** *c'-in-and-c'-ls-c*: $C' \in A \wedge C' \preceq C$
by *auto*
then have $(C', \text{Active}) \in \text{labeled-formulas-of } (P, \{\}, A)$
by *auto*
then have $YY \sqsupseteq L \text{Active}$
by $(\text{simp add: DL-Prec-L-def})$
then show *?thesis*
by $(\text{metis } \text{c'-in-and-c'-ls-c remove-succ-L state.simps P0A-add-y-formula If-f-in-A-then-fl-in-PYA})$

qed

lemma *dl-simplify-fwd-in-lgc:*

assumes $C \in \text{no-labels.Red-F-}\mathcal{G} (A \cup \{C\})$
shows state $(T, P, \{C\}, A) \rightsquigarrow LGC \text{ state } (T, P, \{C\}, A)$

proof –

let $?N = \text{labeled-formulas-of } (P, \{\}, A)$
and $?M = \{(C, YY)\}$
and $?M' = \{(C', YY)\}$
have $A \cup \{C\} \subseteq \text{fst}' (?N \cup ?M)$
by *auto*
then have $C \in \text{no-labels.Red-F-}\mathcal{G} (\text{fst}' (?N \cup ?M'))$
by $(\text{smt } (\text{verit, ccfv-threshold}) \text{ assms no-labels.Red-F-of-subset subset-iff})$
then have $(C, YY) \in \text{Red-F } (?N \cup ?M')$
using *no-labels-Red-F-imp-Red-F* **by** *simp*
then have $?M \subseteq \text{Red-F-}\mathcal{G} (?N \cup ?M')$
by *simp*
moreover have *active-subset* $?M' = \{\}$
using *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *blast*
ultimately have $(T, \text{labeled-formulas-of } (P, \{\}, A) \cup \{(C, YY)\}) \rightsquigarrow LGC$
 $(T, \text{labeled-formulas-of } (P, \{\}, A) \cup \{(C', YY)\})$
using *process[of - - ?M - ?M']* **by** *auto*
then show *?thesis*
by *simp*

qed

lemma *dl-delete-bwd-in-lgc*:

assumes $C' \in \text{no-labels.Red-F-G } \{C\} \vee C' \cdot \succ C$
shows $\text{state } (T, P, \{C\}, A \cup \{C'\}) \sim_{LGC} \text{state } (T, P, \{C\}, A)$
using *assms*

proof

let $?N = \text{labeled-formulas-of } (P, \{C\}, A)$
assume $c'\text{-in: } C' \in \text{no-labels.Red-F-G } \{C\}$
have $\{C\} \subseteq \text{fst } ' ?N$
by *simp*
then have $C' \in \text{no-labels.Red-F-G } (\text{fst } ' ?N)$
by (*metis* (*no-types, lifting*) *c'-in insert-Diff insert-subset no-labels.Red-F-of-subset*)
then have $(T, ?N \cup \{(C', \text{Active})\}) \sim_{LGC} (T, ?N)$
using *remove-redundant-no-label* **by** *auto*
then show *?thesis*
by (*metis* *state.simps PYA-add-active-formula*)

next

assume $C' \cdot \succ C$
moreover have $(C, YY) \in \text{labeled-formulas-of } (P, \{C\}, A)$
by *simp*
ultimately show *?thesis*
by (*metis* *remove-succ-F state.simps PYA-add-active-formula*)

qed

lemma *dl-simplify-bwd-in-lgc*:

assumes $C' \in \text{no-labels.Red-F-G } \{C, C''\}$
shows $\text{state } (T, P, \{C\}, A \cup \{C'\}) \sim_{LGC} \text{state } (T, P \cup \{C''\}, \{C\}, A)$

proof –

let $?M = \{(C', \text{Active})\}$
and $?M' = \{(C'', \text{Passive})\}$
and $?N = \text{labeled-formulas-of } (P, \{C\}, A)$

have $\{C, C''\} \subseteq \text{fst } ' (?N \cup ?M')$
by *simp*
then have $C' \in \text{no-labels.Red-F-G } (\text{fst } ' (?N \cup ?M'))$
by (*smt* (*z3*) *DiffI Diff-eq-empty-iff assms empty-iff no-labels.Red-F-of-subset*)
then have $M\text{-included: } ?M \subseteq \text{Red-F-G } (?N \cup ?M')$
using *no-labels-Red-F-imp-Red-F* **by** *auto*
then have $\text{active-subset } ?M' = \{\}$
using *active-subset-def* **by** *auto*
then have $(T, ?N \cup ?M) \sim_{LGC} (T, ?N \cup ?M')$
using $M\text{-included process[of - ?M - ?M']}$ **by** *auto*
moreover have $?N \cup ?M = \text{labeled-formulas-of } (P, \{C\}, A \cup \{C'\})$
and $?N \cup ?M' = \text{labeled-formulas-of } (P \cup \{C''\}, \{C\}, A)$
by *auto*
ultimately show *?thesis*
by *auto*

qed

lemma *dl-schedule-infer-in-lgc*:

assumes $T' = \text{no-labels.Inf-between } A \{C\}$
shows $\text{state } (T, P, \{C\}, A) \sim_{LGC} \text{state } (T \cup T', P, \{\}, A \cup \{C\})$

proof –

let $?N = \text{labeled-formulas-of } (P, \{\}, A)$
have $\text{fst } ' (\text{active-subset } ?N) = A$
using *prj-active-subset-of-state* **by** *blast*

then have $T' = \text{no-labels.Inf-between } (\text{fst } ' (\text{active-subset } ?\mathcal{N})) \{C\}$
using *assms* **by** *auto*
then have $(T, \text{labeled-formulas-of } (P, \{\}, A) \cup \{(C, YY)\}) \rightsquigarrow \text{LGC}$
 $(T \cup T', \text{labeled-formulas-of } (P, \{\}, A) \cup \{(C, \text{Active})\})$
using *lgc.step.schedule-infer* **by** *blast*
then show *?thesis*
by $(\text{metis } \text{state.simps } P0A\text{-add-y-formula } PYA\text{-add-active-formula})$
qed

lemma *dl-delete-orphan-infers-in-lgc*:
assumes $T' \cap \text{no-labels.Inf-from } A = \{\}$
shows $\text{state } (T \cup T', P, Y, A) \rightsquigarrow \text{LGC } \text{state } (T, P, Y, A)$

proof –
let $?\mathcal{N} = \text{labeled-formulas-of } (P, Y, A)$
have $\text{fst } ' (\text{active-subset } ?\mathcal{N}) = A$
using *prj-active-subset-of-state* **by** *blast*
then have $T' \cap \text{no-labels.Inf-from } (\text{fst } ' (\text{active-subset } ?\mathcal{N})) = \{\}$
using *assms* **by** *simp*
then have $(T \cup T', ?\mathcal{N}) \rightsquigarrow \text{LGC } (T, ?\mathcal{N})$
using *lgc.step.delete-orphan-infers* **by** *blast*
then show *?thesis*
by *simp*
qed

theorem *DL-step-imp-LGC-step*: $TM \rightsquigarrow \text{DL } TM' \implies TM \rightsquigarrow \text{LGC } TM'$

proof (*induction rule: DL.induct*)
case $(\text{compute-infer } \iota A C T P)$
then show *?case*
using *dl-compute-infer-in-lgc* **by** *blast*
next
case $(\text{choose-p } T P C A)$
then show *?case*
using *dl-choose-p-in-lgc* **by** *auto*
next
case $(\text{delete-fwd } C A T P)$
then show *?case*
using *dl-delete-fwd-in-lgc* **by** *auto*
next
case $(\text{simplify-fwd } C A C' T P)$
then show *?case*
using *dl-simplify-fwd-in-lgc* **by** *blast*
next
case $(\text{delete-bwd } C' C T P A)$
then show *?case*
using *dl-delete-bwd-in-lgc* **by** *blast*
next
case $(\text{simplify-bwd } C' C C'' T P A)$
then show *?case*
using *dl-simplify-bwd-in-lgc* **by** *blast*
next
case $(\text{schedule-infer } T' A C T P)$
then show *?case*
using *dl-schedule-infer-in-lgc* **by** *blast*
next
case $(\text{delete-orphan-infers } T' A T P Y)$

then show *?case*
using *dl-delete-orphan-infers-in-lgc* **by** *blast*
qed

3.4 Completeness

theorem

assumes

dl-chain: *chain* ($\sim DL$) *Sts* **and**
act: *active-subset* (*snd* (*lhd Sts*)) = {} **and**
pas: *passive-subset* (*Liminf-list* (*lmap snd Sts*)) = {} **and**
no-prems-init: $\forall \iota \in Inf-F. \text{prems-of } \iota = [] \longrightarrow \iota \in \text{fst } (lhd Sts)$ **and**
final-sched: *Liminf-list* (*lmap fst Sts*) = {}

shows

DL-Liminf-saturated: *saturated* (*Liminf-list* (*lmap snd Sts*)) **and**
DL-complete-Liminf: $B \in Bot-F \implies \text{fst } 'snd (lhd Sts) \models_{\cap \mathcal{G}} \{B\} \implies$
 $\exists BL \in Bot-FL. BL \in Liminf-list (lmap snd Sts)$ **and**
DL-complete: $B \in Bot-F \implies \text{fst } 'snd (lhd Sts) \models_{\cap \mathcal{G}} \{B\} \implies$
 $\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists BL \in Bot-FL. BL \in \text{snd } (lnth Sts i))$

proof –

have *lgc-chain*: *chain* ($\sim LGC$) *Sts*
using *dl-chain DL-step-imp-LGC-step chain-mono* **by** *blast*

show *saturated* (*Liminf-list* (*lmap snd Sts*))

using *act final-sched lgc.fair-implies-Liminf-saturated lgc-chain lgc-fair lgc-to-red*
no-prems-init pas **by** *blast*

{

assume

bot: $B \in Bot-F$ **and**
unsat: $\text{fst } 'snd (lhd Sts) \models_{\cap \mathcal{G}} \{B\}$

show $\exists BL \in Bot-FL. BL \in Liminf-list (lmap snd Sts)$

by (*rule lgc-complete-Liminf[OF lgc-chain act pas no-prems-init final-sched bot unsat]*)

then show $\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists BL \in Bot-FL. BL \in \text{snd } (lnth Sts i))$

unfolding *Liminf-list-def* **by** *auto*

}

qed

end

end

4 Prover Queues and Fairness

This section covers the passive set data structure that arises in different prover loops in the literature (e.g., DISCOUNT, Otter).

theory *Prover-Queue*

imports

Given-Clause-Loops-Util
Ordered-Resolution-Prover.Lazy-List-Chain

begin

4.1 Basic Lemmas

lemma *set-drop-fold-maybe-append-singleton*:

```

    set (drop k (fold (λy xs. if y ∈ set xs then xs else xs @ [y]) ys xs)) ⊆ set (drop k (xs @ ys))
proof (induct ys arbitrary: xs)
  case (Cons y ys)
  note ih = this(1)
  show ?case
  proof (cases y ∈ set xs)
    case True
    thus ?thesis
    using ih[of xs] set-drop-append-cons[of k xs ys y] by auto
  next
  case False
  then show ?thesis
    using ih[of xs @ [y]]
    by simp
  qed
qed simp

```

```

lemma fold-maybe-append-removeAll:
  assumes y ∈ set xs
  shows fold (λy xs. if y ∈ set xs then xs else xs @ [y]) (removeAll y ys) xs =
    fold (λy xs. if y ∈ set xs then xs else xs @ [y]) ys xs
  using assms by (induct ys arbitrary: xs) auto

```

4.2 More on Relational Chains over Lazy Lists

```

definition finitely-often :: ('a ⇒ 'a ⇒ bool) ⇒ 'a llist ⇒ bool where
  finitely-often R xs ←→
  (∃ i. ∀ j. i ≤ j → enat (Suc j) < llength xs → ¬ R (lnth xs j) (lnth xs (Suc j)))

```

```

abbreviation infinitely-often :: ('a ⇒ 'a ⇒ bool) ⇒ 'a llist ⇒ bool where
  infinitely-often R xs ≡ ¬ finitely-often R xs

```

```

lemma infinitely-often-alt-def:
  infinitely-often R xs ←→
  (∀ i. ∃ j. i ≤ j ∧ enat (Suc j) < llength xs ∧ R (lnth xs j) (lnth xs (Suc j)))
  unfolding finitely-often-def by blast

```

```

lemma infinitely-often-lifting:
  assumes
    r-imp-s: ∀ x x'. R (f x) (f x') → S (g x) (g x') and
    inf-r: infinitely-often R (lmap f xs)
  shows infinitely-often S (lmap g xs)
  using inf-r unfolding infinitely-often-alt-def
  by (metis Suc-ile-eq llength-lmap lnth-lmap order-less-imp-le r-imp-s)

```

4.3 Locales

The passive set of a given clause prover can be organized in different ways—e.g., as a priority queue or as a list of queues. This locale abstracts over the specific data structure.

```

locale prover-queue =
  fixes
    empty :: 'q and
    select :: 'q ⇒ 'e and
    add :: 'e ⇒ 'q ⇒ 'q and
    remove :: 'e ⇒ 'q ⇒ 'q and

```

$felems :: 'q \Rightarrow 'e \text{ fset}$
assumes
 $felems\text{-empty}[simp]: felems\ empty = \{\}\ \mathbf{and}$
 $felems\text{-not-empty}: Q \neq empty \Longrightarrow felems\ Q \neq \{\}\ \mathbf{and}$
 $select\text{-in-felems}[simp]: Q \neq empty \Longrightarrow select\ Q\ |\in| felems\ Q\ \mathbf{and}$
 $felems\text{-add}[simp]: felems\ (add\ e\ Q) = \{e\} \cup felems\ Q\ \mathbf{and}$
 $felems\text{-remove}[simp]: felems\ (remove\ e\ Q) = felems\ Q\ -\ \{e\}\ \mathbf{and}$
 $add\text{-again}: e\ |\in| felems\ Q \Longrightarrow add\ e\ Q = Q$
begin

abbreviation $elems :: 'q \Rightarrow 'e \text{ set where}$
 $elems\ Q \equiv \text{fset}\ (felems\ Q)$

lemma $elems\text{-empty}: elems\ empty = \{\}$
by $simp$

lemma $formula\text{-not-empty}[simp]: Q \neq empty \Longrightarrow elems\ Q \neq \{\}$
by $(metis\ bot\text{-fset.rep-eq}\ felems\text{-not-empty}\ \text{fset-cong})$

lemma
 $elems\text{-add}: elems\ (add\ e\ Q) = \{e\} \cup elems\ Q\ \mathbf{and}$
 $elems\text{-remove}: elems\ (remove\ e\ Q) = elems\ Q - \{e\}$
by $simp+$

lemma $elems\text{-fold-add}[simp]: elems\ (fold\ add\ es\ Q) = set\ es \cup elems\ Q$
by $(induct\ es\ arbitrary: Q)\ auto$

lemma $elems\text{-fold-remove}[simp]: elems\ (fold\ remove\ es\ Q) = elems\ Q - set\ es$
by $(induct\ es\ arbitrary: Q)\ auto$

inductive $queue\text{-step} :: 'q \Rightarrow 'q \Rightarrow bool\ \mathbf{where}$
 $queue\text{-step-fold-addI}: queue\text{-step}\ Q\ (fold\ add\ es\ Q)$
 $| queue\text{-step-fold-removeI}: queue\text{-step}\ Q\ (fold\ remove\ es\ Q)$

lemma $queue\text{-step-idleI}: queue\text{-step}\ Q\ Q$
using $queue\text{-step-fold-addI}[of\ -\ [],\ simplified]$.

lemma $queue\text{-step-addI}: queue\text{-step}\ Q\ (add\ e\ Q)$
using $queue\text{-step-fold-addI}[of\ -\ [e],\ simplified]$.

lemma $queue\text{-step-removeI}: queue\text{-step}\ Q\ (remove\ e\ Q)$
using $queue\text{-step-fold-removeI}[of\ -\ [e],\ simplified]$.

inductive $select\text{-queue-step} :: 'q \Rightarrow 'q \Rightarrow bool\ \mathbf{where}$
 $select\text{-queue-stepI}: Q \neq empty \Longrightarrow select\text{-queue-step}\ Q\ (remove\ (select\ Q)\ Q)$

end

locale $fair\text{-prover-queue} = \text{prover-queue}\ empty\ select\ add\ remove\ felems$
for
 $empty :: 'q\ \mathbf{and}$
 $select :: 'q \Rightarrow 'e\ \mathbf{and}$
 $add :: 'e \Rightarrow 'q \Rightarrow 'q\ \mathbf{and}$
 $remove :: 'e \Rightarrow 'q \Rightarrow 'q\ \mathbf{and}$
 $felems :: 'q \Rightarrow 'e \text{ fset} +$

```

assumes fair: chain queue-step Qs  $\implies$  infinitely-often select-queue-step Qs  $\implies$ 
  lhd Qs = empty  $\implies$  Liminf-list (lmap elems Qs) = {}
begin
end

```

4.4 Instantiation with FIFO Queue

As a proof of concept, we show that a FIFO queue can serve as a fair prover queue.

```

locale fifo-prover-queue
begin

```

```

sublocale prover-queue [] hd  $\lambda y$  xs. if  $y \in$  set xs then xs else xs @ [y] removeAll fset-of-list
proof

```

```

  show  $\bigwedge Q. Q \neq [] \implies$  fset-of-list Q  $\neq$  {}
  by (metis fset-of-list.rep-eq fset-simps(1) set-empty)
qed (auto simp: fset-of-list-elem)

```

```

lemma queue-step-preserves-distinct:

```

```

assumes
  dist: distinct Q and
  step: queue-step Q Q'
shows distinct Q'
using step
proof cases
case (queue-step-fold-addI es)
  note p' = this(1)
  show ?thesis
  unfolding p'
  using dist
proof (induct es arbitrary: Q)
  case Nil
  then show ?case
  using dist by auto
next
  case (Cons e es)
  note ih = this(1) and dist-p = this(2)

  show ?case
  proof (cases e  $\in$  set Q)
  case True
  then show ?thesis
  using ih[OF dist-p] by simp
  next
  case c-ni: False
  have dist-pc: distinct (Q @ [e])
  using c-ni dist-p by auto
  show ?thesis
  using c-ni using ih[OF dist-pc] by simp
  qed
  qed
next
  case (queue-step-fold-removeI es)
  note p' = this(1)
  show ?thesis
  unfolding p' using dist by (simp add: distinct-fold-removeAll)

```

qed

lemma chain-queue-step-preserves-distinct:

assumes

chain: chain queue-step Qs and

dist-hd: distinct (lhd Qs) and

i-lt: enat i < llength Qs

shows distinct (lnth Qs i)

using i-lt

proof (induct i)

case 0

then show ?case

using dist-hd chain-length-pos[OF chain] by (simp add: lhd-conv-lnth)

next

case (Suc i)

have ih: distinct (lnth Qs i)

using Suc.hyps Suc.premis Suc-ile-eq order-less-imp-le by blast

have queue-step (lnth Qs i) (lnth Qs (Suc i))

by (rule chain-lnth-rel[OF chain Suc.premis])

then show ?case

using queue-step-preserves-distinct ih by blast

qed

sublocale fair-prover-queue [] hd $\lambda y xs.$ if $y \in \text{set } xs$ then xs else $xs @ [y]$ removeAll
fset-of-list

proof

fix Qs :: 'e list llist

assume

chain: chain queue-step Qs and

inf-sel: infinitely-often select-queue-step Qs and

hd-emp: lhd Qs = []

show Liminf-llist (lmap elems Qs) = {}

proof (rule ccontr)

assume lim-nemp: Liminf-llist (lmap elems Qs) \neq {}

obtain i :: nat where

i-lt: enat i < llength Qs and

inter-nemp: $\bigcap ((\text{set} \circ \text{lnth } Qs) \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } Qs\}) \neq \{\}$

using lim-nemp unfolding Liminf-llist-def by auto

from inter-nemp obtain e :: 'e where

$\forall Q \in \text{lnth } Qs \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } Qs\}. e \in \text{set } Q$

by auto

hence c-in: $\forall j \geq i. \text{enat } j < \text{llength } Qs \longrightarrow e \in \text{set } (\text{lnth } Qs j)$

by auto

have ps-inf: llength Qs = ∞

proof (rule ccontr)

assume llength Qs $\neq \infty$

obtain n :: nat where

n: enat n = llength Qs

using <llength Qs $\neq \infty$ > by force

```

show False
  using inf-sel[unfolded infinitely-often-alt-def]
  by (metis Suc-lessD enat-ord-simps(2) less-le-not-le n)
qed

have c-in':  $\forall j \geq i. e \in \text{set} (\text{lnth } Qs\ j)$ 
  by (simp add: c-in ps-inf)
then obtain k :: nat where
  k-lt:  $k < \text{length} (\text{lnth } Qs\ i)$  and
  at-k:  $\text{lnth } Qs\ i\ !\ k = e$ 
  by (meson in-set-conv-nth le-refl)

have dist: distinct ( $\text{lnth } Qs\ i$ )
  by (simp add: chain-queue-step-preserves-distinct hd-emp i-lt chain)

have  $\forall k' \leq k + 1. \exists i' \geq i. e \notin \text{set} (\text{drop } k'\ (\text{lnth } Qs\ i'))$ 
proof -
  have  $\exists i' \geq i. e \notin \text{set} (\text{drop } (k + 1 - l)\ (\text{lnth } Qs\ i'))$  for l
  proof (induct l)
    case 0
    have  $e \notin \text{set} (\text{drop } (k + 1)\ (\text{lnth } Qs\ i))$ 
      by (simp add: at-k dist distinct-imp-notin-set-drop-Suc k-lt)
    then show ?case
      by auto
  next
  case (Suc l)
  then obtain i' :: nat where
    i'-ge:  $i' \geq i$  and
    c-ni-i':  $e \notin \text{set} (\text{drop } (k + 1 - l)\ (\text{lnth } Qs\ i'))$ 
    by blast

  obtain i'' :: nat where
    i''-ge:  $i'' \geq i'$  and
    i''-lt:  $\text{enat } (Suc\ i'') < \text{length } Qs$  and
    sel-step: select-queue-step ( $\text{lnth } Qs\ i''$ ) ( $\text{lnth } Qs\ (Suc\ i'')$ )
    using inf-sel[unfolded infinitely-often-alt-def] by blast

  have c-ni-i'-i'':  $e \notin \text{set} (\text{drop } (k + 1 - l)\ (\text{lnth } Qs\ j))$ 
    if j-ge:  $j \geq i'$  and j-le:  $j \leq i''$  for j
    using j-ge j-le
  proof (induct j rule: less-induct)
    case (less d)
    note ih = this(1)

    show ?case
    proof (cases d < i')
      case True
      then show ?thesis
        using less.prem(1) by linarith
    next
    case False
    hence d-ge:  $d \geq i'$ 
      by simp
    then show ?thesis

```



```

proof (cases d > i'')
  case True
  then show ?thesis
    using less.premis(2) linorder-not-less by blast
next
  case False
  hence d-le: d ≤ i''
    by simp

  show ?thesis
proof (cases d = i')
  case True
  then show ?thesis
    using c-ni-i' by blast
next
  case False
  note d-ne-i' = this(1)

  have dm1-bounds:
    d - 1 < d
    i' ≤ d - 1
    d - 1 ≤ i''
    using d-ge d-le d-ne-i' by auto
  have ih-dm1: e ∉ set (drop (k + 1 - l) (lnth Qs (d - 1)))
    by (rule ih[OF dm1-bounds])

  have queue-step (lnth Qs (d - 1)) (lnth Qs d)
    by (metis (no-types, lifting) One-nat-def add-diff-inverse-nat
      bot-nat-0.extremum-unique chain chain-lnth-rel d-ge d-ne-i' dm1-bounds(2)
      enat-ord-code(4) le-less-Suc-eq nat-diff-split plus-1-eq-Suc ps-inf)
  then show ?thesis
proof cases
  case (queue-step-fold-addI es)

  note at-d = this(1)

  have c-in: e ∈| fset-of-list (lnth Qs (d - 1))
    by (meson c-in' dm1-bounds(2) fset-of-list-elem i'-ge order-trans)
  hence e ∉ set (drop (k + 1 - l)
    (fold (λy xs. if y ∈ set xs then xs else xs @ [y]) (removeAll e es)
      (lnth Qs (d - 1))))
  proof -
    have set (drop (k + 1 - l)
      (fold (λy xs. if y ∈ set xs then xs else xs @ [y]) (removeAll e es)
        (lnth Qs (d - 1)))) ⊆
      set (drop (k + 1 - l) (lnth Qs (d - 1) @ removeAll e es))
    using set-drop-fold-maybe-append-singleton .
  have e ∉ set (drop (k + 1 - l) (lnth Qs (d - 1)))
    using ih-dm1 by blast
  hence e ∉ set (drop (k + 1 - l) (lnth Qs (d - 1) @ removeAll e es))
    using set-drop-append-subseteq by force
  thus ?thesis
    using set-drop-fold-maybe-append-singleton by force
qed
  hence e ∉ set (drop (k + 1 - l)

```

```

      (fold (λy xs. if y ∈ set xs then xs else xs @ [y]) es (lnth Qs (d - 1)))
    using c-in fold-maybe-append-removeAll
    by (metis (mono-tags, lifting) fset-of-list-elem)
  thus ?thesis
    unfolding at-d by fastforce
  next
    case (queue-step-fold-removeI es)
    note at-d = this(1)
    show ?thesis
      unfolding at-d using ih-dm1 set-drop-fold-removeAll by fastforce
    qed
  qed
  qed
  qed
  qed

  have Suc i'' > i
    using i''-ge i'-ge by linarith
  moreover have e ∉ set (drop (k + 1 - Suc l) (lnth Qs (Suc i'')))
    using sel-step
  proof cases
    case select-queue-stepI
    note at-si'' = this(1) and at-i''-nemp = this(2)

    have at-i''-nnil: lnth Qs i'' ≠ []
      using at-i''-nemp by auto

    have dist-i'': distinct (lnth Qs i'')
      by (simp add: chain-queue-step-preserves-distinct hd-emp chain ps-inf)

    have c-ni-i'': e ∉ set (drop (k + 1 - l) (lnth Qs i''))
      using c-ni-i'-i'' i''-ge by blast

    show ?thesis
      unfolding at-si''
      by (subst distinct-set-drop-removeAll-hd[OF dist-i'' at-i''-nnil])
        (metis Suc-diff-Suc bot-nat-0.not-eq-extremum c-ni-i'' drop0 in-set-dropD
          zero-less-diff)
    qed
  ultimately show ?case
    by (rule-tac x = Suc i'' in exI) auto
  qed
  thus ?thesis
    by (metis diff-add-zero drop0 in-set-dropD)
  qed
  then obtain i' :: nat where
    i' ≥ i
    e ∉ set (lnth Qs i')
    by fastforce
  then show False
    using c-in' by auto
  qed
  qed

end

```

end

5 Fair DISCOUNT Loop

The fair DISCOUNT loop assumes that the passive queue is fair and ensures (dynamic) refutational completeness under that assumption.

theory *Fair-DISCOUNT-Loop*

imports

Given-Clause-Loops-Util

DISCOUNT-Loop

Prover-Queue

begin

5.1 Locale

type-synonym ('p, 'f) *DLf-state* = 'p × 'f option × 'f fset

datatype 'f *passive-elem* =

is-passive-inference: *Passive-Inference* (*passive-inference*: 'f *inference*)

| *is-passive-formula*: *Passive-Formula* (*passive-formula*: 'f)

lemma *passive-inference-filter*:

passive-inference ' Set.filter *is-passive-inference* N = {ι. *Passive-Inference* ι ∈ N}

by force

lemma *passive-formula-filter*:

passive-formula ' Set.filter *is-passive-formula* N = {C. *Passive-Formula* C ∈ N}

by force

locale *fair-discount-loop* =

discount-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F +

fair-prover-queue empty select add remove felems

for

Bot-F :: 'f set **and**

Inf-F :: 'f *inference* set **and**

Bot-G :: 'g set **and**

Q :: 'q set **and**

entails-q :: 'q ⇒ 'g set ⇒ 'g set ⇒ bool **and**

Inf-G-q :: 'q ⇒ 'g *inference* set **and**

Red-I-q :: 'q ⇒ 'g set ⇒ 'g *inference* set **and**

Red-F-q :: 'q ⇒ 'g set ⇒ 'g set **and**

G-F-q :: 'q ⇒ 'f ⇒ 'g set **and**

G-I-q :: 'q ⇒ 'f *inference* ⇒ 'g *inference* set option **and**

Equiv-F :: 'f ⇒ 'f ⇒ bool (**infix** <=> 50) **and**

Prec-F :: 'f ⇒ 'f ⇒ bool (**infix** << 50) **and**

empty :: 'p **and**

select :: 'p ⇒ 'f *passive-elem* **and**

add :: 'f *passive-elem* ⇒ 'p ⇒ 'p **and**

remove :: 'f *passive-elem* ⇒ 'p ⇒ 'p **and**

felems :: 'p ⇒ 'f *passive-elem* fset +

fixes

Prec-S :: 'f ⇒ 'f ⇒ bool (**infix** <S 50)

assumes

wf-Prec-S: minimal-element (\prec_S) UNIV and
transp-Prec-S: transp (\prec_S) and
finite-Inf-between: finite $A \implies$ finite (no-labels.Inf-between $A \{C\}$)
begin

lemma *trans-Prec-S: trans $\{(x, y). x \prec_S y\}$*
using *transp-Prec-S transp-trans by blast*

lemma *irreflp-Prec-S: irreflp (\prec_S)*
using *minimal-element.wf wfp-imp-irreflp wf-Prec-S wfp-on-UNIV by blast*

lemma *irrefl-Prec-S: irrefl $\{(x, y). x \prec_S y\}$*
by *(metis CollectD case-prod-conv irrefl-def irreflp-Prec-S irreflp-def)*

5.2 Basic Definitions and Lemmas

abbreviation *passive-of :: ('p, 'f) DLf-state \Rightarrow 'p where*
passive-of St \equiv fst St

abbreviation *yy-of :: ('p, 'f) DLf-state \Rightarrow 'f option where*
yy-of St \equiv fst (snd St)

abbreviation *active-of :: ('p, 'f) DLf-state \Rightarrow 'f fset where*
active-of St \equiv snd (snd St)

definition *passive-inferences-of :: 'p \Rightarrow 'f inference set where*
passive-inferences-of P = $\{\iota. \text{Passive-Inference } \iota \in \text{elems } P\}$

definition *passive-formulas-of :: 'p \Rightarrow 'f set where*
passive-formulas-of P = $\{C. \text{Passive-Formula } C \in \text{elems } P\}$

lemma *finite-passive-inferences-of: finite (passive-inferences-of P)*

proof –

have *inj-pi: inj Passive-Inference*

unfolding *inj-on-def by auto*

show *?thesis*

unfolding *passive-inferences-of-def by (auto intro: finite-inverse-image[OF - inj-pi])*

qed

lemma *finite-passive-formulas-of: finite (passive-formulas-of P)*

proof –

have *inj-pi: inj Passive-Formula*

unfolding *inj-on-def by auto*

show *?thesis*

unfolding *passive-formulas-of-def by (auto intro: finite-inverse-image[OF - inj-pi])*

qed

abbreviation *all-formulas-of :: ('p, 'f) DLf-state \Rightarrow 'f set where*

all-formulas-of St \equiv passive-formulas-of (passive-of St) \cup set-option (yy-of St) \cup
fset (active-of St)

lemma *passive-inferences-of-empty[simp]: passive-inferences-of empty = $\{\}$*

unfolding *passive-inferences-of-def by simp*

lemma *passive-inferences-of-add-Passive-Inference[simp]:*

passive-inferences-of (add (Passive-Inference ι) P) = $\{\iota\} \cup$ passive-inferences-of P

unfolding *passive-inferences-of-def by auto*

lemma *passive-inferences-of-add-Passive-Formula[simp]:*

passive-inferences-of (add (Passive-Formula C) P) = *passive-inferences-of* P
unfolding *passive-inferences-of-def* **by** auto

lemma *passive-inferences-of-fold-add-Passive-Inference[simp]*:
passive-inferences-of (fold (add ∘ Passive-Inference) ιs P) = *passive-inferences-of* P ∪ set ιs
by (induct ιs arbitrary: P) auto

lemma *passive-inferences-of-fold-add-Passive-Formula[simp]*:
passive-inferences-of (fold (add ∘ Passive-Formula) Cs P) = *passive-inferences-of* P
by (induct Cs arbitrary: P) auto

lemma *passive-inferences-of-remove-Passive-Inference[simp]*:
passive-inferences-of (remove (Passive-Inference ι) P) = *passive-inferences-of* P - {ι}
unfolding *passive-inferences-of-def* **by** auto

lemma *passive-inferences-of-remove-Passive-Formula[simp]*:
passive-inferences-of (remove (Passive-Formula C) P) = *passive-inferences-of* P
unfolding *passive-inferences-of-def* **by** auto

lemma *passive-inferences-of-fold-remove-Passive-Inference[simp]*:
passive-inferences-of (fold (remove ∘ Passive-Inference) ιs P) = *passive-inferences-of* P - set ιs
by (induct ιs arbitrary: P) auto

lemma *passive-inferences-of-fold-remove-Passive-Formula[simp]*:
passive-inferences-of (fold (remove ∘ Passive-Formula) Cs P) = *passive-inferences-of* P
by (induct Cs arbitrary: P) auto

lemma *passive-formulas-of-empty[simp]*: *passive-formulas-of* empty = {}
unfolding *passive-formulas-of-def* **by** simp

lemma *passive-formulas-of-add-Passive-Inference[simp]*:
passive-formulas-of (add (Passive-Inference ι) P) = *passive-formulas-of* P
unfolding *passive-formulas-of-def* **by** auto

lemma *passive-formulas-of-add-Passive-Formula[simp]*:
passive-formulas-of (add (Passive-Formula C) P) = {C} ∪ *passive-formulas-of* P
unfolding *passive-formulas-of-def* **by** auto

lemma *passive-formulas-of-fold-add-Passive-Inference[simp]*:
passive-formulas-of (fold (add ∘ Passive-Inference) ιs P) = *passive-formulas-of* P
by (induct ιs arbitrary: P) auto

lemma *passive-formulas-of-fold-add-Passive-Formula[simp]*:
passive-formulas-of (fold (add ∘ Passive-Formula) Cs P) = *passive-formulas-of* P ∪ set Cs
by (induct Cs arbitrary: P) auto

lemma *passive-formulas-of-remove-Passive-Inference[simp]*:
passive-formulas-of (remove (Passive-Inference ι) P) = *passive-formulas-of* P
unfolding *passive-formulas-of-def* **by** auto

lemma *passive-formulas-of-remove-Passive-Formula[simp]*:
passive-formulas-of (remove (Passive-Formula C) P) = *passive-formulas-of* P - {C}
unfolding *passive-formulas-of-def* **by** auto

lemma *passive-formulas-of-fold-remove-Passive-Inference[simp]*:

passive-formulas-of (fold (remove \circ *Passive-Inference*) ι P) = *passive-formulas-of* P
by (induct ι arbitrary: P) auto

lemma *passive-formulas-of-fold-remove-Passive-Formula[simp]*:
passive-formulas-of (fold (remove \circ *Passive-Formula*) C s P) = *passive-formulas-of* P - set C
by (induct C arbitrary: P) auto

fun *fstate* :: (' p , ' f) *DLf-state* \Rightarrow ' f inference set \times (' f \times *DL-label*) set **where**
fstate (P , Y , A) = state (*passive-inferences-of* P , *passive-formulas-of* P , set-option Y , fset A)

lemma *fstate-alt-def*:
fstate St = state (*passive-inferences-of* (fst St), *passive-formulas-of* (fst St),
 set-option (fst (snd St)), fset (snd (snd St)))
by (cases St) auto

definition *Liminf-fstate* :: (' p , ' f) *DLf-state* llist \Rightarrow ' f set \times ' f set \times ' f set **where**
Liminf-fstate Sts =
 (*Liminf-llist* (lmap (*passive-formulas-of* \circ *passive-of*) Sts),
Liminf-llist (lmap (set-option \circ *yy-of*) Sts),
Liminf-llist (lmap (fset \circ *active-of*) Sts))

lemma *Liminf-fstate-commute*:
Liminf-llist (lmap (snd \circ *fstate*) Sts) = labeled-formulas-of (*Liminf-fstate* Sts)

proof -

have *Liminf-llist* (lmap (snd \circ *fstate*) Sts) =
 (λC . (C , *Passive*)) '*Liminf-llist* (lmap (*passive-formulas-of* \circ *passive-of*) Sts) \cup
 (λC . (C , *YY*)) '*Liminf-llist* (lmap (set-option \circ *yy-of*) Sts) \cup
 (λC . (C , *Active*)) '*Liminf-llist* (lmap (fset \circ *active-of*) Sts)
unfolding *fstate-alt-def* *state-alt-def*
apply *simp*
apply (subst *Liminf-llist-lmap-union*, *fast*)
apply (subst *Liminf-llist-lmap-image*, *simp* add: *inj-on-convol-ident*)
by auto

thus *?thesis*

unfolding *Liminf-fstate-def* **by** *fastforce*

qed

fun *formulas-union* :: ' f set \times ' f set \times ' f set \Rightarrow ' f set **where**
formulas-union (P , Y , A) = $P \cup Y \cup A$

inductive *fair-DL* :: (' p , ' f) *DLf-state* \Rightarrow (' p , ' f) *DLf-state* \Rightarrow bool (**infix** \rightsquigarrow *DLf* 50) **where**
compute-infer: $P \neq \text{empty} \implies \text{select } P = \text{Passive-Inference } \iota \implies$
 $\iota \in \text{no-labels.Red-I (fset } A \cup \{C\}) \implies$
 (P , *None*, A) \rightsquigarrow *DLf* (remove (select P) P , *Some* C , A)
 | *choose-p*: $P \neq \text{empty} \implies \text{select } P = \text{Passive-Formula } C \implies$
 (P , *None*, A) \rightsquigarrow *DLf* (remove (select P) P , *Some* C , A)
 | *delete-fwd*: $C \in \text{no-labels.Red-F (fset } A) \vee (\exists C' \in \text{fset } A. C' \preceq C) \implies$
 (P , *Some* C , A) \rightsquigarrow *DLf* (P , *None*, A)
 | *simplify-fwd*: $C' \prec_S C \implies C \in \text{no-labels.Red-F (fset } A \cup \{C'\}) \implies$
 (P , *Some* C , A) \rightsquigarrow *DLf* (P , *Some* C' , A)
 | *delete-bwd*: $C' \notin A \implies C' \in \text{no-labels.Red-F } \{C\} \vee C' \succ C \implies$
 (P , *Some* C , $A \cup \{C'\}$) \rightsquigarrow *DLf* (P , *Some* C , A)
 | *simplify-bwd*: $C' \notin A \implies C'' \prec_S C' \implies C' \in \text{no-labels.Red-F } \{C, C''\} \implies$
 (P , *Some* C , $A \cup \{C'\}$) \rightsquigarrow *DLf* (add (*Passive-Formula* C'') P , *Some* C , A)
 | *schedule-infer*: set ι = *no-labels.Inf-between* (fset A) $\{C\} \implies$

$(P, \text{Some } C, A) \rightsquigarrow_{DLf} (\text{fold } (\text{add} \circ \text{Passive-Inference}) \text{ } \iota s P, \text{None}, A \cup \{|C\})$
 $\mid \text{delete-orphan-infers: } \iota s \neq [] \implies \text{set } \iota s \subseteq \text{passive-inferences-of } P \implies$
 $\text{set } \iota s \cap \text{no-labels.Inf-from } (\text{fset } A) = \{\} \implies$
 $(P, Y, A) \rightsquigarrow_{DLf} (\text{fold } (\text{remove} \circ \text{Passive-Inference}) \text{ } \iota s P, Y, A)$

5.3 Initial State and Invariant

inductive *is-initial-DLf-state* :: ('p, 'f) DLf-state \Rightarrow bool **where**
is-initial-DLf-state (empty, None, {\})

inductive *DLf-invariant* :: ('p, 'f) DLf-state \Rightarrow bool **where**
passive-inferences-of P \subseteq Inf-F \implies *DLf-invariant* (P, Y, A)

lemma *initial-DLf-invariant*: *is-initial-DLf-state* St \implies *DLf-invariant* St
unfolding *is-initial-DLf-state.simps* *DLf-invariant.simps* **by** auto

lemma *step-DLf-invariant*:

assumes

inv: *DLf-invariant* St **and**

step: St \rightsquigarrow_{DLf} St'

shows *DLf-invariant* St'

using *step inv*

proof *cases*

case (*schedule-infer* ιs A C P)

note *defs* = *this*(1,2) **and** *is-inf-betw* = *this*(3)

have set $\iota s \subseteq$ Inf-F

using *is-inf-betw* **unfolding** *no-labels.Inf-between-def* *no-labels.Inf-from-def* **by** auto

thus *?thesis*

using *inv* **unfolding** *defs*

by (*auto simp*: *DLf-invariant.simps* *passive-inferences-of-def* *fold-map[symmetric]*)

qed (*auto simp*: *DLf-invariant.simps* *passive-inferences-of-def* *fold-map[symmetric]*)

lemma *chain-DLf-invariant-lnth*:

assumes

chain: *chain* (\rightsquigarrow_{DLf}) Sts **and**

fair-hd: *DLf-invariant* (lhd Sts) **and**

i-lt: *enat* i < *llength* Sts

shows *DLf-invariant* (lnth Sts i)

using *i-lt*

proof (*induct* i)

case 0

thus *?case*

using *fair-hd* *lhd-conv-lnth* *zero-enat-def* **by** *fastforce*

next

case (*Suc* i)

note *ih* = *this*(1) **and** *si-lt* = *this*(2)

have *enat* i < *llength* Sts

using *si-lt* *Suc-ile-eq* *nless-le* **by** *blast*

hence *inv-i*: *DLf-invariant* (lnth Sts i)

by (*rule* *ih*)

have *step*: *lnth* Sts i \rightsquigarrow_{DLf} *lnth* Sts (*Suc* i)

using *chain* *chain-lnth-rel* *si-lt* **by** *blast*

show *?case*

by (*rule* *step-DLf-invariant[OF inv-i step]*)

qed

lemma *chain-DLf-invariant-llast*:

assumes

chain: *chain* (\rightsquigarrow DLf) *Sts* **and**

fair-hd: DLf-invariant (*lhd Sts*) **and**

fin: lfinite *Sts*

shows DLf-invariant (*llast Sts*)

proof –

obtain *i* :: nat **where**

i: *llength Sts* = *enat i*

using lfinite-llength-enat[OF *fin*] **by** *blast*

have *im1-lt*: *enat (i - 1)* < *llength Sts*

by (*metis chain chain-length-pos diff-less enat-ord-simps(2) i zero-enat-def zero-less-one*)

show ?thesis

using *chain-DLf-invariant-lnth*[OF *chain fair-hd im1-lt*]

by (*metis Suc-diff-1 chain chain-length-pos eSuc-enat enat-ord-simps(2) i llast-conv-lnth zero-enat-def*)

qed

5.4 Final State

inductive *is-final-DLf-state* :: ('p, 'f) DLf-state \Rightarrow bool **where**

is-final-DLf-state (*empty, None, A*)

lemma *is-final-DLf-state-iff-no-DLf-step*:

assumes *inv*: DLf-invariant *St*

shows *is-final-DLf-state St* \longleftrightarrow ($\forall St'. \neg St \rightsquigarrow$ DLf *St'*)

proof

assume *is-final-DLf-state St*

then obtain *A* :: 'f fset **where**

st: *St* = (*empty, None, A*)

by (*auto simp: is-final-DLf-state.simps*)

show $\forall St'. \neg St \rightsquigarrow$ DLf *St'*

unfolding *st*

proof (*intro allI notI*)

fix *St'*

assume (*empty, None, A*) \rightsquigarrow DLf *St'*

thus *False*

by *cases auto*

qed

next

assume *no-step*: $\forall St'. \neg St \rightsquigarrow$ DLf *St'*

show *is-final-DLf-state St*

proof (*rule ccontr*)

assume *not-fin*: \neg *is-final-DLf-state St*

obtain *P* :: 'p **and** *Y* :: 'f option **and** *A* :: 'f fset **where**

st: *St* = (*P, Y, A*)

by (*cases St*)

have *P* \neq *empty* \vee *Y* \neq *None*

using *not-fin* **unfolding** *st is-final-DLf-state.simps* **by** *auto*

moreover {


```

assume
  p: P ≠ empty and
  y: Y = None

have ∃ St'. St ∼DLf St'
proof (cases select P)
  case sel: (Passive-Inference ι)
  hence ι-inf: ι ∈ Inf-F
    using inv p unfolding st by (metis DLf-invariant.cases fst-conv mem-Collect-eq
      passive-inferences-of-def select-in-felems subset-iff)
  have ι-red: ι ∈ no-labels.Red-I-G (fset A ∪ {concl-of ι})
    using ι-inf no-labels.empty-ord.Red-I-of-Inf-to-N by auto
  show ?thesis
    using fair-DL.compute-infer[OF p sel ι-red] unfolding st p y by blast
next
  case (Passive-Formula C)
  then show ?thesis
    using fair-DL.choose-p[OF p] unfolding st p y by fast
qed
} moreover {
assume Y ≠ None
then obtain C :: 'f where
  y: Y = Some C
  by blast

have fin: finite (no-labels.Inf-between (fset A) {C})
  by (rule finite-Inf-between[of fset A, simplified])
obtain ιs :: 'f inference list where
  ιs: set ιs = no-labels.Inf-between (fset A) {C}
  using finite-imp-set-eq[OF fin] by blast

have ∃ St'. St ∼DLf St'
  using fair-DL.schedule-infer[OF ιs] unfolding st y by fast
} ultimately show False
using no-step by force
qed
qed

```

5.5 Refinement

lemma *fair-DL-step-imp-DL-step*:

```

assumes dlf: (P, Y, A) ∼DLf (P', Y', A')
shows fstate (P, Y, A) ∼DL fstate (P', Y', A')
using dlf

```

proof *cases*

```

case (compute-infer ι C)

```

```

note defs = this(1-4) and p-nemp = this(5) and sel = this(6) and ι-red = this(7)

```

```

have pas-min-ι-uni-ι: passive-inferences-of P - {ι} ∪ {ι} = passive-inferences-of P
  by (metis Un-insert-right insert-Diff-single insert-absorb mem-Collect-eq p-nemp
    passive-inferences-of-def sel select-in-felems sup-bot.right-neutral)

```

show ?*thesis*

```

unfolding defs fstate-alt-def

```

```

using DL.compute-infer[OF ι-red,

```

```

  of passive-inferences-of (remove (select P) P) passive-formulas-of P]

```

by (*simp only: sel prod.sel option.set passive-inferences-of-remove-Passive-Inference*
passive-formulas-of-remove-Passive-Inference pas-min-ι-uni-ι)

next
 case (*choose-p C*)
 note *defs = this(1-4)* and *p-nemp = this(5)* and *sel = this(6)*

have *pas-min-c-uni-c: passive-formulas-of P - {C} ∪ {C} = passive-formulas-of P*
 by (*metis Un-insert-right insert-Diff mem-Collect-eq p-nemp passive-formulas-of-def sel*
select-in-felems sup-bot.right-neutral)

show ?thesis
unfolding *defs fstate-alt-def*
using *DL.choose-p[of passive-inferences-of P passive-formulas-of (remove (select P) P) C*
fset A]
unfolding *sel* by (*simp only: prod.sel option.set passive-formulas-of-remove-Passive-Formula*
passive-inferences-of-remove-Passive-Formula pas-min-c-uni-c)

next
 case (*delete-fwd C*)
 note *defs = this(1-4)* and *c-red = this(5)*
show ?thesis
unfolding *defs fstate-alt-def* **using** *DL.delete-fwd[OF c-red]* by *simp*

next
 case (*simplify-fwd C' C*)
 note *defs = this(1-4)* and *c-red = this(6)*
show ?thesis
unfolding *defs fstate-alt-def* **using** *DL.simplify-fwd[OF c-red]* by *simp*

next
 case (*delete-bwd C' C*)
 note *defs = this(1-4)* and *c'-red = this(6)*
show ?thesis
unfolding *defs fstate-alt-def* **using** *DL.delete-bwd[OF c'-red]* by *simp*

next
 case (*simplify-bwd C'' C' C*)
 note *defs = this(1-4)* and *c''-red = this(7)*
show ?thesis
unfolding *defs fstate-alt-def* **using** *DL.simplify-bwd[OF c''-red]* by *simp*

next
 case (*schedule-infer ιs C*)
 note *defs = this(1-4)* and *ιs = this(5)*
show ?thesis
unfolding *defs fstate-alt-def*
using *DL.schedule-infer[OF ιs, of passive-inferences-of P passive-formulas-of P]* by *simp*

next
 case (*delete-orphan-infers ιs*)
 note *defs = this(1-3)* and *ιs-ne = this(4)* and *ιs-pas = this(5)* and *inter = this(6)*

have *pas-min-ιs-uni-ιs: passive-inferences-of P - set ιs ∪ set ιs = passive-inferences-of P*
 by (*simp add: ιs-pas set-eq-subset*)

show ?thesis
unfolding *defs fstate-alt-def*
using *DL.delete-orphan-infers[OF inter,*
of passive-inferences-of (fold (remove ∘ Passive-Inference) ιs P)
passive-formulas-of P set-option Y]
 by (*simp only: prod.sel passive-inferences-of-fold-remove-Passive-Inference*)

passive-formulas-of-fold-remove-Passive-Inference pas-min-ts-uni-ts)
qed

lemma *fair-DL-step-imp-GC-step*:

$(P, Y, A) \rightsquigarrow_{DLf} (P', Y', A') \implies \text{fstate } (P, Y, A) \rightsquigarrow_{LGC} \text{fstate } (P', Y', A')$
by (*rule DL-step-imp-LGC-step[OF fair-DL-step-imp-DL-step]*)

5.6 Completeness

fun *mset-of-fstate* :: $(\text{'p}, \text{'f}) \text{DLf-state} \Rightarrow \text{'f multiset}$ **where**

mset-of-fstate $(P, Y, A) =$
image-mset concl-of (*mset-set* (*passive-inferences-of* P)) + *mset-set* (*passive-formulas-of* P) +
mset-set (*set-option* Y) + *mset-set* (*fset* A)

abbreviation *Precprec-S* :: $\text{'f multiset} \Rightarrow \text{'f multiset} \Rightarrow \text{bool}$ (**infix** $\prec\prec S$ 50) **where**

$(\prec\prec S) \equiv \text{multp } (\prec S)$

lemma *wfP-Precprec-S*: $\text{wfP } (\prec\prec S)$

using *minimal-element-def wfP-multp wf-Prec-S wfp-on-UNIV* **by** *blast*

definition *Less-state* :: $(\text{'p}, \text{'f}) \text{DLf-state} \Rightarrow (\text{'p}, \text{'f}) \text{DLf-state} \Rightarrow \text{bool}$ (**infix** \sqsubset 50) **where**

$St' \sqsubset St \iff$
 $(\text{yy-of } St' = \text{None} \wedge \text{yy-of } St \neq \text{None})$
 $\vee ((\text{yy-of } St' = \text{None} \iff \text{yy-of } St = \text{None}) \wedge \text{mset-of-fstate } St' \prec\prec S \text{mset-of-fstate } St)$

lemma *wfP-Less-state*: $\text{wfP } (\sqsubset)$

proof –

let *?boolset* = $\{(b', b :: \text{bool}). b' < b\}$
let *?msetset* = $\{(M', M). M' \prec\prec S M\}$
let *?pair-of* = $\lambda St. (\text{yy-of } St \neq \text{None}, \text{mset-of-fstate } St)$

have *wf-boolset*: $\text{wf } ?\text{boolset}$

by (*rule Wellfounded.wellorder-class.wf*)

have *wf-msetset*: $\text{wf } ?\text{msetset}$

using *wfP-Precprec-S wfP-def* **by** *auto*

have *wf-lex-prod*: $\text{wf } (??\text{boolset} <*\text{lex}*> ?\text{msetset})$

by (*rule wf-lex-prod[OF wf-boolset wf-msetset]*)

have *Less-state-alt-def*:

$\bigwedge St' St. St' \sqsubset St \iff (??\text{pair-of } St', ?\text{pair-of } St) \in ?\text{boolset} <*\text{lex}*> ?\text{msetset}$

unfolding *Less-state-def* **by** *auto*

show *?thesis*

unfolding *wfP-def Less-state-alt-def* **using** *wf-app[of - ?pair-of] wf-lex-prod* **by** *blast*

qed

lemma *non-compute-infer-choose-p-DLf-step-imp-Less-state*:

assumes

step: $St \rightsquigarrow_{DLf} St'$ **and**

yy: $\text{yy-of } St \neq \text{None} \vee \text{yy-of } St' = \text{None}$

shows $St' \sqsubset St$

using *step*

proof *cases*

case (*compute-infer* $P \iota A C$)

note *defs* = *this*(1,2)

have *False*

```

    using step yy unfolding defs by simp
  thus ?thesis
    by blast
next
case (choose-p P C A)
note defs = this(1,2)
have False
  using step yy unfolding defs by simp
thus ?thesis
  by blast
next
case (delete-fwd C A P)
note defs = this(1,2)
show ?thesis
  unfolding defs Less-state-def by (auto intro!: subset-implies-multip)
next
case (simplify-fwd C' C A P)
note defs = this(1,2) and prec = this(3)

let ?new-bef = image-mset concl-of (mset-set (passive-inferences-of P)) +
  mset-set (passive-formulas-of P) + mset-set (fset A) + {#C#}
let ?new-aft = image-mset concl-of (mset-set (passive-inferences-of P)) +
  mset-set (passive-formulas-of P) + mset-set (fset A) + {#C'#}

have lt-new: ?new-aft <<S ?new-bef
  unfolding multp-def
proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
  show {#C'#} <<S {#C#}
    unfolding multp-def using prec by (auto intro!: singletons-in-mult)
qed
thus ?thesis
  unfolding defs Less-state-def by simp
next
case (delete-bwd C' A C P)
note defs = this(1,2) and c-ni = this(3)
show ?thesis
  unfolding defs Less-state-def using c-ni
  by (auto intro!: subset-implies-multip)
next
case (simplify-bwd C' A C'' C P)
note defs = this(1,2) and c'-ni = this(3) and prec = this(4)

show ?thesis
proof (cases C'' ∈ passive-formulas-of P)
  case c''-in: True
  show ?thesis
    unfolding defs Less-state-def using c'-ni
    by (auto simp: insert-absorb[OF c''-in] intro!: subset-implies-multip)
next
case c''-ni: False

have bef: add-mset C (image-mset concl-of (mset-set (passive-inferences-of P)) +
  mset-set (passive-formulas-of P) + mset-set (insert C' (fset A))) =
  add-mset C
  (image-mset concl-of (mset-set (passive-inferences-of P)) +

```

```

      mset-set (passive-formulas-of P) + mset-set (fset A) + {#C'#} (is ?old-bef = ?new-bef)
using c'-ni by auto
have aft: add-mset C
  (image-mset concl-of (mset-set (passive-inferences-of P)) +
   mset-set (insert C'' (passive-formulas-of P)) + mset-set (fset A)) =
add-mset C
  (image-mset concl-of (mset-set (passive-inferences-of P)) +
   mset-set (passive-formulas-of P) + mset-set (fset A) + {#C''#}) (is ?old-aft = ?new-aft)
using c''-ni by (simp add: finite-passive-formulas-of)

have lt-new: ?new-aft <-<S ?new-bef
  unfolding multp-def
proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
  show {#C''#} <-<S {#C'#}
    unfolding multp-def using prec by (auto intro: singletons-in-mult)
qed
show ?thesis
  unfolding defs Less-state-def by simp (simp only: bef aft lt-new)
qed
next
case (schedule-infer  $\iota$  S A C P)
note defs = this(1,2)
show ?thesis
  unfolding defs Less-state-def by auto
next
case (delete-orphan-infers  $\iota$  S P A Y)
note defs = this(1,2) and  $\iota$ S-nnil = this(3) and  $\iota$ S-sub = this(4) and  $\iota$ S-inter = this(5)
have image-mset concl-of (mset-set (passive-inferences-of P - set  $\iota$ S))  $\subset$ #
  image-mset concl-of (mset-set (passive-inferences-of P))
by (metis Diff-empty Diff-subset  $\iota$ S-nnil  $\iota$ S-sub double-diff empty-subsetI
  finite-passive-inferences-of finite-subset image-mset-subset-mono mset-set-eq-iff set-empty
  subset-imp-msubset-mset-set subset-mset.nless-le)
thus ?thesis
  unfolding defs Less-state-def by (auto intro!: subset-implies-multp)
qed

lemma yy-nonempty-DLf-step-imp-Less-state:
assumes
  step: St  $\rightsquigarrow$ DLf St' and
  yy: yy-of St  $\neq$  None and
  yy': yy-of St'  $\neq$  None
shows St'  $\sqsubset$  St
proof -
have yy-of St  $\neq$  None  $\vee$  yy-of St' = None
  using yy by blast
thus ?thesis
  using non-compute-infer-choose-p-DLf-step-imp-Less-state[OF step] by blast
qed

lemma fair-DL-Liminf-yy-empty:
assumes
  len: llength Sts =  $\infty$  and
  full: full-chain ( $\rightsquigarrow$ DLf) Sts and
  inv: DLf-invariant (lhd Sts)
shows Liminf-list (lmap (set-option  $\circ$  yy-of) Sts) = {}

```

proof (*rule ccontr*)

assume *lim-nemp*: $\text{Liminf-llist} (\text{lmap} (\text{set-option} \circ \text{yy-of}) \text{ Sts}) \neq \{\}$

obtain *i* :: *nat* **where**

i-lt: $\text{enat } i < \text{llength } \text{Sts}$ **and**

inter-nemp: $\bigcap ((\text{set-option} \circ \text{yy-of} \circ \text{lth } \text{Sts}) \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } \text{Sts}\}) \neq \{\}$

using *lim-nemp* **unfolding** *Liminf-llist-def* **by** *auto*

from *inter-nemp* **obtain** *C* :: '*f* **where**

c-in: $\forall P \in \text{lth } \text{Sts} \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } \text{Sts}\}. C \in \text{set-option} (\text{yy-of } P)$

by *auto*

hence *c-in'*: $\forall j \geq i. \text{enat } j < \text{llength } \text{Sts} \longrightarrow C \in \text{set-option} (\text{yy-of} (\text{lth } \text{Sts } j))$

by *auto*

have *si-lt*: $\text{enat} (\text{Suc } i) < \text{llength } \text{Sts}$

unfolding *len* **by** *auto*

have *yy-j*: $\text{yy-of} (\text{lth } \text{Sts } j) \neq \text{None}$ **if** *j-ge*: $j \geq i$ **for** *j*

using *c-in'* *len* *j-ge* **by** *auto*

hence *yy-sj*: $\text{yy-of} (\text{lth } \text{Sts} (\text{Suc } j)) \neq \text{None}$ **if** *j-ge*: $j \geq i$ **for** *j*

using *le-Suc-eq* *that* **by** *presburger*

have *step*: $\text{lth } \text{Sts } j \rightsquigarrow \text{DLf } \text{lth } \text{Sts} (\text{Suc } j)$ **if** *j-ge*: $j \geq i$ **for** *j*

using *full-chain-imp-chain[OF full]* *infinite-chain-lth-rel* *len* *llength-eq-infty-conv-lfinite*

by *blast*

have $\text{lth } \text{Sts} (\text{Suc } j) \sqsubseteq \text{lth } \text{Sts } j$ **if** *j-ge*: $j \geq i$ **for** *j*

using *yy-nonempty-DLf-step-imp-Less-state* **by** (*meson* *step* *j-ge* *yy-j* *yy-sj*)

hence $(\sqsubseteq)^{-1-1} (\text{lth } \text{Sts } j) (\text{lth } \text{Sts} (\text{Suc } j))$ **if** *j-ge*: $j \geq i$ **for** *j*

using *j-ge* **by** *blast*

hence *inf-down-chain*: $\text{chain} (\sqsubseteq)^{-1-1} (\text{ldropn } i \text{ Sts})$

by (*simp* *add*: *chain-ldropnI* *si-lt*)

have *inf-i*: $\neg \text{lfinite} (\text{ldropn } i \text{ Sts})$

using *len* **by** (*simp* *add*: *llength-eq-infty-conv-lfinite*)

show *False*

using *inf-i* *inf-down-chain* *wfP-iff-no-infinite-down-chain-llist[of (\sqsubseteq)]* *wfP-Less-state*

by *metis*

qed

lemma *DLf-step-imp-queue-step*:

assumes $\text{St} \rightsquigarrow \text{DLf } \text{St}'$

shows *queue-step* (*passive-of* *St*) (*passive-of* *St'*)

using *assms*

by *cases* (*auto* *simp*: *fold-map[symmetric]* *intro*: *queue-step-idleI* *queue-step-addI*

queue-step-removeI *queue-step-fold-addI* *queue-step-fold-removeI*)

lemma *fair-DL-Liminf-passive-empty*:

assumes

len: $\text{llength } \text{Sts} = \infty$ **and**

full: $\text{full-chain} (\rightsquigarrow \text{DLf}) \text{ Sts}$ **and**

init: *is-initial-DLf-state* (*lhd* *Sts*)

shows $\text{Liminf-llist} (\text{lmap} (\text{elems} \circ \text{passive-of}) \text{ Sts}) = \{\}$

proof –

have *chain-step*: $\text{chain } \text{queue-step} (\text{lmap } \text{passive-of } \text{Sts})$

```

using DLf-step-imp-queue-step chain-lmap full-chain-imp-chain[OF full]
by (metis (no-types, lifting))

have inf-oft: infinitely-often select-queue-step (lmap passive-of Sts)
proof
  assume finitely-often select-queue-step (lmap passive-of Sts)
  then obtain i :: nat where
    no-sel:
       $\forall j \geq i. \neg \text{select-queue-step (passive-of (lnth Sts j)) (passive-of (lnth Sts (Suc j)))}$ 
    by (metis (no-types, lifting) enat-ord-code(4) finitely-often-def len llength-lmap lnth-lmap)

  have si-lt: enat (Suc i) < llength Sts
    unfolding len by auto

  have step: lnth Sts j  $\rightsquigarrow$  DLf lnth Sts (Suc j) if j-ge: j  $\geq$  i for j
    using full-chain-imp-chain[OF full] infinite-chain-lnth-rel len llength-eq-infty-conv-lfinite
    by blast

  have yy: yy-of (lnth Sts j)  $\neq$  None  $\vee$  yy-of (lnth Sts (Suc j)) = None if j-ge: j  $\geq$  i for j
    using step[OF j-ge]
  proof cases
    case (compute-infer P  $\iota$  A C)
      note defs = this(1,2) and p-ne = this(3)
      have False
        using no-sel defs p-ne select-queue-stepI that by fastforce
      thus ?thesis
        by blast
    next
      case (choose-p P C A)
        note defs = this(1,2) and p-ne = this(3)
        have False
          using no-sel defs p-ne select-queue-stepI that by fastforce
        thus ?thesis
          by blast
  qed auto

  have lnth Sts (Suc j)  $\sqsubset$  lnth Sts j if j-ge: j  $\geq$  i for j
    by (rule non-compute-infer-choose-p-DLf-step-imp-Less-state[OF step[OF j-ge] yy[OF j-ge]])
  hence ( $\sqsubset$ )-1-1 (lnth Sts j) (lnth Sts (Suc j)) if j-ge: j  $\geq$  i for j
    using j-ge by blast
  hence inf-down-chain: chain ( $\sqsubset$ )-1-1 (ldropn i Sts)
    using chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id] by simp

  have inf-i:  $\neg$  lfinite (ldropn i Sts)
    using len lfinite-ldropn llength-eq-infty-conv-lfinite by blast

  show False
    using inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of ( $\sqsubset$ )] wfP-Less-state
    by blast
qed

have hd-emp: lhd (lmap passive-of Sts) = empty
  using init full full-chain-not-lnull unfolding is-initial-DLf-state.simps by fastforce

have Liminf-llist (lmap elems (lmap passive-of Sts)) = {}

```

by (rule fair[of lmap passive-of Sts, OF chain-step inf-oft hd-emp])
 thus ?thesis
 by (simp add: llist.map-comp)
 qed

lemma fair-DL-Liminf-passive-formulas-empty:

assumes
 len: llength Sts = ∞ **and**
 full: full-chain (\rightsquigarrow DLf) Sts **and**
 init: is-initial-DLf-state (lhd Sts)
shows Liminf-llist (lmap (passive-formulas-of \circ passive-of) Sts) = {}
proof –
have lim-filt: Liminf-llist (lmap (Set.filter is-passive-formula \circ elems \circ passive-of) Sts) = {}
using fair-DL-Liminf-passive-empty Liminf-llist-subset
by (metis (no-types) empty-iff full init len llength-lmap llist.map-comp lnth-lmap member-filter subsetI subset-antisym)

let ?g = Set.filter is-passive-formula \circ elems \circ passive-of

have inj-on passive-formula (Set.filter is-passive-formula (UNIV :: 'f passive-*elem* set))
unfolding inj-on-def **by** (metis member-filter passive-*elem*.collapse(2))
moreover **have** Sup-llist (lmap ?g Sts) \subseteq Set.filter is-passive-formula UNIV
unfolding Sup-llist-def **by** auto
ultimately **have** inj-pi: inj-on passive-formula (Sup-llist (lmap ?g Sts))
using inj-on-subset **by** blast

have lim-pass: Liminf-llist (lmap (λx . passive-formula ‘
 (Set.filter is-passive-formula \circ elems \circ passive-of) x) Sts) = {}
using Liminf-llist-lmap-image[OF inj-pi] lim-filt **by** simp

have Liminf-llist (lmap (λSt . {C. Passive-Formula C \in elems (passive-of St)}) Sts) = {}
using lim-pass passive-formula-filter **by** (smt (verit) Collect-cong comp-apply llist.map-cong)

thus ?thesis
unfolding passive-formulas-of-def comp-apply .
 qed

lemma fair-DL-Liminf-passive-inferences-empty:

assumes
 len: llength Sts = ∞ **and**
 full: full-chain (\rightsquigarrow DLf) Sts **and**
 init: is-initial-DLf-state (lhd Sts)
shows Liminf-llist (lmap (passive-inferences-of \circ passive-of) Sts) = {}
proof –
have lim-filt: Liminf-llist (lmap (Set.filter is-passive-inference \circ elems \circ passive-of) Sts) = {}
using fair-DL-Liminf-passive-empty Liminf-llist-subset
by (metis (no-types) empty-iff full init len llength-lmap llist.map-comp lnth-lmap member-filter subsetI subset-antisym)

let ?g = Set.filter is-passive-inference \circ elems \circ passive-of

have inj-on passive-inference (Set.filter is-passive-inference (UNIV :: 'f passive-*elem* set))
unfolding inj-on-def **by** (metis member-filter passive-*elem*.collapse(1))
moreover **have** Sup-llist (lmap ?g Sts) \subseteq Set.filter is-passive-inference UNIV
unfolding Sup-llist-def **by** auto
ultimately **have** inj-pi: inj-on passive-inference (Sup-llist (lmap ?g Sts))

using *inj-on-subset* **by** *blast*

have *lim-pass*: $\text{Liminf-llist } (\text{lmap } (\lambda x. \text{passive-inference } (\text{Set.filter is-passive-inference } \circ \text{elems } \circ \text{passive-of }) x) \text{ Sts}) = \{\}$
using *Liminf-llist-lmap-image*[*OF inj-pi*] *lim-filt* **by** *simp*

have *Liminf-llist* $(\text{lmap } (\lambda \text{St}. \{ \iota. \text{Passive-Inference } \iota \in \text{elems } (\text{passive-of } \text{St}) \}) \text{ Sts}) = \{\}$
using *lim-pass* *passive-inference-filter* **by** (*smt* (*verit*) *Collect-cong comp-apply llist.map-cong*)
thus *?thesis*
unfolding *passive-inferences-of-def comp-apply* .
qed

theorem

assumes

full: *full-chain* $(\rightsquigarrow \text{DLf}) \text{ Sts}$ **and**

init: *is-initial-DLf-state* (*lhd Sts*)

shows

fair-DL-Liminf-saturated: *saturated* (*labeled-formulas-of* (*Liminf-fstate Sts*)) **and**

fair-DL-complete-Liminf: $B \in \text{Bot-F} \implies \text{passive-formulas-of } (\text{passive-of } (\text{lhd } \text{Sts})) \models_{\cap \mathcal{G}} \{B\} \implies$
 $\exists B' \in \text{Bot-F}. B' \in \text{formulas-union } (\text{Liminf-fstate } \text{Sts})$ **and**

fair-DL-complete: $B \in \text{Bot-F} \implies \text{passive-formulas-of } (\text{passive-of } (\text{lhd } \text{Sts})) \models_{\cap \mathcal{G}} \{B\} \implies$
 $\exists i. \text{enat } i < \text{llength } \text{Sts} \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of } (\text{lnth } \text{Sts } i))$

proof –

have *chain*: *chain* $(\rightsquigarrow \text{DLf}) \text{ Sts}$

by (*rule full-chain-imp-chain*[*OF full*])

hence *dl-chain*: *chain* $(\rightsquigarrow \text{DL}) (\text{lmap } \text{fstate } \text{Sts})$

by (*smt* (*verit*, *del-insts*) *chain-lmap fair-DL-step-imp-DL-step mset-of-fstate.cases*)

have *inv*: *DLf-invariant* (*lhd Sts*)

using *init initial-DLf-invariant* **by** *auto*

have *nnul*: $\neg \text{lnull } \text{Sts}$

using *chain chain-not-lnull* **by** *blast*

hence *lhd-lmap*: $\bigwedge f. \text{lhd } (\text{lmap } f \text{ Sts}) = f (\text{lhd } \text{Sts})$

by (*rule llist.map-sel*(1))

have *active-of* (*lhd Sts*) = $\{\|\}$

by (*metis is-initial-DLf-state.cases init snd-conv*)

hence *act*: *active-subset* (*snd* (*lhd* (*lmap* *fstate* *Sts*))) = $\{\}$

unfolding *active-subset-def lhd-lmap* **by** (*cases lhd Sts*) *auto*

have *pas-fml-and-t-inf*: *passive-subset* (*Liminf-llist* (*lmap* (*snd* \circ *fstate*) *Sts*)) = $\{\}$ \wedge

Liminf-llist (*lmap* (*fst* \circ *fstate*) *Sts*) = $\{\}$ (**is** *?pas-fml* \wedge *?t-inf*)

proof (*cases lfinite Sts*)

case *fin*: *True*

have *lim-fst*: *Liminf-llist* (*lmap* (*fst* \circ *fstate*) *Sts*) = *fst* (*fstate* (*llast* *Sts*)) **and**

lim-snd: *Liminf-llist* (*lmap* (*snd* \circ *fstate*) *Sts*) = *snd* (*fstate* (*llast* *Sts*))

using *lfinite-Liminf-llist fin nnul*

by (*metis comp-eq-dest-lhs lfinite-lmap llast-lmap llist.map-disc-iff*)+

have *last-inv*: *DLf-invariant* (*llast Sts*)

by (*rule chain-DLf-invariant-llast*[*OF chain inv fin*])

have $\forall \text{St}'. \neg \text{llast } \text{Sts} \rightsquigarrow \text{DLf } \text{St}'$

```

  using full-chain-lnth-not-rel[OF full] by (metis fin full-chain-iff-chain full)
hence is-final-DLf-state (llast Sts)
  unfolding is-final-DLf-state-iff-no-DLf-step[OF last-inv] .
then obtain A :: 'f fset where
  at-l: llast Sts = (empty, None, A)
  unfolding is-final-DLf-state.simps by blast

have ?pas-fml
  unfolding passive-subset-def lim-snd at-l by auto
moreover have ?t-inf
  unfolding lim-fst at-l by simp
ultimately show ?thesis
  by blast
next
case False
hence len: llength Sts = ∞
  by (simp add: not-lfinite-llength)

have ?pas-fml
  unfolding Liminf-fstate-commute passive-subset-def Liminf-fstate-def
  using fair-DL-Liminf-passive-formulas-empty[OF len full init]
  fair-DL-Liminf-yy-empty[OF len full inv]
  by simp
moreover have ?t-inf
  unfolding fstate-alt-def using fair-DL-Liminf-passive-inferences-empty[OF len full init]
  by simp
ultimately show ?thesis
  by blast
qed
note pas-fml = pas-fml-and-t-inf[THEN conjunct1] and
  t-inf = pas-fml-and-t-inf[THEN conjunct2]

have pas-fml': passive-subset (Liminf-llist (lmap snd (lmap fstate Sts))) = {}
  using pas-fml by (simp add: llist.map-comp)
have t-inf': Liminf-llist (lmap fst (lmap fstate Sts)) = {}
  using t-inf by (simp add: llist.map-comp)

have no-prems-init: ∀ ι ∈ Inf-F. prems-of ι = [] ⟶ ι ∈ fst (lhd (lmap fstate Sts))
  using inf-have-prems by blast

show saturated (labeled-formulas-of (Liminf-fstate Sts))
  using DL-Liminf-saturated[OF dl-chain act pas-fml' no-prems-init t-inf']
  unfolding Liminf-fstate-commute[folded llist.map-comp] .

{
  assume
    bot: B ∈ Bot-F and
    unsat: passive-formulas-of (passive-of (lhd Sts)) ⊨ $\cap$  G {B}

  have unsat': fst ' snd (lhd (lmap fstate Sts)) ⊨ $\cap$  G {B}
    using unsat unfolding lhd-lmap by (cases lhd Sts) (auto intro: no-labels-entails-mono-left)

  show ∃ B' ∈ Bot-F. B' ∈ formulas-union (Liminf-fstate Sts)
    using DL-complete-Liminf[OF dl-chain act pas-fml' no-prems-init t-inf' bot unsat']
    unfolding Liminf-fstate-commute[folded llist.map-comp]

```

```

    by (cases Liminf-fstate Sts) auto
  thus  $\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of } (\text{lth } Sts \ i))$ 
    unfolding Liminf-fstate-def Liminf-llist-def by auto
}
qed

end

```

5.7 Specialization with FIFO Queue

As a proof of concept, we specialize the passive set to use a FIFO queue, thereby eliminating the locale assumptions about the passive set.

```

locale fifo-discount-loop =
  discount-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F
for
  Bot-F :: 'f set and
  Inf-F :: 'f inference set and
  Bot-G :: 'g set and
  Q :: 'q set and
  entails-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set  $\Rightarrow$  bool and
  Inf-G-q :: 'q  $\Rightarrow$  'g inference set and
  Red-I-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g inference set and
  Red-F-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set and
  G-F-q :: 'q  $\Rightarrow$  'f  $\Rightarrow$  'g set and
  G-I-q :: 'q  $\Rightarrow$  'f inference  $\Rightarrow$  'g inference set option and
  Equiv-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix  $\langle \equiv \rangle$  50) and
  Prec-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix  $\langle \prec \cdot \rangle$  50) +
fixes
  Prec-S :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix  $\prec S$  50)
assumes
  wf-Prec-S: minimal-element ( $\prec S$ ) UNIV and
  transp-Prec-S: transp ( $\prec S$ ) and
  finite-Inf-between: finite A  $\implies$  finite (no-labels.Inf-between A {C})
begin

sublocale fifo-prover-queue
  .

sublocale fair-discount-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q
  Equiv-F Prec-F [] hd  $\lambda y \ xs. \text{if } y \in \text{set } xs \text{ then } xs \text{ else } xs @ [y]$  removeAll fset-of-list Prec-S
proof
  show po-on ( $\prec S$ ) UNIV
    using wf-Prec-S minimal-element.po by blast
next
  show wfp-on ( $\prec S$ ) UNIV
    using wf-Prec-S minimal-element.wf by blast
next
  show transp ( $\prec S$ )
    by (rule transp-Prec-S)
next
  show  $\bigwedge A \ C. \text{finite } A \implies \text{finite } (\text{no-labels.Inf-between } A \ \{C\})$ 
    by (fact finite-Inf-between)
qed

end

```

end

6 Otter Loop

The Otter loop is one of the two best-known given clause procedures. It is formalized as an instance of the abstract procedure GC .

theory *Otter-Loop*

imports

More-Given-Clause-Architectures

Given-Clause-Loops-Util

begin

datatype $OL\text{-label} =$

New | *XX* | *Passive* | *YY* | *Active*

primrec $nat\text{-of-}OL\text{-label} :: OL\text{-label} \Rightarrow nat$ **where**

$nat\text{-of-}OL\text{-label } New = 4$

| $nat\text{-of-}OL\text{-label } XX = 3$

| $nat\text{-of-}OL\text{-label } Passive = 2$

| $nat\text{-of-}OL\text{-label } YY = 1$

| $nat\text{-of-}OL\text{-label } Active = 0$

definition $OL\text{-Prec-}L :: OL\text{-label} \Rightarrow OL\text{-label} \Rightarrow bool$ (**infix** $\sqsubset L$ 50) **where**

$OL\text{-Prec-}L \ l \ l' \longleftrightarrow nat\text{-of-}OL\text{-label } l < nat\text{-of-}OL\text{-label } l'$

locale $otter\text{-loop} = labeled\text{-lifting-}intersection \ Bot\text{-}F \ Inf\text{-}F \ Bot\text{-}G \ Q \ entails\text{-}q \ Inf\text{-}G\text{-}q \ Red\text{-}I\text{-}q$

$Red\text{-}F\text{-}q \ \mathcal{G}\text{-}F\text{-}q \ \mathcal{G}\text{-}I\text{-}q$

$\{\iota_{FL} :: (f \times OL\text{-label}) \text{ inference. Infer } (map \text{fst } (prems\text{-of } \iota_{FL})) \text{ (fst } (concl\text{-of } \iota_{FL})) \in Inf\text{-}F\}$

for

$Bot\text{-}F :: 'f \text{ set}$

and $Inf\text{-}F :: 'f \text{ inference set}$

and $Bot\text{-}G :: 'g \text{ set}$

and $Q :: 'g \text{ set}$

and $entails\text{-}q :: 'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ set} \Rightarrow bool$

and $Inf\text{-}G\text{-}q :: \langle 'q \Rightarrow 'g \text{ inference set} \rangle$

and $Red\text{-}I\text{-}q :: 'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ inference set}$

and $Red\text{-}F\text{-}q :: 'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ set}$

and $\mathcal{G}\text{-}F\text{-}q :: 'q \Rightarrow 'f \Rightarrow 'g \text{ set}$

and $\mathcal{G}\text{-}I\text{-}q :: 'q \Rightarrow 'f \text{ inference} \Rightarrow 'g \text{ inference set option}$

+ **fixes**

$Equiv\text{-}F :: 'f \Rightarrow 'f \Rightarrow bool$ (**infix** \doteq 50) **and**

$Prec\text{-}F :: 'f \Rightarrow 'f \Rightarrow bool$ (**infix** \prec 50)

assumes

$equiv\text{-}equiv\text{-}F$: $equivp (\doteq)$ **and**

$wf\text{-}prec\text{-}F$: $minimal\text{-}element (\prec) \ UNIV$ **and**

$compat\text{-}equiv\text{-}prec$: $C1 \doteq D1 \Longrightarrow C2 \doteq D2 \Longrightarrow C1 \prec \cdot C2 \Longrightarrow D1 \prec \cdot D2$ **and**

$equiv\text{-}F\text{-}grounding$: $q \in Q \Longrightarrow C1 \doteq C2 \Longrightarrow \mathcal{G}\text{-}F\text{-}q \ q \ C1 \subseteq \mathcal{G}\text{-}F\text{-}q \ q \ C2$ **and**

$prec\text{-}F\text{-}grounding$: $q \in Q \Longrightarrow C2 \prec \cdot C1 \Longrightarrow \mathcal{G}\text{-}F\text{-}q \ q \ C1 \subseteq \mathcal{G}\text{-}F\text{-}q \ q \ C2$ **and**

$static\text{-}ref\text{-}comp$: $statically\text{-}complete\text{-}calculus \ Bot\text{-}F \ Inf\text{-}F \ (\models \cap \mathcal{G})$

$no\text{-}labels.Red\text{-}I\text{-}\mathcal{G}$ $no\text{-}labels.Red\text{-}F\text{-}\mathcal{G}\text{-}empty$ **and**

$inf\text{-}have\text{-}prems$: $\iota F \in Inf\text{-}F \Longrightarrow prems\text{-of } \iota F \neq []$

begin

lemma *po-on-OL-Prec-L*: *po-on* ($\sqsubset L$) *UNIV*
by (*metis* (*mono-tags*, *lifting*) *OL-Prec-L-def* *irreflp-onI* *less-imp-neq* *order.strict-trans* *po-on-def* *transp-onI*)

lemma *wfp-on-OL-Prec-L*: *wfp-on* ($\sqsubset L$) *UNIV*
unfolding *wfp-on-UNIV* *OL-Prec-L-def* **by** (*simp* *add*: *wfP-app*)

lemma *Active-minimal*: $l2 \neq \text{Active} \implies \text{Active} \sqsubset L l2$
by (*cases* *l2*) (*auto* *simp*: *OL-Prec-L-def*)

lemma *at-least-two-labels*: $\exists l2. \text{Active} \sqsubset L l2$
using *Active-minimal* **by** *blast*

sublocale *gc?*: *given-clause* *Bot-F* *Inf-F* *Bot-G* *Q* *entails-q* *Inf-G-q* *Red-I-q* *Red-F-q* *G-F-q* *G-I-q*
Equiv-F *Prec-F* *OL-Prec-L* *Active*
apply *unfold-locales*
apply (*rule* *equiv-equiv-F*)
apply (*simp* *add*: *minimal-element.po* *wf-prec-F*)
using *minimal-element.wf* *wf-prec-F* **apply** *blast*
apply (*rule* *po-on-OL-Prec-L*)
apply (*rule* *wfp-on-OL-Prec-L*)
apply (*fact* *compat-equiv-prec*)
apply (*fact* *equiv-F-grounding*)
apply (*fact* *prec-F-grounding*)
apply (*fact* *Active-minimal*)
apply (*rule* *at-least-two-labels*)
using *static-ref-comp* *statically-complete-calculus.statically-complete* **apply** *fastforce*
apply (*fact* *inf-have-prems*)
done

notation *gc.step* (**infix** $\rightsquigarrow GC$ 50)

6.1 Basic Definitions and Lemmas

fun *state* :: '*f* *set* \times '*f* *set* \times '*f* *set* \times '*f* *set* \times '*f* *set* \implies ('*f* \times *OL-label*) *set* **where**
state (*N*, *X*, *P*, *Y*, *A*) =
 $\{(C, \text{New}) \mid C. C \in N\} \cup \{(C, \text{XX}) \mid C. C \in X\} \cup \{(C, \text{Passive}) \mid C. C \in P\} \cup$
 $\{(C, \text{YY}) \mid C. C \in Y\} \cup \{(C, \text{Active}) \mid C. C \in A\}$

lemma *state-alt-def*:
state (*N*, *X*, *P*, *Y*, *A*) =
 $(\lambda C. (C, \text{New})) \cdot N \cup (\lambda C. (C, \text{XX})) \cdot X \cup (\lambda C. (C, \text{Passive})) \cdot P \cup (\lambda C. (C, \text{YY})) \cdot Y \cup$
 $(\lambda C. (C, \text{Active})) \cdot A$
by *auto*

inductive *OL* :: ('*f* \times *OL-label*) *set* \implies ('*f* \times *OL-label*) *set* \implies *bool* (**infix** $\rightsquigarrow OL$ 50) **where**
choose-n: $C \notin N \implies \text{state} (N \cup \{C\}, \{\}, P, \{\}, A) \rightsquigarrow OL \text{state} (N, \{C\}, P, \{\}, A)$
| *delete-fwd*: $C \in \text{no-labels.Red-F} (P \cup A) \vee (\exists C' \in P \cup A. C' \preceq C) \implies$
 $\text{state} (N, \{C\}, P, \{\}, A) \rightsquigarrow OL \text{state} (N, \{\}, P, \{\}, A)$
| *simplify-fwd*: $C \in \text{no-labels.Red-F} (P \cup A \cup \{C^\eta\}) \implies$
 $\text{state} (N, \{C\}, P, \{\}, A) \rightsquigarrow OL \text{state} (N, \{C^\eta\}, P, \{\}, A)$
| *delete-bwd-p*: $C' \in \text{no-labels.Red-F} \{C\} \vee C \prec C' \implies$
 $\text{state} (N, \{C\}, P \cup \{C^\eta\}, \{\}, A) \rightsquigarrow OL \text{state} (N, \{C\}, P, \{\}, A)$
| *simplify-bwd-p*: $C' \in \text{no-labels.Red-F} \{C, C''\} \implies$
 $\text{state} (N, \{C\}, P \cup \{C^\eta\}, \{\}, A) \rightsquigarrow OL \text{state} (N \cup \{C''\}, \{C\}, P, \{\}, A)$
| *delete-bwd-a*: $C' \in \text{no-labels.Red-F} \{C\} \vee C \prec C' \implies$

$state(N, \{C\}, P, \{\}, A \cup \{C'\}) \rightsquigarrow_{OL} state(N, \{C\}, P, \{\}, A)$
| *simplify-bwd-a*: $C' \in no-labels.Red-F(\{C, C''\}) \implies$
 $state(N, \{C\}, P, \{\}, A \cup \{C'\}) \rightsquigarrow_{OL} state(N \cup \{C''\}, \{C\}, P, \{\}, A)$
| *transfer*: $state(N, \{C\}, P, \{\}, A) \rightsquigarrow_{OL} state(N, \{\}, P \cup \{C\}, \{\}, A)$
| *choose-p*: $C \notin P \implies state(\{\}, \{\}, P \cup \{C\}, \{\}, A) \rightsquigarrow_{OL} state(\{\}, \{\}, P, \{C\}, A)$
| *infer*: *no-labels.Inf-between* $A \{C\} \subseteq no-labels.Red-I(A \cup \{C\} \cup M) \implies$
 $state(\{\}, \{\}, P, \{C\}, A) \rightsquigarrow_{OL} state(M, \{\}, P, \{\}, A \cup \{C\})$

lemma *prj-state-union-sets* [*simp*]: *fst* ‘ $state(N, X, P, Y, A) = N \cup X \cup P \cup Y \cup A$
using *prj-ft-set-to-f-set-distr-union prj-labeledN-eq-N* **by** *auto*

lemma *active-subset-of-setOfFormulasWithLabelDiffActive*:

$l \neq Active \implies active-subset(\{(C', l)\}) = \{\}$
by (*simp add: active-subset-def*)

lemma *state-add-C-New*: $state(N, X, P, Y, A) \cup \{(C, New)\} = state(N \cup \{C\}, X, P, Y, A)$
by *auto*

lemma *state-add-C-XX*: $state(N, X, P, Y, A) \cup \{(C, XX)\} = state(N, X \cup \{C\}, P, Y, A)$
by *auto*

lemma *state-add-C-Passive*: $state(N, X, P, Y, A) \cup \{(C, Passive)\} = state(N, X, P \cup \{C\}, Y, A)$
by *auto*

lemma *state-add-C-YY*: $state(N, X, P, Y, A) \cup \{(C, YY)\} = state(N, X, P, Y \cup \{C\}, A)$
by *auto*

lemma *state-add-C-Active*: $state(N, X, P, Y, A) \cup \{(C, Active)\} = state(N, X, P, Y, A \cup \{C\})$
by *auto*

lemma *prj-ActiveSubset-of-state*: *fst* ‘ $active-subset(state(N, X, P, Y, A)) = A$
unfolding *active-subset-def* **by** *force*

6.2 Refinement

lemma *chooseN-in-GC*: $state(N \cup \{C\}, \{\}, P, \{\}, A) \rightsquigarrow_{GC} state(N, \{C\}, P, \{\}, A)$

proof –

have *XX-ls-New*: $XX \sqsubset_L New$
by (*simp add: OL-Prec-L-def*)

hence *almost-thesis*:

$state(N, \{\}, P, \{\}, A) \cup \{(C, New)\} \rightsquigarrow_{GC} state(N, \{\}, P, \{\}, A) \cup \{(C, XX)\}$
using *relabel-inactive* **by** *blast*

have *rewrite-left*: $state(N, \{\}, P, \{\}, A) \cup \{(C, New)\} = state(N \cup \{C\}, \{\}, P, \{\}, A)$
using *state-add-C-New* **by** *blast*

moreover **have** *rewrite-right*: $state(N, \{\}, P, \{\}, A) \cup \{(C, XX)\} = state(N, \{C\}, P, \{\}, A)$
using *state-add-C-XX* **by** *auto*

ultimately show *?thesis*

using *almost-thesis rewrite-left rewrite-right* **by** *simp*

qed

lemma *deleteFwd-in-GC*:

assumes $C \in no-labels.Red-F(P \cup A) \vee (\exists C' \in P \cup A. C' \preceq C)$

shows $state(N, \{C\}, P, \{\}, A) \rightsquigarrow_{GC} state(N, \{\}, P, \{\}, A)$

using *assms*

proof

assume *c-in-redf-PA*: $C \in no-labels.Red-F(P \cup A)$

have $P \cup A \subseteq N \cup \{\} \cup P \cup \{\} \cup A$ **by** *auto*
then have $no\text{-}labels.Red\text{-}F (P \cup A) \subseteq no\text{-}labels.Red\text{-}F (N \cup \{\} \cup P \cup \{\} \cup A)$
using $no\text{-}labels.Red\text{-}F\text{-}of\text{-}subset$ **by** *simp*
then have $c\text{-}in\text{-}redf\text{-}NPA: C \in no\text{-}labels.Red\text{-}F (N \cup \{\} \cup P \cup \{\} \cup A)$
using $c\text{-}in\text{-}redf\text{-}PA$ **by** *auto*
have $NPA\text{-}eq\text{-}prj\text{-}state\text{-}NPA: N \cup \{\} \cup P \cup \{\} \cup A = fst' \text{ state } (N, \{\}, P, \{\}, A)$
using $prj\text{-}state\text{-}union\text{-}sets$ **by** *simp*
have $C \in no\text{-}labels.Red\text{-}F (fst' \text{ state } (N, \{\}, P, \{\}, A))$
using $c\text{-}in\text{-}redf\text{-}NPA$ $NPA\text{-}eq\text{-}prj\text{-}state\text{-}NPA$ **by** *fastforce*
then show *?thesis*
using $remove\text{-}redundant\text{-}no\text{-}label$ **by** *auto*
next
assume $\exists C' \in P \cup A. C' \preceq C$
then obtain C' **where** $C' \in P \cup A$ **and** $c'\text{-}le\text{-}c: C' \preceq C$
by *auto*
then have $C' \in P \vee C' \in A$
by *blast*
then show *?thesis*
proof
assume $C' \in P$
then have $c'\text{-}Passive\text{-}in: (C', Passive) \in state (N, \{\}, P, \{\}, A)$
by *simp*
have $Passive \sqsubseteq L XX$
by (*simp add: OL-Prec-L-def*)
then have $state (N, \{\}, P, \{\}, A) \cup \{(C, XX)\} \rightsquigarrow GC \text{ state } (N, \{\}, P, \{\}, A)$
using $remove\text{-}succ\text{-}L$ $c'\text{-}le\text{-}c$ $c'\text{-}Passive\text{-}in$ **by** *blast*
then show *?thesis*
by *auto*
next
assume $C' \in A$
then have $c'\text{-}Active\text{-}in\text{-}state\text{-}NPA: (C', Active) \in state (N, \{\}, P, \{\}, A)$
by *simp*
also have $Active\text{-}ls\text{-}x: Active \sqsubseteq L XX$
using $Active\text{-}minimal$ **by** *simp*
then have $state (N, \{\}, P, \{\}, A) \cup \{(C, XX)\} \rightsquigarrow GC \text{ state } (N, \{\}, P, \{\}, A)$
using $remove\text{-}succ\text{-}L$ $c'\text{-}le\text{-}c$ $Active\text{-}ls\text{-}x$ $c'\text{-}Active\text{-}in\text{-}state\text{-}NPA$ **by** *blast*
then show *?thesis*
by *auto*
qed
qed

lemma *simplifyFwd-in-GC:*

$C \in no\text{-}labels.Red\text{-}F (P \cup A \cup \{C'\}) \implies$
 $state (N, \{C\}, P, \{\}, A) \rightsquigarrow GC \text{ state } (N, \{C'\}, P, \{\}, A)$

proof –

assume $c\text{-}in: C \in no\text{-}labels.Red\text{-}F (P \cup A \cup \{C'\})$

let $?N = state (N, \{\}, P, \{\}, A)$

and $?M = \{(C, XX)\}$ **and** $?M' = \{(C', XX)\}$

have $P \cup A \cup \{C'\} \subseteq fst' (?N \cup ?M')$

by *auto*

then have $no\text{-}labels.Red\text{-}F (P \cup A \cup \{C'\}) \subseteq no\text{-}labels.Red\text{-}F (fst' (?N \cup ?M'))$

using $no\text{-}labels.Red\text{-}F\text{-}of\text{-}subset$ **by** *auto*

then have $C \in no\text{-}labels.Red\text{-}F (fst' (?N \cup ?M'))$

using $c\text{-}in$ **by** *auto*

then have $c\text{-}x\text{-in}$: $(C, XX) \in \text{Red-F } (?N \cup ?M')$
using $\text{no-labels-Red-F-imp-Red-F}$ **by** auto
then have $?M \subseteq \text{Red-F } (?N \cup ?M')$
by auto
then have $\text{active-subset-of-}m'$: $\text{active-subset } ?M' = \{\}$
using $\text{active-subset-of-setOfFormulasWithLabelDiffActive}$ **by** auto
show $?thesis$
using $c\text{-}x\text{-in active-subset-of-}m'$ $\text{process[of - - ?M - ?M']}$ **by** auto
qed

lemma deleteBwdP-in-GC :

assumes $C' \in \text{no-labels.Red-F } \{C\} \vee C \prec C'$
shows $\text{state } (N, \{C\}, P \cup \{C'\}, \{\}, A) \rightsquigarrow_{GC} \text{state } (N, \{C\}, P, \{\}, A)$
using assms

proof

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$
assume $c\text{-}ls\text{-}c'$: $C \prec C'$

have $(C, XX) \in \text{state } (N, \{C\}, P, \{\}, A)$
by simp

then have $?N \cup \{(C', \text{Passive})\} \rightsquigarrow_{GC} ?N$
using $c\text{-}ls\text{-}c'$ remove-succ-F **by** blast

also have $?N \cup \{(C', \text{Passive})\} = \text{state } (N, \{C\}, P \cup \{C'\}, \{\}, A)$
by auto

finally show $?thesis$
by auto

next

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$

assume $c'\text{-in-redf-c}$: $C' \in \text{no-labels.Red-F-G } \{C\}$

have $\{C\} \subseteq \text{fst}' ?N$ **by** auto

then have $\text{no-labels.Red-F } \{C\} \subseteq \text{no-labels.Red-F } (\text{fst}' ?N)$
using $\text{no-labels.Red-F-of-subset}$ **by** auto

then have $C' \in \text{no-labels.Red-F } (\text{fst}' ?N)$
using $c'\text{-in-redf-c}$ **by** blast

then have $?N \cup \{(C', \text{Passive})\} \rightsquigarrow_{GC} ?N$
using $\text{remove-redundant-no-label}$ **by** blast

then show $?thesis$

by $(\text{metis state-add-C-Passive})$

qed

lemma $\text{simplifyBwdP-in-GC}$:

assumes $C' \in \text{no-labels.Red-F } \{C, C''\}$

shows $\text{state } (N, \{C\}, P \cup \{C'\}, \{\}, A) \rightsquigarrow_{GC} \text{state } (N \cup \{C''\}, \{C\}, P, \{\}, A)$

proof –

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$

and $?M = \{(C', \text{Passive})\}$

and $?M' = \{(C'', \text{New})\}$

have $\{C, C''\} \subseteq \text{fst}' (?N \cup ?M')$

by $(\text{smt } (z\beta) \text{ Un-commute Un-empty-left Un-insert-right insert-absorb2}$
 $\text{subset-Un-eq state-add-C-New prj-state-union-sets})$

then have $\text{no-labels.Red-F } \{C, C''\} \subseteq \text{no-labels.Red-F } (\text{fst}' (?N \cup ?M'))$

using $\text{no-labels.Red-F-of-subset}$ **by** auto

then have $C' \in \text{no-labels.Red-F } (\text{fst}' (?N \cup ?M'))$

using assms **by** auto

then have $(C', \text{Passive}) \in \text{Red-F } (?N \cup ?M')$
using *no-labels-Red-F-imp-Red-F* **by** *auto*
then have $\mathcal{M}\text{-in-redf: } ?M \subseteq \text{Red-F } (?N \cup ?M')$ **by** *auto*

have *active-subset-M'*: *active-subset* $?M' = \{\}$
using *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *auto*

have $?N \cup ?M \rightsquigarrow_{GC} ?N \cup ?M'$
using $\mathcal{M}\text{-in-redf}$ *active-subset-M'* *process*[of - - $?M - ?M'$] **by** *auto*
also have $?N \cup \{(C', \text{Passive})\} = \text{state } (N, \{C\}, P \cup \{C\}, \{\}, A)$
by *force*
also have $?N \cup \{(C'', \text{New})\} = \text{state } (N \cup \{C''\}, \{C\}, P, \{\}, A)$
using *state-add-C-New* **by** *blast*
finally show *?thesis*
by *auto*

qed

lemma *deleteBwdA-in-GC*:

assumes $C' \in \text{no-labels.Red-F } \{C\} \vee C \prec \cdot C'$
shows $\text{state } (N, \{C\}, P, \{\}, A \cup \{C'\}) \rightsquigarrow_{GC} \text{state } (N, \{C\}, P, \{\}, A)$
using *assms*

proof

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$
assume *c-ls-c'*: $C \prec \cdot C'$

have $(C, XX) \in \text{state } (N, \{C\}, P, \{\}, A)$
by *simp*

then have $?N \cup \{(C', \text{Active})\} \rightsquigarrow_{GC} ?N$
using *c-ls-c'* *remove-succ-F* **by** *blast*

also have $?N \cup \{(C', \text{Active})\} = \text{state } (N, \{C\}, P, \{\}, A \cup \{C'\})$
by *auto*

finally show $\text{state } (N, \{C\}, P, \{\}, A \cup \{C'\}) \rightsquigarrow_{GC} \text{state } (N, \{C\}, P, \{\}, A)$
by *auto*

next

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$
assume *c'-in-redf-c*: $C' \in \text{no-labels.Red-F-G } \{C\}$

have $\{C\} \subseteq \text{fst}' ?N$
by (*metis Un-commute Un-upper2 le-supI2 prj-state-union-sets*)

then have $\text{no-labels.Red-F } \{C\} \subseteq \text{no-labels.Red-F } (\text{fst}' ?N)$
using *no-labels.Red-F-of-subset* **by** *auto*

then have $C' \in \text{no-labels.Red-F } (\text{fst}' ?N)$
using *c'-in-redf-c* **by** *blast*

then have $?N \cup \{(C', \text{Active})\} \rightsquigarrow_{GC} ?N$
using *remove-redundant-no-label* **by** *auto*

then show *?thesis*
by (*metis state-add-C-Active*)

qed

lemma *simplifyBwdA-in-GC*:

assumes $C' \in \text{no-labels.Red-F } \{C, C''\}$
shows $\text{state } (N, \{C\}, P, \{\}, A \cup \{C'\}) \rightsquigarrow_{GC} \text{state } (N \cup \{C''\}, \{C\}, P, \{\}, A)$

proof –

let $?N = \text{state } (N, \{C\}, P, \{\}, A)$ **and** $?M = \{(C', \text{Active})\}$ **and** $?M' = \{(C'', \text{New})\}$

have $\{C, C''\} \subseteq \text{fst}'(\ ?\mathcal{N} \cup \ ?\mathcal{M}')$
by *simp*
then have *no-labels.Red-F* $\{C, C''\} \subseteq \text{no-labels.Red-F}(\text{fst}'(\ ?\mathcal{N} \cup \ ?\mathcal{M}'))$
using *no-labels.Red-F-of-subset* **by** *auto*
then have $C' \in \text{no-labels.Red-F}(\text{fst}'(\ ?\mathcal{N} \cup \ ?\mathcal{M}'))$
using *assms* **by** *auto*
then have $(C', \text{Active}) \in \text{Red-F}(\ ?\mathcal{N} \cup \ ?\mathcal{M}')$
using *no-labels.Red-F-imp-Red-F* **by** *auto*
then have *M-included*: $\ ?\mathcal{M} \subseteq \text{Red-F}(\ ?\mathcal{N} \cup \ ?\mathcal{M}')$
by *auto*

have *active-subset* $\ ?\mathcal{M}' = \{ \}$
using *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *auto*
then have *state* $(N, \{C\}, P, \{ \}, A) \cup \{(C', \text{Active})\} \rightsquigarrow_{GC} \text{state}(N, \{C\}, P, \{ \}, A) \cup \{(C'', \text{New})\}$
using *M-included process* **where** $\ ?M = \ ?\mathcal{M}$ **and** $\ ?M' = \ ?\mathcal{M}'$ **by** *auto*
then show *?thesis*
by *(metis state-add-C-New state-add-C-Active)*

qed

lemma *transfer-in-GC*: $\text{state}(N, \{C\}, P, \{ \}, A) \rightsquigarrow_{GC} \text{state}(N, \{ \}, P \cup \{C\}, \{ \}, A)$
proof –
let $\ ?\mathcal{N} = \text{state}(N, \{ \}, P, \{ \}, A)$

have *Passive* $\sqsubseteq_L \text{XX}$
by *(simp add: OL-Prec-L-def)*
then have $\ ?\mathcal{N} \cup \{(C, \text{XX})\} \rightsquigarrow_{GC} \ ?\mathcal{N} \cup \{(C, \text{Passive})\}$
using *relabel-inactive* **by** *auto*
then show *?thesis*
by *(metis sup-bot-left state-add-C-XX state-add-C-Passive)*

qed

lemma *chooseP-in-GC*: $\text{state}(\{ \}, \{ \}, P \cup \{C\}, \{ \}, A) \rightsquigarrow_{GC} \text{state}(\{ \}, \{ \}, P, \{C\}, A)$
proof –
let $\ ?\mathcal{N} = \text{state}(\{ \}, \{ \}, P, \{ \}, A)$

have *YY* $\sqsubseteq_L \text{Passive}$
by *(simp add: OL-Prec-L-def)*
moreover have $\text{YY} \neq \text{Active}$
by *simp*
ultimately have $\ ?\mathcal{N} \cup \{(C, \text{Passive})\} \rightsquigarrow_{GC} \ ?\mathcal{N} \cup \{(C, \text{YY})\}$
using *relabel-inactive* **by** *auto*
then show *?thesis*
by *(metis sup-bot-left state-add-C-Passive state-add-C-YY)*

qed

lemma *infer-in-GC*:
assumes *no-labels.Inf-between* $A \ \{C\} \subseteq \text{no-labels.Red-I}(A \cup \{C\} \cup M)$
shows $\text{state}(\{ \}, \{ \}, P, \{C\}, A) \rightsquigarrow_{GC} \text{state}(M, \{ \}, P, \{ \}, A \cup \{C\})$
proof –
let $\ ?\mathcal{M} = \{(C', \text{New}) \mid C'. C' \in M\}$
let $\ ?\mathcal{N} = \text{state}(\{ \}, \{ \}, P, \{ \}, A)$

have *active-subset-of-M*: *active-subset* $\ ?\mathcal{M} = \{ \}$
using *active-subset-def* **by** *auto*

have $A \cup \{C\} \cup M \subseteq (\text{fst}' \ ?\mathcal{N}) \cup \{C\} \cup (\text{fst}' \ ?\mathcal{M})$
by *fastforce*
then have $\text{no-labels.Red-I } (A \cup \{C\} \cup M) \subseteq \text{no-labels.Red-I } ((\text{fst}' \ ?\mathcal{N}) \cup \{C\} \cup (\text{fst}' \ ?\mathcal{M}))$
using *no-labels.empty-ord.Red-I-of-subset* **by** *auto*
moreover have $\text{fst}' \ (\text{active-subset } \ ?\mathcal{N}) = A$
using *prj-ActiveSubset-of-state* **by** *blast*
ultimately have $\text{no-labels.Inf-between } (\text{fst}' \ (\text{active-subset } \ ?\mathcal{N})) \ \{C\} \subseteq$
 $\text{no-labels.Red-I } ((\text{fst}' \ ?\mathcal{N}) \cup \{C\} \cup (\text{fst}' \ ?\mathcal{M}))$
using *assms* **by** *auto*

then have $\ ?\mathcal{N} \cup \{(C, YY)\} \sim_{GC} \ ?\mathcal{N} \cup \{(C, \text{Active})\} \cup \ ?\mathcal{M}$
using *active-subset-of-M prj-fl-set-to-f-set-distr-union step.infer* **by** *force*
also have $\ ?\mathcal{N} \cup \{(C, YY)\} = \text{state } (\{\}, \{\}, P, \{C\}, A)$
by *simp*
also have $\ ?\mathcal{N} \cup \{(C, \text{Active})\} \cup \ ?\mathcal{M} = \text{state } (M, \{\}, P, \{\}, A \cup \{C\})$
by *force*
finally show *?thesis*
by *simp*
qed

theorem *OL-step-imp-GC-step*: $M \sim_{OL} M' \implies M \sim_{GC} M'$

proof (*induction rule: OL.induct*)
case (*choose-n N C P A*)
then show *?case*
using *chooseN-in-GC* **by** *auto*
next
case (*delete-fwd C P A N*)
then show *?case*
using *deleteFwd-in-GC* **by** *auto*
next
case (*simplify-fwd C P A C' N*)
then show *?case*
using *simplifyFwd-in-GC* **by** *auto*
next
case (*delete-bwd-p C' C N P A*)
then show *?case*
using *deleteBwdP-in-GC* **by** *auto*
next
case (*simplify-bwd-p C' C C'' N P A*)
then show *?case*
using *simplifyBwdP-in-GC* **by** *auto*
next
case (*delete-bwd-a C' C N P A*)
then show *?case*
using *deleteBwdA-in-GC* **by** *auto*
next
case (*simplify-bwd-a C' C N P A C''*)
then show *?case*
using *simplifyBwdA-in-GC* **by** *blast*
next
case (*transfer N C P A*)
then show *?case*
using *transfer-in-GC* **by** *auto*
next
case (*choose-p P C A*)

```

then show ?case
  using chooseP-in-GC by auto
next
case (infer A C M P)
then show ?case
  using infer-in-GC by auto
qed

```

6.3 Completeness

theorem

assumes

ol-chain: $\text{chain } (\sim OL) \text{ } Sts$ **and**

act: $\text{active-subset } (lhd \text{ } Sts) = \{\}$ **and**

pas: $\text{passive-subset } (Liminf\text{-l}list \text{ } Sts) = \{\}$

shows

OL-Liminf-saturated: $\text{saturated } (Liminf\text{-l}list \text{ } Sts)$ **and**

OL-complete-Liminf: $B \in Bot\text{-}F \implies \text{fst } 'lhd \text{ } Sts \models \cap \mathcal{G} \{B\} \implies$

$\exists BL \in Bot\text{-}FL. BL \in Liminf\text{-l}list \text{ } Sts$ **and**

OL-complete: $B \in Bot\text{-}F \implies \text{fst } 'lhd \text{ } Sts \models \cap \mathcal{G} \{B\} \implies$

$\exists i. \text{enat } i < \text{l}length \text{ } Sts \wedge (\exists BL \in Bot\text{-}FL. BL \in \text{l}nth \text{ } Sts \ i)$

proof –

have *gc-chain*: $\text{chain } (\sim GC) \text{ } Sts$

using *ol-chain OL-step-imp-GC-step chain-mono* **by** blast

show $\text{saturated } (Liminf\text{-l}list \text{ } Sts)$

using *assms(2) gc.fair-implies-Liminf-saturated gc-chain gc-fair gc-to-red pas* **by** blast

{

assume

bot: $B \in Bot\text{-}F$ **and**

unsat: $\text{fst } 'lhd \text{ } Sts \models \cap \mathcal{G} \{B\}$

show $\exists BL \in Bot\text{-}FL. BL \in Liminf\text{-l}list \text{ } Sts$

by (rule *gc-complete-Liminf[OF gc-chain act pas bot unsat]*)

then show $\exists i. \text{enat } i < \text{l}length \text{ } Sts \wedge (\exists BL \in Bot\text{-}FL. BL \in \text{l}nth \text{ } Sts \ i)$

unfolding *Liminf-l}list-def* **by** auto

}

qed

end

end

7 Definition of Fair Otter Loop

The fair Otter loop assumes that the passive queue is fair and ensures (dynamic) refutational completeness under that assumption. This section contains only the loop's definition.

theory *Fair-Otter-Loop-Def*

imports

Otter-Loop

Prover-Queue

begin

7.1 Locale

type-synonym ($'p, 'f$) *OLf-state* = $'f \text{ fset} \times 'f \text{ option} \times 'p \times 'f \text{ option} \times 'f \text{ fset}$

locale *fair-otter-loop* =

otter-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F + fair-prover-queue empty select add remove felems

for

Bot-F :: $'f \text{ set}$ **and**
Inf-F :: $'f \text{ inference set}$ **and**
Bot-G :: $'g \text{ set}$ **and**
Q :: $'q \text{ set}$ **and**
entails-q :: $'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ set} \Rightarrow \text{bool}$ **and**
Inf-G-q :: $'q \Rightarrow 'g \text{ inference set}$ **and**
Red-I-q :: $'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ inference set}$ **and**
Red-F-q :: $'q \Rightarrow 'g \text{ set} \Rightarrow 'g \text{ set}$ **and**
G-F-q :: $'q \Rightarrow 'f \Rightarrow 'g \text{ set}$ **and**
G-I-q :: $'q \Rightarrow 'f \text{ inference} \Rightarrow 'g \text{ inference set option}$ **and**
Equiv-F :: $'f \Rightarrow 'f \Rightarrow \text{bool}$ (**infix** $\langle \doteq \rangle$ 50) **and**
Prec-F :: $'f \Rightarrow 'f \Rightarrow \text{bool}$ (**infix** $\langle \prec \cdot \rangle$ 50) **and**
empty :: $'p$ **and**
select :: $'p \Rightarrow 'f$ **and**
add :: $'f \Rightarrow 'p \Rightarrow 'p$ **and**
remove :: $'f \Rightarrow 'p \Rightarrow 'p$ **and**
felems :: $'p \Rightarrow 'f \text{ fset} +$

fixes

Prec-S :: $'f \Rightarrow 'f \Rightarrow \text{bool}$ (**infix** \prec_S 50)

assumes

wf-Prec-S: *minimal-element* (\prec_S) *UNIV* **and**
transp-Prec-S: *transp* (\prec_S) **and**
finite-Inf-between: $\text{finite } A \Longrightarrow \text{finite } (\text{no-labels.Inf-between } A \{C\})$

begin

lemma *trans-Prec-S*: $\text{trans } \{(x, y). x \prec_S y\}$

using *transp-Prec-S transp-trans* **by** *blast*

lemma *irreflp-Prec-S*: *irreflp* (\prec_S)

using *minimal-element.wf wfp-imp-irreflp wf-Prec-S wfp-on-UNIV* **by** *blast*

lemma *irrefl-Prec-S*: *irrefl* $\{(x, y). x \prec_S y\}$

by (*metis CollectD case-prod-conv irrefl-def irreflp-Prec-S irreflp-def*)

7.2 Basic Definitions and Lemmas

abbreviation *new-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'f \text{ fset}$ **where**

new-of St $\equiv \text{fst } St$

abbreviation *xx-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'f \text{ option}$ **where**

xx-of St $\equiv \text{fst } (\text{snd } St)$

abbreviation *passive-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'p$ **where**

passive-of St $\equiv \text{fst } (\text{snd } (\text{snd } St))$

abbreviation *yy-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'f \text{ option}$ **where**

yy-of St $\equiv \text{fst } (\text{snd } (\text{snd } (\text{snd } St)))$

abbreviation *active-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'f \text{ fset}$ **where**

active-of St $\equiv \text{snd } (\text{snd } (\text{snd } (\text{snd } St)))$

abbreviation *all-formulas-of* :: $('p, 'f) \text{ OLf-state} \Rightarrow 'f \text{ set}$ **where**

$all\text{-formulas-of } St \equiv fset (new\text{-of } St) \cup set\text{-option } (xx\text{-of } St) \cup elems (passive\text{-of } St) \cup set\text{-option } (yy\text{-of } St) \cup fset (active\text{-of } St)$

fun $fstate :: 'f fset \times 'f option \times 'p \times 'f option \times 'f fset \Rightarrow ('f \times OL\text{-label}) set$ **where**
 $fstate (N, X, P, Y, A) = state (fset N, set\text{-option } X, elems P, set\text{-option } Y, fset A)$

lemma $fstate\text{-alt-def}$:

$fstate St =$
 $state (fset (fst St), set\text{-option } (fst (snd St)), elems (fst (snd (snd St))),$
 $set\text{-option } (fst (snd (snd (snd St))))), fset (snd (snd (snd (snd St))))))$
by $(cases St) auto$

definition

$Liminf\text{-fstate} :: ('p, 'f) OLf\text{-state } llist \Rightarrow 'f set \times 'f set \times 'f set \times 'f set \times 'f set$
where

$Liminf\text{-fstate } Sts =$
 $(Liminf\text{-llist } (lmap (fset \circ new\text{-of}) Sts),$
 $Liminf\text{-llist } (lmap (set\text{-option} \circ xx\text{-of}) Sts),$
 $Liminf\text{-llist } (lmap (elems \circ passive\text{-of}) Sts),$
 $Liminf\text{-llist } (lmap (set\text{-option} \circ yy\text{-of}) Sts),$
 $Liminf\text{-llist } (lmap (fset \circ active\text{-of}) Sts))$

lemma $Liminf\text{-fstate-commute}$: $Liminf\text{-llist } (lmap fstate Sts) = state (Liminf\text{-fstate } Sts)$

proof –

have $Liminf\text{-llist } (lmap fstate Sts) =$
 $(\lambda C. (C, New)) ' Liminf\text{-llist } (lmap (fset \circ new\text{-of}) Sts) \cup$
 $(\lambda C. (C, XX)) ' Liminf\text{-llist } (lmap (set\text{-option} \circ xx\text{-of}) Sts) \cup$
 $(\lambda C. (C, Passive)) ' Liminf\text{-llist } (lmap (elems \circ passive\text{-of}) Sts) \cup$
 $(\lambda C. (C, YY)) ' Liminf\text{-llist } (lmap (set\text{-option} \circ yy\text{-of}) Sts) \cup$
 $(\lambda C. (C, Active)) ' Liminf\text{-llist } (lmap (fset \circ active\text{-of}) Sts)$
unfolding $fstate\text{-alt-def } state\text{-alt-def}$
apply $(subst Liminf\text{-llist-lmap-union}, fast)+$
apply $(subst Liminf\text{-llist-lmap-image}, simp add: inj-on-convol-ident)+$
by $auto$

thus $?thesis$

unfolding $Liminf\text{-fstate-def}$ **by** $fastforce$

qed

fun $state\text{-union} :: 'f set \times 'f set \times 'f set \times 'f set \times 'f set \Rightarrow 'f set$ **where**
 $state\text{-union } (N, X, P, Y, A) = N \cup X \cup P \cup Y \cup A$

inductive $fair\text{-OL} :: ('p, 'f) OLf\text{-state} \Rightarrow ('p, 'f) OLf\text{-state} \Rightarrow bool$ (**infix** $\rightsquigarrow OLf 50$) **where**
 $choose\text{-n}: C \notin N \Longrightarrow (N \cup \{C\}, None, P, None, A) \rightsquigarrow OLf (N, Some C, P, None, A)$
 $| delete\text{-fwd}: C \in no\text{-labels.Red-F } (elems P \cup fset A) \vee (\exists C' \in elems P \cup fset A. C' \preceq C) \Longrightarrow$
 $(N, Some C, P, None, A) \rightsquigarrow OLf (N, None, P, None, A)$
 $| simplify\text{-fwd}: C' \prec_S C \Longrightarrow C \in no\text{-labels.Red-F } (elems P \cup fset A \cup \{C'\}) \Longrightarrow$
 $(N, Some C, P, None, A) \rightsquigarrow OLf (N, Some C', P, None, A)$
 $| delete\text{-bwd-p}: C' \in elems P \Longrightarrow C' \in no\text{-labels.Red-F } \{C\} \vee C \prec \cdot C' \Longrightarrow$
 $(N, Some C, P, None, A) \rightsquigarrow OLf (N, Some C, remove C' P, None, A)$
 $| simplify\text{-bwd-p}: C'' \prec_S C' \Longrightarrow C' \in elems P \Longrightarrow C' \in no\text{-labels.Red-F } \{C, C''\} \Longrightarrow$
 $(N, Some C, P, None, A) \rightsquigarrow OLf (N \cup \{C''\}, Some C, remove C' P, None, A)$
 $| delete\text{-bwd-a}: C' \notin A \Longrightarrow C' \in no\text{-labels.Red-F } \{C\} \vee C \prec \cdot C' \Longrightarrow$
 $(N, Some C, P, None, A \cup \{C'\}) \rightsquigarrow OLf (N, Some C, P, None, A)$
 $| simplify\text{-bwd-a}: C'' \prec_S C' \Longrightarrow C' \notin A \Longrightarrow C' \in no\text{-labels.Red-F } \{C, C''\} \Longrightarrow$
 $(N, Some C, P, None, A \cup \{C''\}) \rightsquigarrow OLf (N \cup \{C''\}, Some C, P, None, A)$

| *transfer*: $(N, \text{Some } C, P, \text{None}, A) \rightsquigarrow \text{OLf } (N, \text{None}, \text{add } C \ P, \text{None}, A)$
| *choose-p*: $P \neq \text{empty} \implies$
 $(\{\|\}, \text{None}, P, \text{None}, A) \rightsquigarrow \text{OLf } (\{\|\}, \text{None}, \text{remove } (\text{select } P) \ P, \text{Some } (\text{select } P), A)$
| *infer*: $\text{no-labels.Inf-between } (\text{fset } A) \ \{C\} \subseteq \text{no-labels.Red-I } (\text{fset } A \cup \{C\} \cup \text{fset } M) \implies$
 $(\{\|\}, \text{None}, P, \text{Some } C, A) \rightsquigarrow \text{OLf } (M, \text{None}, P, \text{None}, A \ |\cup| \ \{C\})$

7.3 Initial State and Invariant

inductive *is-initial-OLf-state* :: $('p, 'f) \text{OLf-state} \Rightarrow \text{bool}$ **where**
is-initial-OLf-state $(N, \text{None}, \text{empty}, \text{None}, \{\|\})$

inductive *OLf-invariant* :: $('p, 'f) \text{OLf-state} \Rightarrow \text{bool}$ **where**
 $(N = \{\|\} \wedge X = \text{None}) \vee Y = \text{None} \implies \text{OLf-invariant } (N, X, P, Y, A)$

lemma *initial-OLf-invariant*: $\text{is-initial-OLf-state } St \implies \text{OLf-invariant } St$
unfolding *is-initial-OLf-state.simps* *OLf-invariant.simps* **by** *auto*

lemma *step-OLf-invariant*:
assumes *step*: $St \rightsquigarrow \text{OLf } St'$
shows *OLf-invariant* St'
using *step* **by** *cases* (*auto intro: OLf-invariant.intros*)

lemma *chain-OLf-invariant-lnth*:
assumes
chain: $\text{chain } (\rightsquigarrow \text{OLf}) \ Sts$ **and**
fair-hd: *OLf-invariant* $(\text{lhd } Sts)$ **and**
i-lt: $\text{enat } i < \text{llength } Sts$
shows *OLf-invariant* $(\text{lnth } Sts \ i)$
using *i-lt*
proof (*induct i*)
case 0
thus *?case*
using *fair-hd lhd-conv-lnth zero-enat-def* **by** *fastforce*
next
case $(\text{Suc } i)$
thus *?case*
using *chain chain-lnth-rel step-OLf-invariant* **by** *blast*
qed

lemma *chain-OLf-invariant-llast*:
assumes
chain: $\text{chain } (\rightsquigarrow \text{OLf}) \ Sts$ **and**
fair-hd: *OLf-invariant* $(\text{lhd } Sts)$ **and**
fin: *lfinite* Sts
shows *OLf-invariant* $(\text{llast } Sts)$
proof –
obtain $i :: \text{nat}$ **where**
 $i: \text{llength } Sts = \text{enat } i$
using *lfinite-llength-enat[OF fin]* **by** *blast*

have *im1-lt*: $\text{enat } (i - 1) < \text{llength } Sts$
using i **by** (*metis chain chain-length-pos diff-less enat-ord-simps(2) less-numeral-extra(1) zero-enat-def*)

show *?thesis*
using *chain-OLf-invariant-lnth[OF chain fair-hd im1-lt]*

by (*metis Suc-diff-1 chain chain-length-pos eSuc-enat enat-ord-simps(2) i llast-conv-lnth zero-enat-def*)

qed

7.4 Final State

inductive *is-final-OLf-state* :: ('p, 'f) *OLf-state* \Rightarrow *bool* **where**
is-final-OLf-state ({}|}, None, empty, None, A)

lemma *is-final-OLf-state-iff-no-OLf-step*:

assumes *inv*: *OLf-invariant St*

shows *is-final-OLf-state St* \longleftrightarrow ($\forall St'. \neg St \rightsquigarrow OLf St'$)

proof

assume *is-final-OLf-state St*

then obtain *A* :: 'f *fset* **where**

st: *St* = ({}|}, None, empty, None, A)

by (*auto simp: is-final-OLf-state.simps*)

show $\forall St'. \neg St \rightsquigarrow OLf St'$

unfolding *st*

proof (*intro allI notI*)

fix *St'*

assume ({}|}, None, empty, None, A) $\rightsquigarrow OLf St'$

thus *False*

by *cases auto*

qed

next

assume *no-step*: $\forall St'. \neg St \rightsquigarrow OLf St'$

show *is-final-OLf-state St*

proof (*rule ccontr*)

assume *not-fin*: $\neg is-final-OLf-state St$

obtain *N A* :: 'f *fset* **and** *X Y* :: 'f *option* **and** *P* :: 'p **where**

st: *St* = (*N*, *X*, *P*, *Y*, *A*)

by (*cases St*)

have *inv'*: (*N* = {}|} \wedge *X* = None) \vee *Y* = None

using *inv st OLf-invariant.simps* **by** *simp*

have *N* \neq {}|} \vee *X* \neq None \vee *P* \neq empty \vee *Y* \neq None

using *not-fin unfolding st is-final-OLf-state.simps* **by** *auto*

moreover {

assume

n: *N* \neq {}|} **and**

x: *X* = None

obtain *N'* :: 'f *fset* **and** *C* :: 'f **where**

n': *N* = *N'* \cup {|*C*|} **and**

c-ni: *C* $\not\subseteq$ *N'*

using *n finsert-is-union* **by** *blast*

have *y*: *Y* = None

using *n x inv'* **by** *meson*

have $\exists St'. St \rightsquigarrow OLf St'$

using *fair-OL.choose-n[OF c-ni]* **unfolding** *st n' x y* **by** *fast*

} **moreover** {

assume *X* \neq None


```

then obtain  $C :: 'f$  where
   $x: X = \text{Some } C$ 
  by blast

have  $y: Y = \text{None}$ 
  using  $x \text{ inv}'$  by auto

have  $\exists St'. St \rightsquigarrow \text{OLf } St'$ 
  using fair-OL.transfer unfolding st x y by fast
} moreover {
  assume
     $p: P \neq \text{empty}$  and
     $n: N = \{\|\}$  and
     $x: X = \text{None}$  and
     $y: Y = \text{None}$ 

  have  $\exists St'. St \rightsquigarrow \text{OLf } St'$ 
    using fair-OL.choose-p[OF p] unfolding st n x y by fast
} moreover {
  assume  $Y \neq \text{None}$ 
  then obtain  $C :: 'f$  where
     $y: Y = \text{Some } C$ 
    by blast

  have  $n: N = \{\|\}$  and
     $x: X = \text{None}$ 
    using  $y \text{ inv}'$  by blast+

  let  $?M = \text{concl-of 'no-labels.Inf-between (fset A) \{C\}}$ 

  have  $\text{fin}: \text{finite } ?M$ 
    by (simp add: finite-Inf-between)
  have  $\text{fset-abs-m}: \text{fset (Abs-fset ?M)} = ?M$ 
    by (rule Abs-fset-inverse[simplified, OF fin])

  have  $\text{inf-red}: \text{no-labels.Inf-between (fset A) \{C\}}$ 
     $\subseteq \text{no-labels.Red-I-G (fset A} \cup \{C\} \cup \text{fset (Abs-fset ?M))}$ 
    by (simp add: fset-abs-m no-labels.Inf-if-Inf-between no-labels.empty-ord.Red-I-of-Inf-to-N subsetI)

  have  $\exists St'. St \rightsquigarrow \text{OLf } St'$ 
    using fair-OL.infer[OF inf-red] unfolding st n x y by fast
} ultimately show False
  using no-step by force
qed
qed

```

7.5 Refinement

lemma *fair-OL-step-imp-OL-step:*

```

assumes olf:  $(N, X, P, Y, A) \rightsquigarrow \text{OLf } (N', X', P', Y', A')$ 
shows fstate  $(N, X, P, Y, A) \rightsquigarrow \text{OL fstate } (N', X', P', Y', A')$ 
using olf
proof cases
  case (choose-n C)
  note  $\text{defs} = \text{this}(1-\gamma)$  and  $\text{c-ni} = \text{this}(8)$ 

```

```

show ?thesis
  unfolding defs fstate.simps option.set using OL.choose-n c-ni by simp
next
  case (delete-fwd C)
  note defs = this(1- $\gamma$ ) and c-red = this(8)
  show ?thesis
    unfolding defs fstate.simps option.set by (rule OL.delete-fwd[OF c-red])
next
  case (simplify-fwd C' C)
  note defs = this(1- $\gamma$ ) and c-red = this(9)
  show ?thesis
    unfolding defs fstate.simps option.set by (rule OL.simplify-fwd[OF c-red])
next
  case (delete-bwd-p C' C)
  note defs = this(1- $\gamma$ ) and c'-in-p = this(8) and c'-red = this(9)

  have p-rm-c'-uni-c': elems (remove C' P)  $\cup$  {C'} = elems P
    unfolding felems-remove by (auto intro: c'-in-p)
  have p-mns-c': elems P - {C'} = elems (remove C' P)
    unfolding felems-remove by auto

  show ?thesis
    unfolding defs fstate.simps option.set
    by (rule OL.delete-bwd-p[OF c'-red, of - elems P - {C'}],
        unfolded p-rm-c'-uni-c' p-mns-c')
next
  case (simplify-bwd-p C'' C' C)
  note defs = this(1- $\gamma$ ) and c'-in-p = this(9) and c'-red = this(10)

  have p-rm-c'-uni-c': elems (remove C' P)  $\cup$  {C'} = elems P
    unfolding felems-remove by (auto intro: c'-in-p)
  have p-mns-c': elems P - {C'} = elems (remove C' P)
    unfolding felems-remove by auto

  show ?thesis
    unfolding defs fstate.simps option.set
    using OL.simplify-bwd-p[OF c'-red, of fset N elems P - {C'}],
    unfolded p-rm-c'-uni-c' p-mns-c'
    by simp
next
  case (delete-bwd-a C' C)
  note defs = this(1- $\gamma$ ) and c'-red = this(9)
  show ?thesis
    unfolding defs fstate.simps option.set using OL.delete-bwd-a[OF c'-red] by simp
next
  case (simplify-bwd-a C' C'' C)
  note defs = this(1- $\gamma$ ) and c'-red = this(10)
  show ?thesis
    unfolding defs fstate.simps option.set using OL.simplify-bwd-a[OF c'-red] by simp
next
  case (transfer C)
  note defs = this(1- $\gamma$ )

  have p-uni-c: elems P  $\cup$  {C} = elems (add C P)
    using felems-add by auto

```

```

show ?thesis
  unfolding defs fstate.simps option.set
  by (rule OL.transfer[of - C elems P, unfolded p-uni-c])
next
case choose-p
note defs = this(1-8) and p-nemp = this(9)

have sel-ni-rm: select P  $\notin$  elems (remove (select P) P)
  unfolding felems-remove by auto

have rm-sel-uni-sel: elems (remove (select P) P)  $\cup$  {select P} = elems P
  unfolding felems-remove using p-nemp select-in-felems
  by (metis Un-insert-right finsert.rep-eq finsert-fminus sup-bot-right)

show ?thesis
  unfolding defs fstate.simps option.set
  using OL.choose-p[of select P elems (remove (select P) P), OF sel-ni-rm,
    unfolded rm-sel-uni-sel]
  by simp
next
case (infer C)
note defs = this(1-7) and infers = this(8)
show ?thesis
  unfolding defs fstate.simps option.set using OL.infer[OF infers] by simp
qed

lemma fair-OL-step-imp-GC-step:
  (N, X, P, Y, A)  $\rightsquigarrow_{OLf}$  (N', X', P', Y', A')  $\implies$ 
  fstate (N, X, P, Y, A)  $\rightsquigarrow_{GC}$  fstate (N', X', P', Y', A')
  by (rule OL-step-imp-GC-step[OF fair-OL-step-imp-OL-step])

end
end

```

8 iProver Loop

The iProver loop is a variant of the Otter loop that supports the elimination of clauses that are made redundant by their own children.

```

theory iProver-Loop
  imports Otter-Loop
begin

```

```

context otter-loop
begin

```

8.1 Definition

```

inductive IL :: ('f  $\times$  OL-label) set  $\Rightarrow$  ('f  $\times$  OL-label) set  $\Rightarrow$  bool (infix  $\rightsquigarrow_{IL}$  50)
where
  ol: St  $\rightsquigarrow_{OL}$  St'  $\implies$  St  $\rightsquigarrow_{IL}$  St'
| red-by-children: C  $\in$  no-labels.Red-F (A  $\cup$  M)  $\vee$  (M = {C'}  $\wedge$  C'  $\prec$  C)  $\implies$ 
  state ({}, {}, P, {C}, A)  $\rightsquigarrow_{IL}$  state (M, {}, P, {}, A)

```

8.2 Refinement

lemma *red-by-children-in-GC*:

assumes $C \in \text{no-labels.Red-F } (A \cup M) \vee (M = \{C'\} \wedge C' \prec C)$
shows $\text{state } (\{\}, \{\}, P, \{C\}, A) \rightsquigarrow_{GC} \text{state } (M, \{\}, P, \{\}, A)$

proof –

let $?N = \text{state } (\{\}, \{\}, P, \{\}, A)$
and $?St = \{(C, YY)\}$
and $?St' = \{(x, \text{New}) \mid x. x \in M\}$

have $(C, YY) \in \text{Red-F } (?N \cup ?St')$
using *assms*

proof

assume *c-in*: $C \in \text{no-labels.Red-F } (A \cup M)$

have $A \cup M \subseteq A \cup M \cup P$ **by** *auto*

also have $\text{fst } '(?N \cup ?St') = A \cup M \cup P$
by *auto*

then have $C \in \text{no-labels.Red-F } (\text{fst } '(?N \cup ?St'))$

by (*metis* (*no-types*, *lifting*) *c-in calculation in-mono no-labels.Red-F-of-subset*)

then show $(C, YY) \in \text{Red-F } (?N \cup ?St')$

using *no-labels-Red-F-imp-Red-F* **by** *blast*

next

assume *assm*: $M = \{C'\} \wedge C' \prec C$

then have $C' \in \text{fst } '(?N \cup ?St')$

by *simp*

then show $(C, YY) \in \text{Red-F } (?N \cup ?St')$

by (*metis* (*mono-tags*) *assm succ-F-imp-Red-F*)

qed

then have *St-included-in*: $?St \subseteq \text{Red-F } (?N \cup ?St')$

by *auto*

have *prj-of-active-subset-of-St'*: $\text{fst } '(\text{active-subset } ?St') = \{\}$
by (*simp add: active-subset-def*)

have $?N \cup ?St \rightsquigarrow_{GC} ?N \cup ?St'$

using *process*[*of - ?N ?St - ?St'*] *St-included-in prj-of-active-subset-of-St'* **by** *auto*

moreover have $?N \cup ?St = \text{state } (\{\}, \{\}, P, \{C\}, A)$

by *simp*

moreover have $?N \cup ?St' = \text{state } (M, \{\}, P, \{\}, A)$

by *auto*

ultimately show $\text{state } (\{\}, \{\}, P, \{C\}, A) \rightsquigarrow_{GC} \text{state } (M, \{\}, P, \{\}, A)$

by *simp*

qed

theorem *IL-step-imp-GC-step*: $M \rightsquigarrow_{IL} M' \implies M \rightsquigarrow_{GC} M'$

proof (*induction rule: IL.induct*)

case (*ol St St'*)

then show *?case*

by (*simp add: OL-step-imp-GC-step*)

next

case (*red-by-children C A M C' P*)

then show *?case* **using** *red-by-children-in-GC*

by *auto*

qed

8.3 Completeness

theorem

assumes

il-chain: $\text{chain } (\rightsquigarrow IL) \text{ } Sts$ **and**

act: $\text{active-subset } (\text{lhs } Sts) = \{\}$ **and**

pas: $\text{passive-subset } (\text{Liminf-list } Sts) = \{\}$

shows

IL-Liminf-saturated: $\text{saturated } (\text{Liminf-list } Sts)$ **and**

IL-complete-Liminf: $B \in \text{Bot-F} \implies \text{fst } \text{' lhs } Sts \models_{\cap \mathcal{G}} \{B\} \implies$

$\exists BL \in \text{Bot-FL}. BL \in \text{Liminf-list } Sts$ **and**

IL-complete: $B \in \text{Bot-F} \implies \text{fst } \text{' lhs } Sts \models_{\cap \mathcal{G}} \{B\} \implies$

$\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists BL \in \text{Bot-FL}. BL \in \text{lnth } Sts \ i)$

proof –

have *gc-chain*: $\text{chain } (\rightsquigarrow GC) \text{ } Sts$

using *il-chain IL-step-imp-GC-step chain-mono* **by** *blast*

show *saturated* ($\text{Liminf-list } Sts$)

using *gc.fair-implies-Liminf-saturated gc-chain gc-fair gc-to-red act pas* **by** *blast*

{

assume

bot: $B \in \text{Bot-F}$ **and**

unsat: $\text{fst } \text{' lhs } Sts \models_{\cap \mathcal{G}} \{B\}$

show $\exists BL \in \text{Bot-FL}. BL \in \text{Liminf-list } Sts$

by (*rule gc-complete-Liminf[OF gc-chain act pas bot unsat]*)

then show $\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists BL \in \text{Bot-FL}. BL \in \text{lnth } Sts \ i)$

unfolding *Liminf-list-def* **by** *auto*

}

qed

end

end

9 Fair iProver Loop

The fair iProver loop assumes that the passive queue is fair and ensures (dynamic) refutational completeness under that assumption. From this completeness proof, we also easily derive (in a separate section) the completeness of the Otter loop.

theory *Fair-iProver-Loop*

imports

Given-Clause-Loops-Util

Fair-Otter-Loop-Def

iProver-Loop

begin

9.1 Locale

context *fair-otter-loop*

begin

9.2 Basic Definition

inductive *fair-IL* :: ('p, 'f) *OLf-state* \Rightarrow ('p, 'f) *OLf-state* \Rightarrow *bool* (**infix** \rightsquigarrow *ILf* 50) **where**
ol: $St \rightsquigarrow$ *OLf* $St' \Longrightarrow St \rightsquigarrow$ *ILf* St'
| *red-by-children*: $C \in \text{no-labels.Red-F } (fset A \cup fset M) \vee fset M = \{C'\} \wedge C' \prec C \Longrightarrow$
 $(\{\|\}, None, P, Some C, A) \rightsquigarrow$ *ILf* $(M, None, P, None, A)$

9.3 Initial State and Invariant

lemma *step-ILf-invariant*:
assumes $St \rightsquigarrow$ *ILf* St'
shows *OLf-invariant* St'
using *assms*
proof *cases*
case *ol*
then show *?thesis*
using *step-OLf-invariant* **by** *auto*
next
case $(\text{red-by-children } C A M C' P)$
then show *?thesis*
using *OLf-invariant.intros* **by** *presburger*
qed

lemma *chain-ILf-invariant-lnth*:
assumes
chain: $chain (\rightsquigarrow$ *ILf*) Sts **and**
fair-hd: *OLf-invariant* $(lhd Sts)$ **and**
i-lt: $enat i < llength Sts$
shows *OLf-invariant* $(lnth Sts i)$
using *i-lt*
proof $(induct i)$
case 0
thus *?case*
using *fair-hd lhd-conv-lnth zero-enat-def* **by** *fastforce*
next
case $(Suc i)$
thus *?case*
using *chain chain-lnth-rel step-ILf-invariant* **by** *blast*
qed

lemma *chain-ILf-invariant-llast*:
assumes
chain: $chain (\rightsquigarrow$ *ILf*) Sts **and**
fair-hd: *OLf-invariant* $(lhd Sts)$ **and**
fin: *lfinite* Sts
shows *OLf-invariant* $(llast Sts)$
proof –
obtain $i :: nat$ **where**
i: $llength Sts = enat i$
using *lfinite-llength-enat[OF fin]* **by** *blast*

have *im1-lt*: $enat (i - 1) < llength Sts$
using i **by** $(metis chain chain-length-pos diff-less enat-ord-simps(2) less-numeral-extra(1) zero-enat-def)$

show *?thesis*

using *chain-ILf-invariant-lnth*[*OF chain fair-hd im1-lt*]
by (*metis Suc-diff-1 chain chain-length-pos eSuc-enat enat-ord-simps*(2) *i llast-conv-lnth zero-enat-def*)
qed

9.4 Final State

lemma *is-final-OLf-state-iff-no-ILf-step*:
assumes *inv*: *OLf-invariant St*
shows *is-final-OLf-state St* \longleftrightarrow $(\forall St'. \neg St \rightsquigarrow_{ILf} St')$
proof
assume *final*: *is-final-OLf-state St*
then obtain *A* :: '*f fset* **where**
st: *St = ({|}, None, empty, None, A)*
by (*auto simp: is-final-OLf-state.simps*)
show $\forall St'. \neg St \rightsquigarrow_{ILf} St'$
unfolding *st*
proof (*intro allI notI*)
fix *St'*
assume $({|}, None, empty, None, A) \rightsquigarrow_{ILf} St'$
thus *False*
proof *cases*
case *ol*
then show *False*
using *final st is-final-OLf-state-iff-no-OLf-step*[*OF inv*] **by** *blast*
qed
qed
next
assume $\forall St'. \neg St \rightsquigarrow_{ILf} St'$
hence $\forall St'. \neg St \rightsquigarrow_{OLf} St'$
using *fair-IL.ol* **by** *blast*
thus *is-final-OLf-state St*
using *inv is-final-OLf-state-iff-no-OLf-step* **by** *blast*
qed

9.5 Refinement

lemma *fair-IL-step-imp-IL-step*:
assumes *ilf*: $(N, X, P, Y, A) \rightsquigarrow_{ILf} (N', X', P', Y', A')$
shows *fstate* $(N, X, P, Y, A) \rightsquigarrow_{IL} fstate (N', X', P', Y', A')$
using *ilf*
proof *cases*
case *ol*
note *olf = this(1)*
have *ol*: $fstate (N, X, P, Y, A) \rightsquigarrow_{OL} fstate (N', X', P', Y', A')$
by (*rule fair-OL-step-imp-OL-step*[*OF olf*])
show *?thesis*
by (*rule IL.ol*[*OF ol*])
next
case (*red-by-children C C'*)
note *defs = this(1-7)* **and** *c-in = this(8)*
have *il*: $state (\{\}, \{\}, elems P, \{C\}, fset A) \rightsquigarrow_{IL} state (fset N', \{\}, elems P, \{\}, fset A)$
by (*rule IL.red-by-children*[*OF c-in*])
show *?thesis*
unfolding *defs* **using** *il* **by** *auto*
qed

lemma *fair-IL-step-imp-GC-step*:

$(N, X, P, Y, A) \rightsquigarrow_{ILf} (N', X', P', Y', A') \implies$
 $fstate (N, X, P, Y, A) \rightsquigarrow_{GC} fstate (N', X', P', Y', A')$
by (*rule IL-step-imp-GC-step[OF fair-IL-step-imp-IL-step]*)

9.6 Completeness

fun *mset-of-fstate* :: ('p, 'f) *OLf-state* \Rightarrow 'f *multiset* **where**

mset-of-fstate (N, X, P, Y, A) =
mset-set (fset N) + *mset-set* (set-option X) + *mset-set* (elems P) + *mset-set* (set-option Y) +
mset-set (fset A)

abbreviation *Precprec-S* :: 'f *multiset* \Rightarrow 'f *multiset* \Rightarrow bool (**infix** $\prec\prec_S$ 50) **where**

$\prec\prec_S \equiv multp \prec_S$

lemma *wfP-Precprec-S*: *wfP* ($\prec\prec_S$)

using *minimal-element-def wfP-multp wf-Prec-S wfp-on-UNIV* **by** *blast*

definition *Less1-state* :: ('p, 'f) *OLf-state* \Rightarrow ('p, 'f) *OLf-state* \Rightarrow bool (**infix** \sqsubset_1 50) **where**

St' \sqsubset_1 St \longleftrightarrow
mset-of-fstate *St'* $\prec\prec_S$ *mset-of-fstate* *St*
 \vee (*mset-of-fstate* *St'* = *mset-of-fstate* *St*)
 \wedge (*mset-set* (fset (new-of *St'*)) $\prec\prec_S$ *mset-set* (fset (new-of *St*)))
 \vee (*mset-set* (fset (new-of *St'*)) = *mset-set* (fset (new-of *St*)))
 \wedge *mset-set* (set-option (xx-of *St'*)) $\prec\prec_S$ *mset-set* (set-option (xx-of *St*))))

lemma *wfP-Less1-state*: *wfP* (\sqsubset_1)

proof –

let ?*msetset* = {(M', M). M' $\prec\prec_S$ M}

let ?*triple-of* =

$\lambda St. (mset-of-fstate St, mset-set (fset (new-of St)), mset-set (set-option (xx-of St)))$

have *wf-msetset*: *wf* ?*msetset*

using *wfP-Precprec-S wfP-def* **by** *auto*

have *wf-lex-prod*: *wf* (?*msetset* $\langle *lex* \rangle$?*msetset* $\langle *lex* \rangle$?*msetset*)

by (*rule wf-lex-prod[OF wf-msetset wf-lex-prod[OF wf-msetset wf-msetset]]*)

have *Less1-state-alt-def*: $\bigwedge St' St. St' \sqsubset_1 St \longleftrightarrow$

(?*triple-of* *St'*, ?*triple-of* *St*) \in ?*msetset* $\langle *lex* \rangle$?*msetset* $\langle *lex* \rangle$?*msetset*

unfolding *Less1-state-def* **by** *simp*

show ?*thesis*

unfolding *wfP-def Less1-state-alt-def* **using** *wf-app[of - ?triple-of] wf-lex-prod* **by** *blast*

qed

definition *Less2-state* :: ('p, 'f) *OLf-state* \Rightarrow ('p, 'f) *OLf-state* \Rightarrow bool (**infix** \sqsubset_2 50) **where**

St' \sqsubset_2 St \equiv
mset-set (set-option (yy-of *St'*)) $\prec\prec_S$ *mset-set* (set-option (yy-of *St*))
 \vee (*mset-set* (set-option (yy-of *St'*)) = *mset-set* (set-option (yy-of *St*)))
 \wedge *St' \sqsubset_1 St*)

lemma *wfP-Less2-state*: *wfP* (\sqsubset_2)

proof –

let ?*msetset* = {(M', M). M' $\prec\prec_S$ M}

let ?*stateset* = {(St', St). St' \sqsubset_1 St}


```

let ?pair-of = λSt. (mset-set (set-option (yy-of St)), St)

have wf-msetset: wf ?msetset
  using wfP-Precprec-S wfP-def by auto
have wf-stateset: wf ?stateset
  using wfP-Less1-state wfP-def by auto
have wf-lex-prod: wf (?msetset <*lex*> ?stateset)
  by (rule wf-lex-prod[OF wf-msetset wf-stateset])

have Less2-state-alt-def:
  ∧St' St. St' ⊆ St ↔ (?pair-of St', ?pair-of St) ∈ ?msetset <*lex*> ?stateset
  unfolding Less2-state-def by simp

show ?thesis
  unfolding wfP-def Less2-state-alt-def using wf-app[of - ?pair-of] wf-lex-prod by blast
qed

lemma fair-IL-Liminf-yy-empty:
  assumes
    full: full-chain (↪ILf) Sts and
    inv: OLf-invariant (lhd Sts)
  shows Liminf-llist (lmap (set-option ∘ yy-of) Sts) = {}
proof (rule ccontr)
  assume lim-nemp: Liminf-llist (lmap (set-option ∘ yy-of) Sts) ≠ {}

  have chain: chain (↪ILf) Sts
    by (rule full-chain-imp-chain[OF full])

  obtain i :: nat where
    i-lt: enat i < llength Sts and
    inter-nemp: ∩ ((set-option ∘ yy-of ∘ lnth Sts) ' {j. i ≤ j ∧ enat j < llength Sts}) ≠ {}
    using lim-nemp unfolding Liminf-llist-def by auto

  have inv-at-i: OLf-invariant (lnth Sts i)
    by (simp add: chain-ILf-invariant-lnth i-lt inv)

  from inter-nemp obtain C :: 'f where
    c-in: ∀ P ∈ lnth Sts ' {j. i ≤ j ∧ enat j < llength Sts}. C ∈ set-option (yy-of P)
    by auto
  hence c-in': ∀ j ≥ i. enat j < llength Sts → C ∈ set-option (yy-of (lnth Sts j))
    by auto

  have yy-at-i: yy-of (lnth Sts i) = Some C
    using c-in' i-lt by blast
  have new-at-i: new-of (lnth Sts i) = {||} and
    xx-at-i: new-of (lnth Sts i) = {||}
    using yy-at-i chain-ILf-invariant-lnth[OF chain inv i-lt]
    by (force simp: OLf-invariant.simps)+

  have ∃ St'. lnth Sts i ↪ILf St'
    using is-final-OLf-state-iff-no-ILf-step[OF inv-at-i]
    by (metis fst-conv is-final-OLf-state.cases option.simps(3) snd-conv yy-at-i)
  hence si-lt: enat (Suc i) < llength Sts
    by (metis Suc-ile-eq full full-chain-lnth-not-rel i-lt order-le-imp-less-or-eq)

```

```

obtain  $P :: 'p$  and  $A :: 'f$  fset where
   $at-i: lnth\ Sts\ i = (\{\|\}, None, P, Some\ C, A)$ 
  using OLf-invariant.simps inv-at-i yy-at-i by auto

have  $lnth\ Sts\ i \rightsquigarrow ILf\ lnth\ Sts\ (Suc\ i)$ 
  by (simp add: chain chain-lnth-rel si-lt)
hence  $(\{\|\}, None, P, Some\ C, A) \rightsquigarrow ILf\ lnth\ Sts\ (Suc\ i)$ 
  unfolding at-i .
hence  $yy-of\ (lnth\ Sts\ (Suc\ i)) = None$ 
proof cases
  case ol
  then show ?thesis
    by cases simp
next
  case (red-by-children M C')
  then show ?thesis
    by simp
qed
thus False
  using c-in' si-lt by simp
qed

lemma xx-nonempty-OLf-step-imp-Precprec-S:
  assumes
     $step: St \rightsquigarrow OLf\ St'$  and
     $xx: xx-of\ St \neq None$  and
     $xx': xx-of\ St' \neq None$ 
  shows  $mset-of-fstate\ St' \prec\prec_S\ mset-of-fstate\ St$ 
  using step
proof cases
  case (simplify-fwd C' C P A N)
  note  $defs = this(1,2)$  and  $prec = this(3)$ 

  have  $aft: add-mset\ C'\ (mset-set\ (fset\ N) + mset-set\ (elems\ P) + mset-set\ (fset\ A)) =$ 
     $mset-set\ (fset\ N) + mset-set\ (elems\ P) + mset-set\ (fset\ A) + \{\#C'\#\}$ 
    (is  $?old-aft = ?new-aft$ )
    by auto
  have  $bef: add-mset\ C\ (mset-set\ (fset\ N) + mset-set\ (elems\ P) + mset-set\ (fset\ A)) =$ 
     $mset-set\ (fset\ N) + mset-set\ (elems\ P) + mset-set\ (fset\ A) + \{\#C\#\}$ 
    (is  $?old-bef = ?new-bef$ )
    by auto

  have  $?new-aft \prec\prec_S\ ?new-bef$ 
    unfolding multp-def
  proof (subst mult-cancelL[OF trans-Prec-S irreft-Prec-S], fold multp-def)
    show  $\{\#C'\#\} \prec\prec_S\ \{\#C\#\}$ 
      by (simp add: multp-def prec singletons-in-mult)
  qed
  hence  $?old-aft \prec\prec_S\ ?old-bef$ 
    unfolding bef aft .
  thus ?thesis
    unfolding defs by auto
next
  case (delete-bwd-p C' P C N A)
  note  $defs = this(1,2)$  and  $c'-in = this(3)$ 

```

```

have mset-set (elems P - {C'})  $\subset\#$  mset-set (elems P)
  by (metis Diff-iff c'-in finite-fset finite-set-mset-mset-set elems-remove insertCI
    insert-Diff subset-imp-msubset-mset-set subset-insertI subset-mset.less-le)
thus ?thesis
  unfolding defs using c'-in
  by (auto simp: elems-remove intro!: subset-implies-multip)
next
case (simplify-bwd-p C'' C' P C N A)
note defs = this(1,2) and prec = this(3) and c'-in = this(4)

let ?old-aft = add-mset C (mset-set (insert C'' (fset N)) + mset-set (elems (remove C' P)) +
  mset-set (fset A))
let ?old-bef = add-mset C (mset-set (fset N) + mset-set (elems P) + mset-set (fset A))

have ?old-aft  $\prec\prec_S$  ?old-bef
proof (cases C''  $\in$  fset N)
  case c''-in: True

    have mset-set (elems P - {C'})  $\subset\#$  mset-set (elems P)
      by (metis c'-in finite-fset mset-set.remove multi-psub-of-add-self)
    thus ?thesis
      unfolding defs
      by (auto simp: elems-remove insert-absorb[OF c''-in] intro!: subset-implies-multip)
    next
    case c''-ni: False

      have aft: ?old-aft = add-mset C (mset-set (fset N) + mset-set (elems (remove C' P)) +
        mset-set (fset A)) + {#C''#}
        (is - = ?new-aft)
        using c''-ni by auto
      have bef: ?old-bef = add-mset C (mset-set (fset N) + mset-set (elems (remove C' P)) +
        mset-set (fset A)) + {#C' #}
        (is - = ?new-bef)
        using c'-in by (auto simp: elems-remove mset-set.remove)

      have ?new-aft  $\prec\prec_S$  ?new-bef
        unfolding multp-def
      proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
        show {#C''#}  $\prec\prec_S$  {#C' #}
          unfolding multp-def using prec by (auto intro: singletons-in-mult)
      qed
      thus ?thesis
        unfolding bef aft .
    qed
    thus ?thesis
      unfolding defs by auto
  next
  case (delete-bwd-a C' A C N P)
  note defs = this(1,2) and c'-ni = this(3)
  show ?thesis
    unfolding defs using c'-ni by (auto intro!: subset-implies-multip)
  next
  case (simplify-bwd-a C'' C' A C N P)
  note defs = this(1,2) and prec = this(3) and c'-ni = this(4)

```

```

have aft:
  add-mset C (mset-set (insert C'' (fset N)) + mset-set (elems P) + mset-set (fset A)) =
    {#C#} + mset-set (elems P) + mset-set (fset A) + mset-set (insert C'' (fset N))
  (is ?old-aft = ?new-aft)
  by auto
have bef:
  add-mset C' (add-mset C (mset-set (fset N) + mset-set (elems P) + mset-set (fset A))) =
    {#C#} + mset-set (elems P) + mset-set (fset A) + ({#C'#} + mset-set (fset N))
  (is ?old-bef = ?new-bef)
  by auto

have ?new-aft <-<S ?new-bef
  unfolding multp-def
proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
show mset-set (insert C'' (fset N)) <-<S {#C'#} + mset-set (fset N)
proof (cases C'' ∈ fset N)
  case True
  hence ins: insert C'' (fset N) = fset N
  by blast
  show ?thesis
  unfolding ins by (auto intro!: subset-implies-multp)
next
  case c''-ni: False

  have aft: mset-set (insert C'' (fset N)) = mset-set (fset N) + {#C''#}
  using c''-ni by auto
  have bef: {#C'#} + mset-set (fset N) = mset-set (fset N) + {#C'#}
  by auto

  show ?thesis
  unfolding aft bef multp-def
  proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
  show {#C''#} <-<S {#C'#}
  unfolding multp-def using prec by (auto intro: singletons-in-mult)
  qed
qed
qed
hence ?old-aft <-<S ?old-bef
  unfolding bef aft .
thus ?thesis
  unfolding defs using c'-ni by auto
qed (use xx xx' in auto)

lemma xx-nonempty-ILf-step-imp-Precprec-S:
assumes
  step: St ~ ILf St' and
  xx: xx-of St ≠ None and
  xx': xx-of St' ≠ None
shows mset-of-fstate St' <-<S mset-of-fstate St
using step
proof cases
  case ol
  then show ?thesis
  using xx-nonempty-OLf-step-imp-Precprec-S[OF - xx xx'] by blast
next

```

case (*red-by-children* $C A M C' P$)
note $defs = this(1,2)$
have *False*
 using *xx unfolding defs by simp*
thus *?thesis*
 by *blast*
qed

lemma *fair-IL-Liminf-xx-empty*:

assumes

len: $llength\ Sts = \infty$ **and**

full: *full-chain* ($\rightsquigarrow ILf$) *Sts* **and**

inv: *OLf-invariant* (*lhd Sts*)

shows $Liminf\llist (lmap (set\ option \circ xx\ of) Sts) = \{\}$

proof (*rule ccontr*)

assume *lim-nemp*: $Liminf\llist (lmap (set\ option \circ xx\ of) Sts) \neq \{\}$

obtain $i :: nat$ **where**

i-lt: $enat\ i < llength\ Sts$ **and**

inter-nemp: $\bigcap ((set\ option \circ xx\ of \circ lnth\ Sts) \text{ ' } \{j. i \leq j \wedge enat\ j < llength\ Sts\}) \neq \{\}$

using *lim-nemp unfolding Liminf-llist-def by auto*

from *inter-nemp* **obtain** $C :: 'f$ **where**

c-in: $\forall P \in lnth\ Sts \text{ ' } \{j. i \leq j \wedge enat\ j < llength\ Sts\}. C \in set\ option (xx\ of\ P)$

by *auto*

hence *c-in'*: $\forall j \geq i. enat\ j < llength\ Sts \longrightarrow C \in set\ option (xx\ of (lnth\ Sts\ j))$

by *auto*

have *si-lt*: $enat (Suc\ i) < llength\ Sts$

unfolding *len* **by** *auto*

have *xx-j*: $xx\ of (lnth\ Sts\ j) \neq None$ **if** *j-ge*: $j \geq i$ **for** j

using *c-in' len j-ge by auto*

hence *xx-sj*: $xx\ of (lnth\ Sts (Suc\ j)) \neq None$ **if** *j-ge*: $j \geq i$ **for** j

using *le-Suc-eq that by presburger*

have *step*: $lnth\ Sts\ j \rightsquigarrow ILf\ lnth\ Sts (Suc\ j)$ **if** *j-ge*: $j \geq i$ **for** j

using *full-chain-imp-chain[OF full] infinite-chain-lnth-rel len llength-eq-infty-conv-lfinite*

by *blast*

have *mset-of-fstate* ($lnth\ Sts (Suc\ j)$) $\prec\prec S$ *mset-of-fstate* ($lnth\ Sts\ j$) **if** *j-ge*: $j \geq i$ **for** j

using *xx-nonempty-ILf-step-imp-Precprec-S by (meson step j-ge xx-j xx-sj)*

hence $(\prec\prec S)^{-1-1} (mset\ of\ fstate (lnth\ Sts\ j)) (mset\ of\ fstate (lnth\ Sts (Suc\ j)))$

if *j-ge*: $j \geq i$ **for** j

using *j-ge by blast*

hence *inf-down-chain*: $chain (\prec\prec S)^{-1-1} (ldropn\ i (lmap\ mset\ of\ fstate\ Sts))$

using *chain-ldropn-lmapI[OF - si-lt] by blast*

have *inf-i*: $\neg lfinite (ldropn\ i\ Sts)$

using *len by (simp add: llength-eq-infty-conv-lfinite)*

show *False*

using *inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of (\prec\prec S)] wfP-Precprec-S*

by (*metis lfinite-ldropn lfinite-lmap*)

qed

lemma *xx-nonempty-OLf-step-imp-Less1-state*:
assumes *step*: $(N, \text{Some } C, P, Y, A) \rightsquigarrow_{OLf} (N', \text{Some } C', P', Y', A')$ (**is** $?bef \rightsquigarrow_{OLf} ?aft$)
shows $?aft \sqsubset 1 ?bef$
proof –
have *mset-of-fstate* $?aft \prec\prec_S \text{mset-of-fstate } ?bef$
using *xx-nonempty-OLf-step-imp-Precprec-S*
by (*metis fst-conv local.step option.distinct(1) snd-conv*)
thus *?thesis*
unfolding *Less1-state-def* **by** *blast*
qed

lemma *yy-empty-OLf-step-imp-Less1-state*:

assumes
step: $St \rightsquigarrow_{OLf} St'$ **and**
yy: $yy\text{-of } St = \text{None}$ **and**
yy': $yy\text{-of } St' = \text{None}$

shows $St' \sqsubset 1 St$

using *step*

proof *cases*

case (*choose-n C N P A*)

note *defs = this(1,2)* **and** *c-ni = this(3)*

have *mset-eq*: $\text{mset-of-fstate } St' = \text{mset-of-fstate } St$

unfolding *defs* **using** *c-ni* **by** *fastforce*

have *new-lt*: $\text{mset-set } (\text{fset } (\text{new-of } St')) \prec\prec_S \text{mset-set } (\text{fset } (\text{new-of } St))$

unfolding *defs* **using** *c-ni*

by (*auto intro!: subset-implies-multip*)

show *?thesis*

unfolding *Less1-state-def* **using** *mset-eq new-lt* **by** *blast*

next

case (*delete-fwd C P A N*)

note *defs = this(1,2)*

have *mset-of-fstate* $St' \prec\prec_S \text{mset-of-fstate } St$

unfolding *defs* **by** (*auto intro: subset-implies-multip*)

thus *?thesis*

unfolding *Less1-state-def* **by** *blast*

next

case (*simplify-fwd C' C P A N*)

note *defs = this(1,2)*

show *?thesis*

unfolding *defs* **by** (*rule xx-nonempty-OLf-step-imp-Less1-state[OF step[unfolded defs]]*)

next

case (*delete-bwd-p C' P C N A*)

note *defs = this(1,2)*

show *?thesis*

unfolding *defs* **by** (*rule xx-nonempty-OLf-step-imp-Less1-state[OF step[unfolded defs]]*)

next

case (*simplify-bwd-p C'' C' P C N A*)

note *defs = this(1,2)*

show *?thesis*

unfolding *defs* **by** (*rule xx-nonempty-OLf-step-imp-Less1-state[OF step[unfolded defs]]*)

next

case (*delete-bwd-a C' A C N P*)

note *defs = this(1,2)*

```

show ?thesis
  unfolding defs by (rule xx-nonempty-OLf-step-imp-Less1-state[OF step[unfolded defs]])
next
case (simplify-bwd-a C'' C' A C N P)
note defs = this(1,2)
show ?thesis
  unfolding defs by (rule xx-nonempty-OLf-step-imp-Less1-state[OF step[unfolded defs]])
next
case (transfer N C P A)
note defs = this(1,2)
show ?thesis
proof (cases C ∈ elems P)
  case c-in: True
  have mset-of-fstate St' <<S mset-of-fstate St
    unfolding defs using c-in add-again
    by (auto intro!: subset-implies-multp)
  thus ?thesis
    unfolding Less1-state-def by blast
next
  case c-ni: False

  have mset-eq: mset-of-fstate St' = mset-of-fstate St
    unfolding defs using c-ni by (auto simp: elems-add)
  have new-mset-eq: mset-set (fset (new-of St')) = mset-set (fset (new-of St))
    unfolding defs using c-ni by auto
  have xx-lt: mset-set (set-option (xx-of St')) <<S mset-set (set-option (xx-of St))
    unfolding defs using c-ni by (auto intro!: subset-implies-multp)

  show ?thesis
    unfolding Less1-state-def using mset-eq new-mset-eq xx-lt by blast
qed
qed (use yy yy' in auto)

lemma yy-empty-ILf-step-imp-Less1-state:
assumes
  step: St ~ILf St' and
  yy: yy-of St = None and
  yy': yy-of St' = None
shows St' ⊆1 St
using step
proof cases
case ol
then show ?thesis
  using yy-empty-OLf-step-imp-Less1-state[OF - yy yy'] by blast
next
case (red-by-children C A M C' P)
note defs = this(1,2)
have False
  using yy unfolding defs by simp
then show ?thesis
  by blast
qed

```

```

lemma fair-IL-Liminf-new-empty:
assumes

```

len: $\text{llength } Sts = \infty$ **and**
full: $\text{full-chain } (\rightsquigarrow \text{ILf}) Sts$ **and**
inv: $\text{OLf-invariant } (\text{lhd } Sts)$
shows $\text{Liminf-llist } (\text{lmap } (\text{fset} \circ \text{new-of}) Sts) = \{\}$
proof (*rule ccontr*)
assume *lim-nemp*: $\text{Liminf-llist } (\text{lmap } (\text{fset} \circ \text{new-of}) Sts) \neq \{\}$

obtain *i* :: *nat* **where**
i-lt: $\text{enat } i < \text{llength } Sts$ **and**
inter-nemp: $\bigcap ((\text{fset} \circ \text{new-of} \circ \text{lnth } Sts) \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } Sts\}) \neq \{\}$
using *lim-nemp* **unfolding** *Liminf-llist-def* **by** *auto*

from *inter-nemp* **obtain** *C* :: '*f* **where**
c-in: $\forall P \in \text{lnth } Sts \text{ ' } \{j. i \leq j \wedge \text{enat } j < \text{llength } Sts\}. C \in \text{fset } (\text{new-of } P)$
by *auto*
hence *c-in'*: $\forall j \geq i. \text{enat } j < \text{llength } Sts \longrightarrow C \in \text{fset } (\text{new-of } (\text{lnth } Sts j))$
by *auto*

have *si-lt*: $\text{enat } (\text{Suc } i) < \text{llength } Sts$
by (*simp add: len*)

have *new-j*: $\text{new-of } (\text{lnth } Sts j) \neq \{\|\}$ **if** *j-ge*: $j \geq i$ **for** *j*
using *c-in'* *len* **that** **by** *fastforce*

have *yy*: $\text{yy-of } (\text{lnth } Sts j) = \text{None}$ **if** *j-ge*: $j \geq i$ **for** *j*
by (*smt (z3) chain-ILf-invariant-lnth enat-ord-code(4) OLf-invariant.cases fst-conv full full-chain-imp-chain inv len new-j snd-conv j-ge*)
hence *yy'*: $\text{yy-of } (\text{lnth } Sts (\text{Suc } j)) = \text{None}$ **if** *j-ge*: $j \geq i$ **for** *j*
using *j-ge* **by** *auto*

have *step*: $\text{lnth } Sts j \rightsquigarrow \text{ILf } \text{lnth } Sts (\text{Suc } j)$ **if** *j-ge*: $j \geq i$ **for** *j*
using *full-chain-imp-chain[OF full] infinite-chain-lnth-rel len llength-eq-infty-conv-lfinite*
by *blast*

have $\text{lnth } Sts (\text{Suc } j) \sqsubset 1 \text{ lnth } Sts j$ **if** *j-ge*: $j \geq i$ **for** *j*
by (*rule yy-empty-ILf-step-imp-Less1-state[OF step[OF j-ge] yy[OF j-ge] yy'[OF j-ge]]*)
hence $(\sqsubset 1)^{-1-1} (\text{lnth } Sts j) (\text{lnth } Sts (\text{Suc } j))$ **if** *j-ge*: $j \geq i$ **for** *j*
using *j-ge* **by** *blast*
hence *inf-down-chain*: $\text{chain } (\sqsubset 1)^{-1-1} (\text{ldropn } i Sts)$
using *chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id]* **by** *simp*

have *inf-i*: $\neg \text{lfinite } (\text{ldropn } i Sts)$
using *len lfinite-ldropn llength-eq-infty-conv-lfinite* **by** *blast*

show *False*
using *inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of (\sqsubset 1)] wfP-Less1-state*
by *blast*

qed

lemma *yy-empty-OLf-step-imp-Less2-state*:
assumes *step*: $(N, X, P, \text{None}, A) \rightsquigarrow \text{OLf } (N', X', P', \text{None}, A')$ (**is** *?bef* $\rightsquigarrow \text{OLf } ?aft$)
shows *?aft* $\sqsubset 2$ *?bef*
proof –
have *?aft* $\sqsubset 1$ *?bef*
using *yy-empty-OLf-step-imp-Less1-state* **by** (*simp add: step*)
thus *?thesis*

unfolding *Less2-state-def* **by force**
qed

lemma *non-choose-p-OLf-step-imp-Less2-state*:

assumes

step: $St \rightsquigarrow OLf\ St'$ **and**

yy: *yy-of* $St' = None$

shows $St' \sqsubseteq_2 St$

using *step*

proof *cases*

case (*choose-n* $C\ N\ P\ A$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*delete-fwd* $C\ P\ A\ N$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*simplify-fwd* $C'\ C\ P\ A\ N$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*delete-bwd-p* $C'\ P\ C\ N\ A$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*simplify-bwd-p* $C''\ C'\ P\ C\ N\ A$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*delete-bwd-a* $C'\ A\ C\ N\ P$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*simplify-bwd-a* $C''\ C'\ A\ C\ N\ P$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*transfer* $N\ C\ P\ A$)

note *defs* = *this*(1,2)

show *?thesis*

unfolding *defs* **by** (*rule* *yy-empty-OLf-step-imp-Less2-state*[*OF* *step*[*unfolded* *defs*]])

next

case (*choose-p* $P\ A$)

note *defs* = *this*(1,2)

have *False*

using *step* *yy* **unfolding** *defs* **by** *simp*

thus *?thesis*

by *blast*
 next
 case (*infer A C M P*)
 note *defs = this(1,2)*
 have *mset-set (set-option (yy-of St')) <<S mset-set (set-option (yy-of St))*
 unfolding defs by (auto intro!: subset-implies-multip)
 thus *?thesis*
 unfolding Less2-state-def by blast
 qed

lemma *non-choose-p-ILf-step-imp-Less2-state*:

assumes
 step: St ~ILf St' and
 yy: yy-of St' = None
 shows *St' \sqsubset St*
 using *step*
 proof cases
 case *ol*
 then show *?thesis*
 using *non-choose-p-OLf-step-imp-Less2-state[OF - yy] by blast*
 next
 case (*red-by-children C A M C' P*)
 note *defs = this(1,2)*
 show *?thesis*
 unfolding defs Less2-state-def by (simp add: subset-implies-multip)
 qed

lemma *OLf-step-imp-queue-step*:

assumes *St ~OLf St'*
 shows *queue-step (passive-of St) (passive-of St')*
 using *assms by cases (auto intro: queue-step-idleI queue-step-addI queue-step-removeI)*

lemma *ILf-step-imp-queue-step*:

assumes *step: St ~ILf St'*
 shows *queue-step (passive-of St) (passive-of St')*
 using *step*
 proof cases
 case *ol*
 then show *?thesis*
 using *OLf-step-imp-queue-step by blast*
 next
 case (*red-by-children C A M C' P*)
 note *defs = this(1,2)*
 show *?thesis*
 unfolding defs by (auto intro: queue-step-idleI)
 qed

lemma *fair-IL-Liminf-passive-empty*:

assumes
 len: llength Sts = ∞ and
 full: full-chain (\rightsquigarrow ILf) Sts and
 init: is-initial-OLf-state (lhd Sts)
 shows *Liminf-llist (lmap (elems \circ passive-of) Sts) = {}*
 proof –
 have *chain-step: chain queue-step (lmap passive-of Sts)*

```

using ILf-step-imp-queue-step chain-lmap full-chain-imp-chain[OF full]
by (metis (no-types, lifting))

have inf-oft: infinitely-often select-queue-step (lmap passive-of Sts)
proof
  assume finitely-often select-queue-step (lmap passive-of Sts)
  then obtain i :: nat where
    no-sel:
       $\forall j \geq i. \neg \text{select-queue-step (passive-of (lnth Sts j)) (passive-of (lnth Sts (Suc j)))}$ 
    by (metis (no-types, lifting) enat-ord-code(4) finitely-often-def len llength-lmap lnth-lmap)

have si-lt: enat (Suc i) < llength Sts
  unfolding len by auto

have step: lnth Sts j  $\rightsquigarrow$  ILf lnth Sts (Suc j) if j-ge: j  $\geq$  i for j
  using full-chain-imp-chain[OF full] infinite-chain-lnth-rel len llength-eq-infty-conv-lfinite
  by blast

have yy: yy-of (lnth Sts (Suc j)) = None if j-ge: j  $\geq$  i for j
  using step[OF j-ge]
proof cases
  case ol
  then show ?thesis
  proof cases
    case (choose-p P A)
    note defs = this(1,2) and p-ne = this(3)
    have False
      using no-sel defs p-ne select-queue-stepI that by fastforce
    thus ?thesis
      by blast
  qed auto
next
  case (red-by-children C A M C' P)
  then show ?thesis
    by simp
qed

have lnth Sts (Suc j)  $\sqsubset$ 2 lnth Sts j if j-ge: j  $\geq$  i for j
  by (rule non-choose-p-ILf-step-imp-Less2-state[OF step[OF j-ge] yy[OF j-ge]])
hence ( $\sqsubset$ 2)-1-1 (lnth Sts j) (lnth Sts (Suc j)) if j-ge: j  $\geq$  i for j
  using j-ge by blast
hence inf-down-chain: chain ( $\sqsubset$ 2)-1-1 (ldropn i Sts)
  using chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id] by simp

have inf-i:  $\neg$  lfinite (ldropn i Sts)
  using len lfinite-ldropn llength-eq-infty-conv-lfinite by blast

show False
  using inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of ( $\sqsubset$ 2)] wfP-Less2-state
  by blast
qed

have hd-emp: lhd (lmap passive-of Sts) = empty
  using init full full-chain-not-lnull unfolding is-initial-OLf-state.simps by fastforce

```

thm *fair*

have *Liminf-llist* (*lmap elems (lmap passive-of Sts)*) = {}
 by (*rule fair[of lmap passive-of Sts, OF chain-step inf-oft hd-emp]*)
thus *?thesis*
 by (*simp add: llist.map-comp*)
qed

theorem

assumes

full: *full-chain* (\rightsquigarrow ILf) *Sts* **and**
 init: *is-initial-OLf-state* (*lhd Sts*)

shows

fair-IL-Liminf-saturated: *saturated* (*state (Liminf-fstate Sts)*) **and**
 fair-IL-complete-Liminf: $B \in \text{Bot-F} \implies \text{fset} (\text{new-of} (\text{lhd Sts})) \models \cap \mathcal{G} \{B\} \implies$
 $\exists B' \in \text{Bot-F}. B' \in \text{state-union} (\text{Liminf-fstate Sts})$ **and**
 fair-IL-complete: $B \in \text{Bot-F} \implies \text{fset} (\text{new-of} (\text{lhd Sts})) \models \cap \mathcal{G} \{B\} \implies$
 $\exists i. \text{enat } i < \text{llength Sts} \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of} (\text{lnth Sts } i))$

proof –

have *chain*: *chain* (\rightsquigarrow ILf) *Sts*
 by (*rule full-chain-imp-chain[OF full]*)
have *il-chain*: *chain* (\rightsquigarrow IL) (*lmap fstate Sts*)
 by (*rule chain-lmap[OF - chain]*) (*use fair-IL-step-imp-IL-step in force*)

have *inv*: *OLf-invariant* (*lhd Sts*)
 using *init initial-OLf-invariant by blast*

have *nnul*: $\neg \text{lnull Sts}$
 using *chain chain-not-lnull by blast*
hence *lhd-lmap*: $\bigwedge f. \text{lhd} (\text{lmap } f \text{ Sts}) = f (\text{lhd Sts})$
 by (*rule llist.map-sel(1)*)

have *active-of* (*lhd Sts*) = {}
 by (*metis is-initial-OLf-state.cases init snd-conv*)
hence *act*: *active-subset* (*lhd (lmap fstate Sts)*) = {}
 unfolding *active-subset-def lhd-lmap by (cases lhd Sts) auto*

have *pas*: *passive-subset* (*Liminf-llist (lmap fstate Sts)*) = {}
proof (*cases lfinite Sts*)
 case *fin*: *True*

have *lim*: *Liminf-llist (lmap fstate Sts)* = *fstate (llast Sts)*
 using *lfinite-Liminf-llist fin nnul*
 by (*metis chain-not-lnull il-chain lfinite-lmap llast-lmap*)

have *last-inv*: *OLf-invariant* (*llast Sts*)
 by (*rule chain-ILf-invariant-llast[OF chain inv fin]*)

have $\forall St'. \neg \text{llast Sts} \rightsquigarrow \text{ILf } St'$
 using *full-chain-lnth-not-rel[OF full]* **by** (*metis fin full-chain-iff-chain full*)
 hence *is-final-OLf-state* (*llast Sts*)
 unfolding *is-final-OLf-state-iff-no-ILf-step[OF last-inv]* .
 then obtain *A* :: '*f* *fset* **where**
 at-l: *llast Sts* = ({}|, *None*, *empty*, *None*, *A*)
 unfolding *is-final-OLf-state.simps by blast*

```

show ?thesis
  unfolding is-final-OLf-state.simps passive-subset-def lim at-l by auto
next
case False
hence len: llength Sts = ∞
  by (simp add: not-lfinite-llength)
show ?thesis
  unfolding Liminf-fstate-commute passive-subset-def Liminf-fstate-def
  using fair-IL-Liminf-new-empty[OF len full inv]
    fair-IL-Liminf-xx-empty[OF len full inv]
    fair-IL-Liminf-passive-empty[OF len full init]
    fair-IL-Liminf-yy-empty[OF full inv]
  by simp
qed

show saturated (state (Liminf-fstate Sts))
  using IL-Liminf-saturated act Liminf-fstate-commute il-chain pas by fastforce

{
  assume
    bot: B ∈ Bot-F and
    unsat: fset (new-of (lhd Sts)) ⊨∩G {B}

  have unsat': fst ' lhd (lmap fstate Sts) ⊨∩G {B}
    using unsat unfolding lhd-lmap by (cases lhd Sts) (auto intro: no-labels-entails-mono-left)

  have ∃ BL ∈ Bot-FL. BL ∈ Liminf-llist (lmap fstate Sts)
    using IL-complete-Liminf[OF il-chain act pas bot unsat'] .
  thus ∃ B' ∈ Bot-F. B' ∈ state-union (Liminf-fstate Sts)
    unfolding Liminf-fstate-def Liminf-fstate-commute by auto
  thus ∃ i. enat i < llength Sts ∧ (∃ B' ∈ Bot-F. B' ∈ all-formulas-of (lnth Sts i))
    unfolding Liminf-fstate-def Liminf-llist-def by auto
}
qed

end

end

```

10 Completeness of Fair Otter Loop

The Otter loop is a special case of the iProver loop, with fewer rules. We can therefore reuse the fair iProver loop's completeness result to derive the (dynamic) refutational completeness of the fair Otter loop.

```

theory Fair-Otter-Loop-Complete
  imports Fair-iProver-Loop
begin

```

10.1 Completeness

```

context fair-otter-loop
begin

```

```

theorem

```

```

assumes
  full: full-chain ( $\rightsquigarrow$ OLf) Sts and
  init: is-initial-OLf-state (lhd Sts)
shows
  fair-OL-Liminf-saturated: saturated (state (Liminf-fstate Sts)) and
  fair-OL-complete-Liminf:  $B \in \text{Bot-F} \implies \text{fset (new-of (lhd Sts))} \models_{\cap \mathcal{G}} \{B\} \implies$ 
     $\exists B' \in \text{Bot-F}. B' \in \text{state-union (Liminf-fstate Sts)}$  and
  fair-OL-complete:  $B \in \text{Bot-F} \implies \text{fset (new-of (lhd Sts))} \models_{\cap \mathcal{G}} \{B\} \implies$ 
     $\exists i. \text{enat } i < \text{llength Sts} \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of (lnth Sts } i))$ 
proof –
  have ilf-chain: chain ( $\rightsquigarrow$ ILf) Sts
  using Lazy-List-Chain.chain-mono fair-IL.ol full-chain-imp-chain full by blast
  hence ilf-full: full-chain ( $\rightsquigarrow$ ILf) Sts
  by (metis chain-ILf-invariant-llast full-chain-iff-chain initial-OLf-invariant
    is-final-OLf-state-iff-no-ILf-step is-final-OLf-state-iff-no-OLf-step full init)

  show saturated (state (Liminf-fstate Sts))
  by (rule fair-IL-Liminf-saturated[OF ilf-full init])

  {
  assume
    bot:  $B \in \text{Bot-F}$  and
    unsat:  $\text{fset (new-of (lhd Sts))} \models_{\cap \mathcal{G}} \{B\}$ 

    show  $\exists B' \in \text{Bot-F}. B' \in \text{state-union (Liminf-fstate Sts)}$ 
    by (rule fair-IL-complete-Liminf[OF ilf-full init bot unsat])
    show  $\exists i. \text{enat } i < \text{llength Sts} \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of (lnth Sts } i))$ 
    by (rule fair-IL-complete[OF ilf-full init bot unsat])
  }
qed

end

```

10.2 Specialization with FIFO Queue

As a proof of concept, we specialize the passive set to use a FIFO queue, thereby eliminating the locale assumptions about the passive set.

```

locale fifo-otter-loop =
  otter-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F
for
  Bot-F :: 'f set and
  Inf-F :: 'f inference set and
  Bot-G :: 'g set and
  Q :: 'q set and
  entails-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set  $\Rightarrow$  bool and
  Inf-G-q :: 'q  $\Rightarrow$  'g inference set and
  Red-I-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g inference set and
  Red-F-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set and
  G-F-q :: 'q  $\Rightarrow$  'f  $\Rightarrow$  'g set and
  G-I-q :: 'q  $\Rightarrow$  'f inference  $\Rightarrow$  'g inference set option and
  Equiv-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix <math>\langle \doteq \rangle 50) and
  Prec-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix <math>\langle \prec \rangle 50) +
fixes
  Prec-S :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix <math>\prec_S 50)
assumes

```

```

wf-Prec-S: minimal-element ( $\prec_S$ ) UNIV and
transp-Prec-S: transp ( $\prec_S$ ) and
finite-Inf-between: finite  $A \implies$  finite (no-labels.Inf-between  $A \{C\}$ )
begin

sublocale fifo-prover-queue
.

sublocale fair-otter-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q
Equiv-F Prec-F [] hd  $\lambda y$  xs. if  $y \in$  set xs then xs else xs @ [y] removeAll fset-of-list Prec-S
proof
show po-on ( $\prec_S$ ) UNIV
using wf-Prec-S minimal-element.po by blast
next
show wfp-on ( $\prec_S$ ) UNIV
using wf-Prec-S minimal-element.wf by blast
next
show transp ( $\prec_S$ )
by (rule transp-Prec-S)
next
show  $\bigwedge A C$ . finite  $A \implies$  finite (no-labels.Inf-between  $A \{C\}$ )
by (fact finite-Inf-between)
qed

end

end

```

11 Zipperposition Loop with Ghost State

The Zipperposition loop is a variant of the DISCOUNT loop that can cope with inferences generating (countably) infinitely many conclusions. The version formalized here has an additional ghost component D in its state tuple, which is used in the refinement proof from the abstract procedure *LGC*.

```

theory Zipperposition-Loop
imports DISCOUNT-Loop
begin

```

```

context discount-loop
begin

```

11.1 Basic Definitions and Lemmas

```

fun flat-inferences-of :: 'f inference llist multiset  $\implies$  'f inference set where
flat-inferences-of  $T = \bigcup \{lset \iota \mid \iota. \iota \in \# T\}$ 

```

```

fun
zl-state :: 'f inference llist multiset  $\times$  'f inference set  $\times$  'f set  $\times$  'f set  $\times$  'f set  $\implies$ 
'f inference set  $\times$  ('f  $\times$  DL-label) set

```

```

where
zl-state ( $T, D, P, Y, A$ ) = (flat-inferences-of  $T - D$ , labeled-formulas-of ( $P, Y, A$ ))

```

```

lemma zl-state-alt-def:
zl-state ( $T, D, P, Y, A$ ) =

```

(flat-inferences-of $T - D$, $(\lambda C. (C, \text{Passive})) \text{ ' } P \cup (\lambda C. (C, YY)) \text{ ' } Y \cup (\lambda C. (C, \text{Active})) \text{ ' } A$)
 by auto

inductive

ZL :: 'f inference set \times ('f \times DL-label) set \Rightarrow 'f inference set \times ('f \times DL-label) set \Rightarrow bool
 (infix \sim ZL 50)

where

compute-infer: $\iota 0 \in \text{no-labels.Red-I } (A \cup \{C\}) \Rightarrow$
 zl-state $(T + \{\#LCons \iota 0 \ \iota s\# \}, D, P, \{\}, A) \sim$ ZL zl-state $(T + \{\#\iota s\# \}, D \cup \{\iota 0\}, P \cup \{C\}, \{\}, A)$
 | choose-p: zl-state $(T, D, P \cup \{C\}, \{\}, A) \sim$ ZL zl-state $(T, D, P, \{C\}, A)$
 | delete-fwd: $C \in \text{no-labels.Red-F } A \vee (\exists C' \in A. C' \preceq C) \Rightarrow$
 zl-state $(T, D, P, \{C\}, A) \sim$ ZL zl-state $(T, D, P, \{\}, A)$
 | simplify-fwd: $C \in \text{no-labels.Red-F } (A \cup \{C'\}) \Rightarrow$
 zl-state $(T, D, P, \{C\}, A) \sim$ ZL zl-state $(T, D, P, \{C'\}, A)$
 | delete-bwd: $C' \in \text{no-labels.Red-F } \{C\} \vee C' \succ C \Rightarrow$
 zl-state $(T, D, P, \{C\}, A \cup \{C'\}) \sim$ ZL zl-state $(T, D, P, \{C\}, A)$
 | simplify-bwd: $C' \in \text{no-labels.Red-F } \{C, C''\} \Rightarrow$
 zl-state $(T, D, P, \{C\}, A \cup \{C'\}) \sim$ ZL zl-state $(T, D, P \cup \{C''\}, \{C\}, A)$
 | schedule-infer: flat-inferences-of $T' = \text{no-labels.Inf-between } A \ \{C\} \Rightarrow$
 zl-state $(T, D, P, \{C\}, A) \sim$ ZL zl-state $(T + T', D - \text{flat-inferences-of } T', P, \{\}, A \cup \{C\})$
 | delete-orphan-infers: lset $\iota s \cap \text{no-labels.Inf-from } A = \{\} \Rightarrow$
 zl-state $(T + \{\#\iota s\# \}, D, P, Y, A) \sim$ ZL zl-state $(T, D \cup \text{lset } \iota s, P, Y, A)$

11.2 Refinement

lemma zl-compute-infer-in-lgc:

assumes $\iota 0 \in \text{no-labels.Red-I } (A \cup \{C\})$
 shows zl-state $(T + \{\#LCons \iota 0 \ \iota s\# \}, D, P, \{\}, A) \sim$ LGC
 zl-state $(T + \{\#\iota s\# \}, D \cup \{\iota 0\}, P \cup \{C\}, \{\}, A)$

proof –

show ?thesis

proof (cases $\iota 0 \in D$)

case True

hence infs: flat-inferences-of $(T + \{\#LCons \iota 0 \ \iota s\# \}) - D =$
 flat-inferences-of $(T + \{\#\iota s\# \}) - (D \cup \{\iota 0\})$

by fastforce

show ?thesis

unfolding zl-state.simps infs

by (rule step.process[of - labeled-formulas-of $(P, \{\}, A) \ \{\} - \{(C, \text{Passive})\}$])
 (auto simp: active-subset-def)

next

case $i0\text{-ni: False}$

show ?thesis

unfolding zl-state.simps

proof (rule step.compute-infer[of - - $\iota 0$ - - $\{(C, \text{Passive})\}$])

show flat-inferences-of $(T + \{\#LCons \iota 0 \ \iota s\# \}) - D =$
 flat-inferences-of $(T + \{\#\iota s\# \}) - (D \cup \{\iota 0\}) \cup \{\iota 0\}$

using $i0\text{-ni}$ by fastforce

next

show labeled-formulas-of $(P \cup \{C\}, \{\}, A) = \text{labeled-formulas-of } (P, \{\}, A) \cup \{(C, \text{Passive})\}$
 by auto

next

show active-subset $\{(C, \text{Passive})\} = \{\}$

by (auto simp: active-subset-def)

next

show $\iota 0 \in \text{no-labels.Red-I-G } (\text{fst } ' (\text{labeled-formulas-of } (P, \{\}, A) \cup \{(C, \text{Passive})\}))$
by *simp* (*metis* (*no-types*) *Un-commute* *Un-empty-right* *Un-insert-right* *Un-upper1* *assms*
no-labels.empty-ord.Red-I-of-subset *subset-iff*)
qed
qed
qed

lemma *zl-choose-p-in-lgc*: $\text{zl-state } (T, D, P \cup \{C\}, \{\}, A) \rightsquigarrow \text{LGC } \text{zl-state } (T, D, P, \{C\}, A)$

proof –

let $?N = \text{labeled-formulas-of } (P, \{\}, A)$
and $?T = \text{flat-inferences-of } T - D$
have $\text{Passive} \sqsubseteq L \text{ YY}$
by (*simp* *add: DL-Prec-L-def*)
hence $(?T, ?N \cup \{(C, \text{Passive})\}) \rightsquigarrow \text{LGC } (?T, ?N \cup \{(C, \text{YY})\})$
using *relabel-inactive* **by** *blast*
hence $(?T, \text{labeled-formulas-of } (P \cup \{C\}, \{\}, A)) \rightsquigarrow \text{LGC } (?T, \text{labeled-formulas-of } (P, \{C\}, A))$
by (*metis* *PYA-add-passive-formula* *P0A-add-y-formula*)
thus *?thesis*
by *auto*
qed

lemma *zl-delete-fwd-in-lgc*:

assumes $C \in \text{no-labels.Red-F } A \vee (\exists C' \in A. C' \preceq C)$
shows $\text{zl-state } (T, D, P, \{C\}, A) \rightsquigarrow \text{LGC } \text{zl-state } (T, D, P, \{\}, A)$
using *assms*

proof

assume *c-in*: $C \in \text{no-labels.Red-F } A$
hence $A \subseteq \text{fst } ' \text{labeled-formulas-of } (P, \{\}, A)$
by *simp*
hence $C \in \text{no-labels.Red-F } (\text{fst } ' \text{labeled-formulas-of } (P, \{\}, A))$
by (*metis* (*no-types*, *lifting*) *c-in in-mono no-labels.Red-F-of-subset*)
thus *?thesis*
using *remove-redundant-no-label* **by** *auto*

next

assume $\exists C' \in A. C' \preceq C$
then obtain C' **where** *c'-in-and-c'-ls-c*: $C' \in A \wedge C' \preceq C$
by *auto*
hence $(C', \text{Active}) \in \text{labeled-formulas-of } (P, \{\}, A)$
by *auto*
moreover have $\text{YY} \sqsubseteq L \text{ Active}$
by (*simp* *add: DL-Prec-L-def*)
ultimately show *?thesis*
by (*metis* *P0A-add-y-formula* *remove-succ-L c'-in-and-c'-ls-c zl-state.simps*)
qed

lemma *zl-simplify-fwd-in-lgc*:

assumes $C \in \text{no-labels.Red-F-G } (A \cup \{C'\})$
shows $\text{zl-state } (T, D, P, \{C\}, A) \rightsquigarrow \text{LGC } \text{zl-state } (T, D, P, \{C'\}, A)$

proof –

let $?N = \text{labeled-formulas-of } (P, \{\}, A)$
and $?M = \{(C, \text{YY})\}$
and $?M' = \{(C', \text{YY})\}$
have $A \cup \{C'\} \subseteq \text{fst } ' (?N \cup ?M')$
by *auto*
hence $C \in \text{no-labels.Red-F-G } (\text{fst } ' (?N \cup ?M'))$

by (*smt* (*verit*, *ccfv-SIG*) *assms no-labels.Red-F-of-subset subset-iff*)
hence $(C, YY) \in \text{Red-F } (?N \cup ?M')$
 using *no-labels-Red-F-imp-Red-F* **by** *simp*
hence $?M \subseteq \text{Red-F-}\mathcal{G} (?N \cup ?M')$
 by *simp*
moreover have *active-subset* $?M' = \{\}$
 using *active-subset-def* **by** *auto*
ultimately have (*flat-inferences-of* $T - D$, *labeled-formulas-of* $(P, \{\}, A) \cup \{(C, YY)\}$) \rightsquigarrow *LGC*
 (*flat-inferences-of* $T - D$, *labeled-formulas-of* $(P, \{\}, A) \cup \{(C', YY)\}$)
 using *process*[*of* - - $?M - ?M'$] **by** *auto*
thus *?thesis*
 by *simp*
qed

lemma *zl-delete-bwd-in-lgc*:

assumes $C' \in \text{no-labels.Red-F-}\mathcal{G} \{C\} \vee C' \cdot \succ C$
shows *zl-state* $(T, D, P, \{C\}, A \cup \{C'\}) \rightsquigarrow$ *LGC* *zl-state* $(T, D, P, \{C\}, A)$
 using *assms*

proof

let $?N = \text{labeled-formulas-of } (P, \{C\}, A)$
assume *c'-in*: $C' \in \text{no-labels.Red-F-}\mathcal{G} \{C\}$

have $\{C\} \subseteq \text{fst}' ?N$
 by *simp*

hence $C' \in \text{no-labels.Red-F-}\mathcal{G} (\text{fst}' ?N)$

by (*metis* (*no-types*, *lifting*) *c'-in insert-Diff insert-subset no-labels.Red-F-of-subset*)

hence (*flat-inferences-of* $T - D$, $?N \cup \{(C', \text{Active})\}$) \rightsquigarrow *LGC* (*flat-inferences-of* $T - D$, $?N$)
 using *remove-redundant-no-label* **by** *auto*

moreover have $?N \cup \{(C', \text{Active})\} = \text{labeled-formulas-of } (P, \{C\}, A \cup \{C'\})$
 using *PYA-add-active-formula* **by** *blast*

ultimately have (*flat-inferences-of* $T - D$, *labeled-formulas-of* $(P, \{C\}, A \cup \{C'\})$) \rightsquigarrow *LGC*
zl-state $(T, D, P, \{C\}, A)$

by *simp*

thus *?thesis*

by *auto*

next

assume $C' \cdot \succ C$

moreover have $(C, YY) \in \text{labeled-formulas-of } (P, \{C\}, A)$

by *simp*

ultimately show *?thesis*

by (*metis* *remove-succ-F PYA-add-active-formula zl-state.simps*)

qed

lemma *zl-simplify-bwd-in-lgc*:

assumes $C' \in \text{no-labels.Red-F-}\mathcal{G} \{C, C''\}$

shows *zl-state* $(T, D, P, \{C\}, A \cup \{C'\}) \rightsquigarrow$ *LGC* *zl-state* $(T, D, P \cup \{C''\}, \{C\}, A)$

proof –

let $?M = \{(C', \text{Active})\}$

and $?M' = \{(C'', \text{Passive})\}$

and $?N = \text{labeled-formulas-of } (P, \{C\}, A)$

have $\{C, C''\} \subseteq \text{fst}' (?N \cup ?M')$

by *simp*

hence $C' \in \text{no-labels.Red-F-}\mathcal{G} (\text{fst}' (?N \cup ?M'))$

by (*smt* (*z3*) *DiffI Diff-eq-empty-iff assms empty-iff no-labels.Red-F-of-subset*)

hence \mathcal{M} -included: $?M \subseteq \text{Red-F-G } (?N \cup ?M')$
using *no-labels-Red-F-imp-Red-F* **by** *auto*
have *active-subset* $?M' = \{\}$
using *active-subset-def* **by** *auto*
hence (*flat-inferences-of* $T - D, ?N \cup ?M$) \rightsquigarrow *LGC* (*flat-inferences-of* $T - D, ?N \cup ?M'$)
using *M-included process*[*of* - - $?M - ?M'$] **by** *auto*
moreover have $?N \cup ?M = \text{labeled-formulas-of}(P, \{C\}, A \cup \{C'\})$ **and**
 $?N \cup ?M' = \text{labeled-formulas-of}(P \cup \{C''\}, \{C\}, A)$
by *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *zl-schedule-infer-in-lgc*:

assumes *flat-inferences-of* $T' = \text{no-labels.Inf-between } A \{C\}$
shows *zl-state* $(T, D, P, \{C\}, A) \rightsquigarrow$ *LGC*
 $\text{zl-state } (T + T', D - \text{flat-inferences-of } T', P, \{\}, A \cup \{C\})$

proof –

let $?N = \text{labeled-formulas-of } (P, \{\}, A)$
have *fst* ‘ *active-subset* $?N = A$
by (*meson prj-active-subset-of-state*)
hence *infs*: *flat-inferences-of* $T' = \text{no-labels.Inf-between } (\text{fst} \text{ ‘ } \text{active-subset } ?N) \{C\}$
using *assms* **by** *simp*

have *inf*: (*flat-inferences-of* $T - D, ?N \cup \{(C, YY)\}$) \rightsquigarrow *LGC*
 $((\text{flat-inferences-of } T - D) \cup \text{flat-inferences-of } T', ?N \cup \{(C, \text{Active})\})$
by (*rule step.schedule-infer*[*of* - - *flat-inferences-of* $T' - ?N C YY$]) (*use infs in auto*)

have *m-bef*: *labeled-formulas-of* $(P, \{C\}, A) = ?N \cup \{(C, YY)\}$
by *auto*

have *t-aft*: *flat-inferences-of* $(T + T') - (D - \text{flat-inferences-of } T') =$
 $(\text{flat-inferences-of } T - D) \cup \text{flat-inferences-of } T'$
by *auto*

have *m-aft*: *labeled-formulas-of* $(P, \{\}, A \cup \{C\}) = ?N \cup \{(C, \text{Active})\}$
by *auto*

show *?thesis*
unfolding *zl-state.simps m-bef t-aft m-aft* **using** *inf* .

qed

lemma *zl-delete-orphan-infers-in-lgc*:

assumes *inter*: $\text{lset } \iota s \cap \text{no-labels.Inf-from } A = \{\}$
shows *zl-state* $(T + \{\#\iota s\# \}, D, P, Y, A) \rightsquigarrow$ *LGC* *zl-state* $(T, D \cup \text{lset } \iota s, P, Y, A)$

proof –

let $?N = \text{labeled-formulas-of } (P, Y, A)$

have *inf*: (*flat-inferences-of* $T \cup \text{lset } \iota s - D, ?N$)
 \rightsquigarrow *LGC* (*flat-inferences-of* $T - (D \cup \text{lset } \iota s), ?N$)
by (*rule step.delete-orphan-infers*[*of* - - $\text{lset } \iota s - D$])
(use inter prj-active-subset-of-state in auto)

have *t-bef*: *flat-inferences-of* $(T + \{\#\iota s\# \}) - D = \text{flat-inferences-of } T \cup \text{lset } \iota s - D$
by *auto*

show *?thesis*
unfolding *zl-state.simps t-bef* **using** *inf* .

qed

theorem *ZL-step-imp-LGC-step*: $St \rightsquigarrow ZL St' \implies St \rightsquigarrow LGC St'$
proof (*induction rule*: *ZL.induct*)
 case (*compute-infer* $\iota_0 A C T \iota_s D P$)
 thus *?case*
 using *zl-compute-infer-in-lgc* **by** *auto*
next
 case (*choose-p* $T D P C A$)
 thus *?case*
 using *zl-choose-p-in-lgc* **by** *auto*
next
 case (*delete-fwd* $C A T D P$)
 thus *?case*
 using *zl-delete-fwd-in-lgc* **by** *auto*
next
 case (*simplify-fwd* $C A C' T D P$)
 thus *?case*
 using *zl-simplify-fwd-in-lgc* **by** *blast*
next
 case (*delete-bwd* $C' C T D P A$)
 thus *?case*
 using *zl-delete-bwd-in-lgc* **by** *blast*
next
 case (*simplify-bwd* $C' C C'' T D P A$)
 thus *?case*
 using *zl-simplify-bwd-in-lgc* **by** *blast*
next
 case (*schedule-infer* $T' A C T D P$)
 thus *?case*
 using *zl-schedule-infer-in-lgc* **by** *blast*
next
 case (*delete-orphan-infers* $\iota_s A T D P Y$)
 thus *?case*
 using *zl-delete-orphan-infers-in-lgc* **by** *auto*
qed

11.3 Completeness

theorem

assumes

zl-chain: $\text{chain } (\rightsquigarrow ZL) Sts$ **and**

act: $\text{active-subset } (\text{snd } (\text{lhd } Sts)) = \{\}$ **and**

pas: $\text{passive-subset } (\text{Liminf-llist } (\text{lmap } \text{snd } Sts)) = \{\}$ **and**

no-prems-init: $\forall \iota \in \text{Inf-F}. \text{prems-of } \iota = [] \implies \iota \in \text{fst } (\text{lhd } Sts)$ **and**

final-sched: $\text{Liminf-llist } (\text{lmap } \text{fst } Sts) = \{\}$

shows

ZL-Liminf-saturated: $\text{saturated } (\text{Liminf-llist } (\text{lmap } \text{snd } Sts))$ **and**

ZL-complete-Liminf: $B \in \text{Bot-F} \implies \text{fst } ' \text{snd } (\text{lhd } Sts) \models \cap \mathcal{G} \{B\} \implies$

$\exists BL \in \text{Bot-FL}. BL \in \text{Liminf-llist } (\text{lmap } \text{snd } Sts)$ **and**

ZL-complete: $B \in \text{Bot-F} \implies \text{fst } ' \text{snd } (\text{lhd } Sts) \models \cap \mathcal{G} \{B\} \implies$

$\exists i. \text{enat } i < \text{llength } Sts \wedge (\exists BL \in \text{Bot-FL}. BL \in \text{snd } (\text{lnth } Sts \ i))$

proof –

have *lgc-chain*: $\text{chain } (\rightsquigarrow LGC) Sts$

using *zl-chain ZL-step-imp-LGC-step chain-mono* **by** *blast*

show *saturated* $(\text{Liminf-llist } (\text{lmap } \text{snd } Sts))$

using *act final-sched lgc.fair-implies-Liminf-saturated lgc-chain lgc-fair lgc-to-red*
no-prems-init pas **by** *blast*

```
{
  assume
    bot: B ∈ Bot-F and
    unsat: fst ‘ snd (lhd Sts) ⊨NG {B}

  show ZL-complete-Liminf: ∃ BL ∈ Bot-FL. BL ∈ Liminf-llist (lmap snd Sts)
    by (rule lgc-complete-Liminf[OF lgc-chain act pas no-prems-init final-sched bot unsat])
  thus OL-complete: ∃ i. enat i < llength Sts ∧ (∃ BL ∈ Bot-FL. BL ∈ snd (lnth Sts i))
    unfolding Liminf-llist-def by auto
}
qed

end

end
```

12 Prover Lazy List Queues and Fairness

This section covers the to-do data structure that arises in the Zipperposition loop.

```
theory Prover-Lazy-List-Queue
imports Prover-Queue
begin
```

12.1 Basic Lemmas

lemma *ne-and-in-set-take-imp-in-set-take-remove1*:

```
assumes
  z ≠ y and
  z ∈ set (take m xs)
shows z ∈ set (take m (remove1 y xs))
using assms
proof (induct xs arbitrary: m)
case (Cons x xs)
note ih = this(1) and z-ne-y = this(2) and z-in-take-xs = this(3)

show ?case
proof (cases z = x)
  case True
  thus ?thesis
    by (metis (no-types, lifting) List.hd-in-set gr-zeroI hd-take in-set-remove1 list.sel(1)
      remove1.simps(2) take-eq-Nil z-in-take-xs z-ne-y)
next
  case z-ne-x: False

  have z-in-take-xs: z ∈ set (take m xs)
    using z-in-take-xs z-ne-x
    by (smt (verit, del-insts) butlast-take in-set-butlastD in-set-takeD le-cases3 set-ConsD
      take-Cons' take-all)

  show ?thesis
proof (cases y = x)
```

```

case y-eq-x: True
show ?thesis
  using y-eq-x by (simp add: z-in-take-xs)
next
case y-ne-x: False

have m > 0
  by (metis gr0I list.set-cases list.simps(3) take-Cons' z-in-take-xs)
then obtain m' :: nat where
  m: m = Suc m'
  using gr0-implies-Suc by presburger

have z-in-take-xs': z ∈ set (take m' xs)
  using z-in-take-xs z-in-take-xs z-ne-x by (simp add: m)
note ih = ih[OF z-ne-y z-in-take-xs']

show ?thesis
  using y-ne-x ih unfolding m by simp
qed
qed
qed simp

```

12.2 Locales

```

locale prover-lazy-list-queue =
  fixes
    empty :: 'q and
    add-llist :: 'e llist ⇒ 'q ⇒ 'q and
    remove-llist :: 'e llist ⇒ 'q ⇒ 'q and
    pick-elem :: 'q ⇒ 'e × 'q and
    llists :: 'q ⇒ 'e llist multiset
  assumes
    llists-empty[simp]: llists empty = {#} and
    llists-not-empty: Q ≠ empty ⇒ llists Q ≠ {#} and
    llists-add[simp]: llists (add-llist es Q) = llists Q + {#es#} and
    llist-remove[simp]: llists (remove-llist es Q) = llists Q - {#es#} and
    llists-pick-elem: (∃ es ∈# llists Q. es ≠ LNil) ⇒
      ∃ e es. LCons e es ∈# llists Q ∧ fst (pick-elem Q) = e
      ∧ llists (snd (pick-elem Q)) = llists Q - {#LCons e es#} + {#es#}
begin

```

```

abbreviation has-elem :: 'q ⇒ bool where
  has-elem Q ≡ ∃ es ∈# llists Q. es ≠ LNil

```

```

inductive lqueue-step :: 'q × 'e set ⇒ 'q × 'e set ⇒ bool where
  lqueue-step-fold-add-llistI:
    lqueue-step (Q, D) (fold add-llist ess Q, D - ∪ {lset es | es. es ∈ set ess})
  | lqueue-step-fold-remove-llistI:
    lqueue-step (Q, D) (fold remove-llist ess Q, D ∪ ∪ {lset es | es. es ∈ set ess})
  | lqueue-step-pick-elemI: has-elem Q ⇒
    lqueue-step (Q, D) (snd (pick-elem Q), D ∪ {fst (pick-elem Q)})

```

```

lemma lqueue-step-idleI: lqueue-step QD QD
  using lqueue-step-fold-add-llistI [of fst QD snd QD [], simplified] .

```

```

lemma lqueue-step-add-llistI: lqueue-step (Q, D) (add-llist es Q, D - lset es)

```

using *lqueue-step-fold-add-llistI*[of - - [es], *simplified*] .

lemma *lqueue-step-remove-llistI*: *lqueue-step* (Q, D) (remove-llist es Q, D \cup lset es)
using *lqueue-step-fold-remove-llistI*[of - - [es], *simplified*] .

lemma *llists-fold-add-llist[simp]*: *llists* (fold add-llist es Q) = *mset* es + *llists* Q
by (induct es arbitrary: Q) auto

lemma *llists-fold-remove-llist[simp]*: *llists* (fold remove-llist es Q) = *llists* Q - *mset* es
by (induct es arbitrary: Q) auto

inductive *pick-lqueue-step-w-details* :: 'q \times 'e set \Rightarrow 'e \Rightarrow 'e llist \Rightarrow 'q \times 'e set \Rightarrow bool **where**
pick-lqueue-step-w-detailsI: LCons e es $\in\#$ *llists* Q \Longrightarrow fst (*pick-elem* Q) = e \Longrightarrow
llists (snd (*pick-elem* Q)) = *llists* Q - {#LCons e es#} + {#es#} \Longrightarrow
pick-lqueue-step-w-details (Q, D) e es (snd (*pick-elem* Q), D \cup {e})

inductive *pick-lqueue-step* :: 'q \times 'e set \Rightarrow 'q \times 'e set \Rightarrow bool **where**
pick-lqueue-stepI: *pick-lqueue-step-w-details* QD e es QD' \Longrightarrow *pick-lqueue-step* QD QD'

inductive
remove-lqueue-step-w-details :: 'q \times 'e set \Rightarrow 'e llist list \Rightarrow 'q \times 'e set \Rightarrow bool
where
remove-lqueue-step-w-detailsI:
remove-lqueue-step-w-details (Q, D) ess
(fold remove-llist ess Q, D \cup \bigcup {lset es | es. es \in set ess})

end

locale *fair-prover-lazy-list-queue* =
prover-lazy-list-queue empty add-llist remove-llist *pick-elem* *llists*
for
empty :: 'q **and**
add-llist :: 'e llist \Rightarrow 'q \Rightarrow 'q **and**
remove-llist :: 'e llist \Rightarrow 'q \Rightarrow 'q **and**
pick-elem :: 'q \Rightarrow 'e \times 'q **and**
llists :: 'q \Rightarrow 'e llist multiset +
assumes *fair*: chain *lqueue-step* QDs \Longrightarrow infinitely-often *pick-lqueue-step* QDs \Longrightarrow
LCons e es $\in\#$ *llists* (fst (lnth QDs i)) \Longrightarrow
 $\exists j \geq i. (\exists$ ess. LCons e es \in set ess
 \wedge *remove-lqueue-step-w-details* (lnth QDs j) ess (lnth QDs (Suc j)))
 \vee *pick-lqueue-step-w-details* (lnth QDs j) e es (lnth QDs (Suc j))
begin

lemma *fair-strong*:
assumes
chain: chain *lqueue-step* QDs **and**
inf: infinitely-often *pick-lqueue-step* QDs **and**
es-in: es $\in\#$ *llists* (fst (lnth QDs i)) **and**
k-lt: enat k < llength es
shows $\exists j \geq i.$
($\exists k' \leq k. \exists$ ess. ldrop k' es \in set ess
 \wedge *remove-lqueue-step-w-details* (lnth QDs j) ess (lnth QDs (Suc j)))
 \vee *pick-lqueue-step-w-details* (lnth QDs j) (lnth es k) (ldrop (Suc k) es) (lnth QDs (Suc j))
using *k-lt*
proof (induct k)

```

case 0
note zero-lt = this
have es-in': LCons (lnth es 0) (ldrop (Suc 0) es) ∈# llists (fst (lnth QDs i))
  using es-in by (metis zero-lt ldrop-0 ldrop-enat ldropn-Suc-conv-ldropn zero-enat-def)
show ?case
  using fair[OF chain inf es-in']
  by (metis dual-order.refl ldrop-enat ldropn-Suc-conv-ldropn zero-lt)
next
case (Suc k)
note ih = this(1) and sk-lt = this(2)

have k-lt: enat k < llength es
  using sk-lt Suc-ile-eq order-less-imp-le by blast

obtain j :: nat where
  j-ge: j ≥ i and
  rem-or-pick-step: (∃ k' ≤ k. ∃ ess. ldrop (enat k') es ∈ set ess
    ∧ remove-lqueue-step-w-details (lnth QDs j) ess (lnth QDs (Suc j)))
  ∨ pick-lqueue-step-w-details (lnth QDs j) (lnth es k) (ldrop (enat (Suc k)) es)
    (lnth QDs (Suc j))
  using ih[OF k-lt] by blast

{
  assume ∃ k' ≤ k. ∃ ess. ldrop (enat k') es ∈ set ess
    ∧ remove-lqueue-step-w-details (lnth QDs j) ess (lnth QDs (Suc j))
  hence ?case
    using j-ge le-SucI by blast
}
moreover
{
  assume pick-lqueue-step-w-details (lnth QDs j) (lnth es k) (ldrop (enat (Suc k)) es)
    (lnth QDs (Suc j))
  hence cons-in: LCons (lnth es (Suc k)) (ldrop (enat (Suc (Suc k))) es)
    ∈# llists (fst (lnth QDs (Suc j)))
  unfolding pick-lqueue-step-w-details.simps using sk-lt
  by (metis fst-conv ldrop-enat ldropn-Suc-conv-ldropn union-mset-add-mset-right
    union-single-eq-member)

  have ?case
    using fair[OF chain inf cons-in] j-ge
    by (smt (z3) dual-order.trans ldrop-enat ldropn-Suc-conv-ldropn le-Suc-eq sk-lt)
}
ultimately show ?case
  using rem-or-pick-step by blast
qed
end

```

12.3 Instantiation with FIFO Queue

As a proof of concept, we show that a FIFO queue can serve as a fair prover lazy list queue.

```
type-synonym 'e fifo = nat × ('e × 'e llist) list
```

```
locale fifo-prover-lazy-list-queue
begin
```


definition *empty* :: 'e fifo **where**

empty = (0, [])

fun *add-llist* :: 'e llist \Rightarrow 'e fifo \Rightarrow 'e fifo **where**

add-llist LNil (num-nils, ps) = (num-nils + 1, ps)

| *add-llist* (LCons e es) (num-nils, ps) = (num-nils, ps @ [(e, es)])

fun *remove-llist* :: 'e llist \Rightarrow 'e fifo \Rightarrow 'e fifo **where**

remove-llist LNil (num-nils, ps) = (num-nils - 1, ps)

| *remove-llist* (LCons e es) (num-nils, ps) = (num-nils, remove1 (e, es) ps)

fun *pick-elem* :: 'e fifo \Rightarrow 'e \times 'e fifo **where**

pick-elem (-, []) = undefined

| *pick-elem* (num-nils, (e, es) # ps) =

(e,

(case es of

LNil \Rightarrow (num-nils + 1, ps)

| LCons e' es' \Rightarrow (num-nils, ps @ [(e', es')]))))

fun *llists* :: 'e fifo \Rightarrow 'e llist multiset **where**

llists (num-nils, ps) = replicate-mset num-nils LNil + mset (map (λ (e, es). LCons e es) ps)

sublocale *prover-lazy-list-queue empty add-llist remove-llist pick-elem llists*

proof

show *llists empty* = {#}

unfolding *empty-def* **by** *simp*

next

fix *Q* :: 'e fifo

assume *nemp*: *Q* \neq *empty*

thus *llists Q* \neq {#}

proof (*cases Q*)

case *q*: (Pair num-nils ps)

show *?thesis*

using *nemp* **unfolding** *q empty-def* **by** *auto*

qed

next

fix *es* :: 'e llist **and** *Q* :: 'e fifo

show *llists (add-llist es Q)* = *llists Q* + {#es#}

by (*cases Q*, *cases es*) *auto*

next

fix *es* :: 'e llist **and** *Q* :: 'e fifo

show *llists (remove-llist es Q)* = *llists Q* - {#es#}

proof (*cases Q*)

case *q*: (Pair num-nils ps)

show *?thesis*

proof (*cases es*)

case LNil

note *es* = *this*

have *inter-emp*: {#LCons x y. (x, y) \in # mset ps#} \cap # {#LNil#} = {#}

by *auto*

show *?thesis*

proof (*cases num-nils*)

case *num-nils*: 0

have *nil-ni*: LNil \notin # {#LCons x y. (x, y) \in # mset ps#}

```

    by auto
  show ?thesis
    unfolding q es num-nils by (auto simp: diff-single-trivial[OF nil-ni])
next
  case num-nils: (Suc n)
  show ?thesis
    unfolding q es num-nils by auto
qed
next
  case (LCons e es')
  note es = this
  show ?thesis
  proof (cases (e, es') ∈# mset ps)
    case pair-in: True
    show ?thesis
      unfolding q es using pair-in by (auto simp: multiset-union-diff-assoc image-mset-Diff)
  next
    case pair-ni: False
    have cons-ni:
      LCons e es' ∉# replicate-mset num-nils LNil + {#LCons x y. (x, y) ∈# mset ps#}
      using pair-ni by auto
    show ?thesis
      unfolding q es using pair-ni cons-ni by (auto simp: diff-single-trivial)
  qed
qed
qed
next
  fix Q :: 'e fifo
  assume nnil: ∃ es ∈# llists Q. es ≠ LNil
  show ∃ e es. LCons e es ∈# llists Q ∧ fst (pick-elem Q) = e ∧ llists (snd (pick-elem Q)) = llists Q
  - {#LCons e es#} + {#es#}
  using nnil
  proof (cases Q)
    case q: (Pair num-nils ps)
    show ?thesis
    proof (cases ps)
      case ps: Nil
      have False
        using nnil unfolding q ps by (cases num-nils = 0) auto
      thus ?thesis
        by blast
    next
      case ps: (Cons p ps')
      show ?thesis
      proof (rule exI[of - fst p], rule exI[of - snd p]; intro conjI)
        show LCons (fst p) (snd p) ∈# llists Q
          unfolding q ps by (cases p) auto
      next
        show fst (pick-elem Q) = fst p
          unfolding q ps by (cases p) auto
      next
        show llists (snd (pick-elem Q)) = llists Q - {#LCons (fst p) (snd p)#} + {#snd p#}
      proof (cases p)
        case p: (Pair e es)
        show ?thesis

```

```

proof (cases es)
  case es: LNil
    show ?thesis
      unfolding q ps p es by simp
  next
    case es: (LCons e' es')
      show ?thesis
        unfolding q ps p es by simp
  qed
qed
qed
qed
qed
qed
qed

sublocale fair-prover-lazy-list-queue empty add-llist remove-llist pick-elem llists
proof
  fix
    QDs :: ('e fifo × 'e set) llist and
    e :: 'e and
    es :: 'e llist and
    i :: nat
  assume
    chain: chain lqueue-step QDs and
    inf-pick: infinitely-often pick-lqueue-step QDs and
    cons-in: LCons e es ∈# llists (fst (lnth QDs i))

  have len: llength QDs = ∞
  using inf-pick unfolding infinitely-often-alt-def
  by (metis Suc-ile-eq dual-order.strict-implies-order enat.exhaust enat-ord-simps(2)
    verit-comp-simplify1(3))

  {
  assume not-rem-step:  $\neg (\exists j \geq i. \exists \text{ess}. \text{LCons } e \text{ es} \in \text{set } \text{ess}$ 
     $\wedge \text{remove-lqueue-step-w-details } (\text{lnth } QDs \ j) \ \text{ess} \ (\text{lnth } QDs \ (\text{Suc } j)))$ 

  obtain num-nils :: nat and ps :: ('e × 'e llist) list where
    fst-at-i: fst (lnth QDs i) = (num-nils, ps)
    by fastforce

  obtain k :: nat where
    k-lt: k < length (snd (fst (lnth QDs i))) and
    at-k: snd (fst (lnth QDs i)) ! k = (e, es)
    using cons-in unfolding fst-at-i
    by simp (smt (verit) empty-iff imageE in-set-conv-nth llist.distinct(1) llist.inject
      prod.collapse singleton-iff split-beta)

  have  $\forall k' \leq k. \exists i' \geq i. (e, es) \in \text{set } (\text{take } (\text{Suc } k') \ (\text{snd } (\text{fst } (\text{lnth } QDs \ i'))))$ 
  proof -
    have  $\exists i' \geq i. (e, es) \in \text{set } (\text{take } (k + 1 - l) \ (\text{snd } (\text{fst } (\text{lnth } QDs \ i'))))$ 
      if l-le: l ≤ k for l
      using l-le
    proof (induct l)
      case 0
      show ?case

```

```

proof (rule exI[of - i]; simp)
  show  $(e, es) \in \text{set } (\text{take } (\text{Suc } k) (\text{snd } (\text{fst } (\text{lnth } \text{QDs } i))))$ 
    by (simp add: at-k k-lt take-Suc-conv-app-nth)
qed
next
case (Suc l)
note  $ih = \text{this}(1)$  and  $sl-le = \text{this}(2)$ 

have  $l-le-k: l \leq k$ 
  using  $sl-le$  by linarith
note  $ih = ih[OF\ l-le-k]$ 

obtain  $i' :: \text{nat}$  where
   $i'-ge: i' \geq i$  and
   $\text{cons-at-}i': (e, es) \in \text{set } (\text{take } (k + 1 - l) (\text{snd } (\text{fst } (\text{lnth } \text{QDs } i'))))$ 
  using  $ih$  by blast
then obtain  $j0 :: \text{nat}$  where
   $j0 \geq i'$  and
   $\text{pick-lqueue-step } (\text{lnth } \text{QDs } j0) (\text{lnth } \text{QDs } (\text{Suc } j0))$ 
  using  $\text{inf-pick unfolding infinitely-often-alt-def}$  by auto
then obtain  $j :: \text{nat}$  where
   $j-ge: j \geq i'$  and
   $\text{pick-step: pick-lqueue-step } (\text{lnth } \text{QDs } j) (\text{lnth } \text{QDs } (\text{Suc } j))$  and
   $\text{pick-step-min:}$ 
   $\forall j'. j' \geq i' \longrightarrow j' < j \longrightarrow \neg \text{pick-lqueue-step } (\text{lnth } \text{QDs } j') (\text{lnth } \text{QDs } (\text{Suc } j'))$ 
  using  $\text{wfP-exists-minimal}[OF\ \text{wfP-less, of}$ 
     $\lambda j. j \geq i' \wedge \text{pick-lqueue-step } (\text{lnth } \text{QDs } j) (\text{lnth } \text{QDs } (\text{Suc } j))\ j0\ \lambda j. j]$ 
  by blast

have  $\text{cons-at-le-}j: (e, es) \in \text{set } (\text{take } (k + 1 - l) (\text{snd } (\text{fst } (\text{lnth } \text{QDs } j'))))$ 
  if  $j'-ge: j' \geq i'$  and  $j'-le: j' \leq j$  for  $j'$ 
proof -
  have  $(e, es) \in \text{set } (\text{take } (k + 1 - l) (\text{snd } (\text{fst } (\text{lnth } \text{QDs } (i' + m))))$ 
    if  $i'm-le: i' + m \leq j$  for  $m$ 
    using  $i'm-le$ 
  proof (induct m)
    case 0
    then show ?case
      using  $\text{cons-at-}i'$  by fastforce
  next
  case (Suc m)
  note  $ih = \text{this}(1)$  and  $i'sm-le = \text{this}(2)$ 

  have  $i'm-lt: i' + m < j$ 
    using  $i'sm-le$  by linarith
  have  $i'm-le: i' + m \leq j$ 
    using  $i'sm-le$  by linarith
  note  $ih = ih[OF\ i'm-le]$ 

  have  $\text{step: lqueue-step } (\text{lnth } \text{QDs } (i' + m)) (\text{lnth } \text{QDs } (i' + \text{Suc } m))$ 
    by (simp add: chain chain-lnth-rel len)

  show ?case
    using  $\text{step}$ 
  proof cases

```

```

case (lqueue-step-fold-add-llistI Q D ess)
note defs = this

have in-set-fold-add: (e, es) ∈ set (take n (snd (fold add-llist ess Q)))
  if (e, es) ∈ set (take n (snd Q)) for n
  using that
proof (induct ess arbitrary: Q)
  case (Cons es' ess')
  note ih = this(1) and in-q = this(2)

  have in-add: (e, es) ∈ set (take n (snd (add-llist es' Q)))
  proof (cases Q)
    case q: (Pair num-nils ps)
    show ?thesis
    proof (cases es')
      case es': LNil
      show ?thesis
      using in-q unfolding q es' by simp
    next
      case es': (LCons e'' es'')
      show ?thesis
      using in-q unfolding q es' by simp
    qed
  qed

  show ?case
  using ih[OF in-add] by simp
qed simp

show ?thesis
  using ih unfolding defs by (auto intro: in-set-fold-add)
next
case (lqueue-step-fold-remove-llistI Q D ess)
note defs = this

have notin-set-remove: (e, es) ∈ set (take n (snd (fold remove-llist ess Q)))
  if LCons e es ∉ set ess and (e, es) ∈ set (take n (snd Q)) for n
  using that
proof (induct ess arbitrary: Q)
  case (Cons es' ess')
  note ih = this(1) and ni-es'ess' = this(2) and in-q = this(3)
  have ni-ess': LCons e es ∉ set ess'
  using ni-es'ess' by auto
  have in-rem: (e, es) ∈ set (take n (snd (remove-llist es' Q)))
  by (smt (verit, best) fifo-prover-lazy-list-queue.remove-llist.elims fst-conv in-q
    list.set-intros(1) ne-and-in-set-take-imp-in-set-take-remove1 ni-es'ess'
    snd-conv)
  show ?case
  using ih[OF ni-ess' in-rem] by auto
qed simp

have remove-lqueue-step-w-details (lnth QDs (i' + m)) ess (lnth QDs (i' + Suc m))
  unfolding defs by (rule remove-lqueue-step-w-detailsI)
hence LCons e es ∉ set ess
  using not-rem-step i'-ge by force

```

```

thus ?thesis
  using ih unfolding defs by (auto intro: notin-set-remove)
next
case (lqueue-step-pick-elemI Q D)
note defs = this(1,2) and rest = this(3)

have pick-lqueue-step (lnth QDs (i' + m)) (lnth QDs (i' + Suc m))
proof –
  have  $\exists e \text{ es. } \textit{pick-lqueue-step-w-details (lnth QDs (i' + m)) e es}$ 
    (lnth QDs (i' + Suc m))
  unfolding defs using pick-lqueue-step-w-detailsI
  by (metis add-Suc-right llists-pick-elem lqueue-step-pick-elemI(2) rest)
  thus ?thesis
  using pick-lqueue-stepI by fast
qed
moreover have  $\neg \textit{pick-lqueue-step (lnth QDs (i' + m)) (lnth QDs (i' + Suc m))}$ 
  using pick-step-min[rule-format, OF le-add1 i'm-lt] by simp
ultimately show ?thesis
  by blast
qed
qed
thus ?thesis
  by (metis j'-ge j'-le nat-le-iff-add)
qed

show ?case
proof (cases hd (snd (fst (lnth QDs j))) = (e, es))
case eq-ees: True
show ?thesis
proof (rule exI[of - j]; intro conjI)
  show  $i \leq j$ 
  using i'-ge j-ge le-trans by blast
next
show  $(e, es) \in \textit{set (take (k + 1 - Suc l) (snd (fst (lnth QDs j))))}$ 
  by (metis (no-types, lifting) List.hd-in-set Suc-eq-plus1 cons-at-le-j diff-is-0-eq
    eq-ees hd-take j-ge le-imp-less-Suc nle-le not-less-eq-eq sl-le take-eq-Nil2
    zero-less-diff)
qed
next
case ne-ees: False
show ?thesis
proof (rule exI[of - Suc j], intro conjI)
  show  $i \leq \textit{Suc j}$ 
  using i'-ge j-ge by linarith
next
obtain  $Q :: 'e \text{ fifo}$  and  $D :: 'e \text{ set}$  and  $e' :: 'e$  and  $es' :: 'e \text{ llist}$  where
  at-j: lnth QDs j = (Q, D) and
  at-sj: lnth QDs (Suc j) = (snd (pick-elem Q), D  $\cup$  {e'}) and
  pair-in: LCons e' es'  $\in$  # llists Q and
  fst: fst (pick-elem Q) = e' and
  snd: llists (snd (pick-elem Q)) = llists Q - {#LCons e' es'#} + {#es'#}
  using pick-step unfolding pick-lqueue-step.simps pick-lqueue-step-w-details.simps
  by blast

have cons-at-j: (e, es)  $\in$  set (take (k + 1 - l) (snd (fst (lnth QDs j))))

```

```

using cons-at-le-j[of j] j-ge by blast

show  $(e, es) \in \text{set } (\text{take } (k + 1 - \text{Suc } l) (\text{snd } (\text{fst } (\text{lnth } \text{QDs } (\text{Suc } j))))))$ 
proof (cases Q)
  case q: (Pair num-nils ps)
  show ?thesis
  proof (cases ps)
    case Nil
    hence False
    using at-j cons-at-j q by force
    thus ?thesis
    by blast
  next
  case ps: (Cons p' ps')
  show ?thesis
  proof (cases p')
    case p': (Pair e' es')

    have hd-at-j:  $\text{hd } (\text{snd } (\text{fst } (\text{lnth } \text{QDs } j))) = (e', es')$ 
      by (simp add: at-j p' ps q)

    show ?thesis
    proof (cases es')
      case es': LNil
      show ?thesis
      using cons-at-j ne-ees Suc-diff-le l-le-k
      unfolding q ps p' es' at-j at-sj hd-at-j by force
    next
    case es': (LCons e'' es'')
    show ?thesis
    using cons-at-j ne-ees Suc-diff-le l-le-k
    unfolding q ps p' es' at-j at-sj hd-at-j by force
  qed
qed
qed
qed
qed
qed
qed
qed
thus ?thesis
  by (metis Suc-eq-plus1 add-right-mono diff-Suc-Suc diff-diff-cancel diff-le-self)
qed
then obtain i' :: nat where
  i'-ge:  $i' \geq i$  and
  cons-at-i':  $(e, es) \in \text{set } (\text{take } 1 (\text{snd } (\text{fst } (\text{lnth } \text{QDs } i'))))$ 
  by auto
then obtain j0 :: nat where
  j0  $\geq i'$  and
  pick-lqueue-step  $(\text{lnth } \text{QDs } j0) (\text{lnth } \text{QDs } (\text{Suc } j0))$ 
  using inf-pick unfolding infinitely-often-alt-def by auto
then obtain j :: nat where
  j-ge:  $j \geq i'$  and
  pick-step: pick-lqueue-step  $(\text{lnth } \text{QDs } j) (\text{lnth } \text{QDs } (\text{Suc } j))$  and
  pick-step-min:
   $\forall j'. j' \geq i' \longrightarrow j' < j \longrightarrow \neg \text{pick-lqueue-step } (\text{lnth } \text{QDs } j') (\text{lnth } \text{QDs } (\text{Suc } j'))$ 

```

```

using wfP-exists-minimal[OF wfP-less, of
   $\lambda j. j \geq i' \wedge \text{pick-lqueue-step } (\text{lnth } QDs\ j) (\text{lnth } QDs\ (\text{Suc } j))\ j0\ \lambda j. j]$ 
by blast
hence pick-step-det:  $\exists e\ es. \text{pick-lqueue-step-w-details } (\text{lnth } QDs\ j)\ e\ es\ (\text{lnth } QDs\ (\text{Suc } j))$ 
unfolding pick-lqueue-step.simps by simp
have pick-lqueue-step-w-details  $(\text{lnth } QDs\ j)\ e\ es\ (\text{lnth } QDs\ (\text{Suc } j))$ 
proof –
have cons-at-j:  $(e, es) \in \text{set } (\text{take } 1\ (\text{snd } (\text{fst } (\text{lnth } QDs\ j))))$ 
proof –
have  $(e, es) \in \text{set } (\text{take } 1\ (\text{snd } (\text{fst } (\text{lnth } QDs\ (i' + l))))$  if  $i'l\text{-le}: i' + l \leq j$  for  $l$ 
using  $i'l\text{-le}$ 
proof (induct  $l$ )
case (Suc  $l$ )
note  $ih = \text{this}(1)$  and  $i'sl\text{-le} = \text{this}(2)$ 

have  $i'l\text{-lt}: i' + l < j$ 
using  $i'sl\text{-le}$  by linarith
have  $i'l\text{-le}: i' + l \leq j$ 
using  $i'sl\text{-le}$  by linarith
note  $ih = ih[\text{OF } i'l\text{-le}]$ 

have step:  $\text{lqueue-step } (\text{lnth } QDs\ (i' + l))\ (\text{lnth } QDs\ (i' + \text{Suc } l))$ 
by (simp add: chain chain-lnth-rel len)

show ?case
using step
proof cases
case (lqueue-step-fold-add-llistI  $Q\ D\ ess$ )
note defs = this

have len-q:  $\text{length } (\text{snd } Q) \geq 1$ 
using  $ih$  by (metis Suc-eq-plus1 add.commute empty-iff le-add1 length-0-conv
  list.set(1) list-decode.cases local.lqueue-step-fold-add-llistI(1) prod.sel(1)
  take.simps(1))

have take:  $\text{take } (\text{Suc } 0)\ (\text{snd } (\text{fold } \text{add-llist } \text{ess } Q)) = \text{take } (\text{Suc } 0)\ (\text{snd } Q)$ 
using len-q
proof (induct  $ess$  arbitrary:  $Q$ )
case Nil
show ?case
by (cases  $Q$ ) auto
next
case (Cons  $es'\ ess'$ )
note  $ih = \text{this}(1)$  and  $\text{len-q} = \text{this}(2)$ 

have len-add:  $\text{length } (\text{snd } (\text{add-llist } \text{es}'\ Q)) \geq 1$ 
proof (cases  $Q$ )
case  $q: (\text{Pair } \text{num-nils } ps)$ 
show ?thesis
proof (cases  $es'$ )
case  $es': \text{LNil}$ 
show ?thesis
using len-q unfolding  $q\ es'$  by simp
next
case  $es': (\text{LCons } e''\ es'')$ 

```



```

    show ?thesis
      using len-q unfolding q es' by simp
    qed
  qed

  note ih = ih[OF len-add]

  show ?case
    using len-q by (simp add: ih, cases Q, cases es', auto)
  qed

  show ?thesis
    unfolding defs using ih take
    by simp (metis local.lqueue-step-fold-add-llistI(1) prod.sel(1))
next
  case (lqueue-step-fold-remove-llistI Q D ess)
  note defs = this

  have remove-lqueue-step-w-details (lnth QDs (i' + l)) ess (lnth QDs (i' + Suc l))
    unfolding defs by (rule remove-lqueue-step-w-detailsI)
  moreover have  $\neg (\exists \text{ess}. LCons\ e\ es \in \text{set}\ \text{ess}$ 
     $\wedge \text{remove-lqueue-step-w-details}\ (\text{lnth}\ \text{QDs}\ (i' + l))\ \text{ess}\ (\text{lnth}\ \text{QDs}\ (i' + \text{Suc}\ l)))$ 
    using not-rem-step add-Suc-right i'-ge trans-le-add1 by presburger
  ultimately have ees-ni:  $LCons\ e\ es \notin \text{set}\ \text{ess}$ 
    by blast

  obtain ps' :: ('e × 'e llist) list where
    snd-q:  $\text{snd}\ Q = (e, es) \# ps'$ 
    using ih by (metis (no-types, opaque-lifting) One-nat-def fst-eqD in-set-member
      in-set-takeD length-pos-if-in-set list.exhaust-sel
      lqueue-step-fold-remove-llistI(1) member-rec(1) member-rec(2) nth-Cons-0 take0
      take-Suc-conv-app-nth)

  obtain num-nils' :: nat where
    q:  $Q = (\text{num-nils}', (e, es) \# ps')$ 
    by (metis prod.collapse snd-q)

  have take-1:  $\text{take}\ 1\ (\text{snd}\ (\text{fold}\ \text{remove-llist}\ \text{ess}\ Q)) = \text{take}\ 1\ (\text{snd}\ Q)$ 
    unfolding q using ees-ni
  proof (induct ess arbitrary: num-nils' ps')
    case (Cons es' ess')
    note ih = this(1) and ees-ni = this(2)

    have ees-ni':  $LCons\ e\ es \notin \text{set}\ \text{ess}'$ 
      using ees-ni by simp
    note ih = ih[OF ees-ni']

    have es'-ne:  $es' \neq LCons\ e\ es$ 
      using ees-ni by auto

  show ?case
  proof (cases es')
    case LNil
    then show ?thesis
      using ih by auto
  end

```

```

next
  case es': (LCons e'' es'')
  show ?thesis
    using ih es'-ne unfolding es' by auto
qed
qed auto

show ?thesis
  unfolding defs using ih take-1
  by simp (metis lqueue-step-fold-remove-llistI(1) prod.sel(1))
next
case (lqueue-step-pick-elimI Q D)
note defs = this(1,2) and rest = this(3)

have pick-lqueue-step (lnth QDs (i' + l)) (lnth QDs (Suc (i' + l)))
proof -
  have  $\exists e es. \text{pick-lqueue-step-w-details (lnth QDs (i' + l)) e es}$ 
    (lnth QDs (Suc (i' + l)))
  unfolding defs using pick-lqueue-step-w-detailsI
  by (metis add-Suc-right llists-pick-elim lqueue-step-pick-elimI(2) rest)
  thus ?thesis
    using pick-lqueue-stepI by fast
qed
moreover have  $\neg \text{pick-lqueue-step (lnth QDs (i' + l)) (lnth QDs (Suc (i' + l)))}$ 
  using pick-step-min[rule-format, OF le-add1 i'l-lt] .
ultimately show ?thesis
  by blast
qed
qed (use cons-at-i' in auto)
thus ?thesis
  by (metis dual-order.refl j-ge nat-le-iff-add)
qed
hence cons-in-fst:  $(e, es) \in \text{set (snd (fst (lnth QDs j)))}$ 
  using in-set-takeD by force

obtain ps' :: ('e × 'e llist) list where
  fst-at-j:  $\text{snd (fst (lnth QDs j))} = (e, es) \# ps'$ 
  using cons-at-j by (metis One-nat-def cons-in-fst empty-iff empty-set length-pos-if-in-set
    list.set-cases nth-Cons-0 self-append-conv2 set-ConsD take0 take-Suc-conv-app-nth)

have fst-pick:  $\text{fst (pick-elim (fst (lnth QDs j)))} = e$ 
  using fst-at-j by (metis fst-conv pick-elim.simps(2) surjective-pairing)
have snd-pick:  $\text{llists (snd (pick-elim (fst (lnth QDs j))))} =$ 
   $\text{llists (fst (lnth QDs j))} - \{\#LCons e es\} + \{\#es\}$ 
  by (subst (1 2) surjective-pairing[of fst (lnth QDs j)], unfold fst-at-j, cases es, auto)

obtain Q :: 'e fifo and D :: 'e set where
  at-j:  $\text{lnth QDs j} = (Q, D)$ 
  by fastforce

show ?thesis
  unfolding pick-lqueue-step-w-details.simps
proof (rule exI[of - e], rule exI[of - es], rule exI[of - Q], rule exI[of - D], intro conjI)
  show  $\text{lnth QDs (Suc j)} = (\text{snd (pick-elim Q)}, D \cup \{e\})$ 
  by (smt (verit, best) at-j fst-conv fst-pick pick-lqueue-step-w-details.simps)

```

```

      pick-step-det snd-conv)
next
  have LCons e es ∈# llists (fst (lnth QDs j))
    by (subst surjective-pairing) (auto simp: fst-at-j)
  thus LCons e es ∈# llists Q
    unfolding at-j by simp
next
  show fst (pick-lem Q) = e
    using at-j fst-pick by force
next
  show llists (snd (pick-lem Q)) = llists Q - {#LCons e es#} + {#es#}
    using at-j snd-pick by fastforce
qed (rule refl at-j)+
qed
hence ∃ j ≥ i. pick-lqueue-step-w-details (lnth QDs j) e es (lnth QDs (Suc j))
  using i'-ge j-ge le-trans by blast
}
thus ∃ j ≥ i.
  (∃ ess. LCons e es ∈ set ess ∧ remove-lqueue-step-w-details (lnth QDs j) ess (lnth QDs (Suc j)))
  ∨ pick-lqueue-step-w-details (lnth QDs j) e es (lnth QDs (Suc j))
  by blast
qed
end
end
end

```

13 Fair Zipperposition Loop with Ghosts

```

theory Fair-Zipperposition-Loop
imports
  Given-Clause-Loops-Util
  Zipperposition-Loop
  Prover-Lazy-List-Queue
begin

```

The fair Zipperposition loop makes assumptions about the scheduled inference queue and the passive clause queue and ensures (dynamic) refutational completeness under these assumptions. This version inherits the ghost state component from the “unfair” version of the loop.

13.1 Locale

```

type-synonym ('t, 'p, 'f) ZLf-state = 't × 'f inference set × 'p × 'f option × 'f fset

```

```

locale fair-zipperposition-loop =
  discount-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F +
  todo: fair-prover-lazy-list-queue t-empty t-add-llist t-remove-llist t-pick-lem t-llists +
  passive: fair-prover-queue p-empty p-select p-add p-remove p-felems
for
  Bot-F :: 'f set and
  Inf-F :: 'f inference set and
  Bot-G :: 'g set and
  Q :: 'q set and
  entails-q :: 'q ⇒ 'g set ⇒ 'g set ⇒ bool and
  Inf-G-q :: 'q ⇒ 'g inference set and

```

Red-I-q :: 'q ⇒ 'g set ⇒ 'g inference set **and**
Red-F-q :: 'q ⇒ 'g set ⇒ 'g set **and**
G-F-q :: 'q ⇒ 'f ⇒ 'g set **and**
G-I-q :: 'q ⇒ 'f inference ⇒ 'g inference set option **and**
Equiv-F :: 'f ⇒ 'f ⇒ bool (**infix** ≐ 50) **and**
Prec-F :: 'f ⇒ 'f ⇒ bool (**infix** <· 50) **and**
t-empty :: 't **and**
t-add-llist :: 'f inference llist ⇒ 't ⇒ 't **and**
t-remove-llist :: 'f inference llist ⇒ 't ⇒ 't **and**
t-pick-elem :: 't ⇒ 'f inference × 't **and**
t-llists :: 't ⇒ 'f inference llist multiset **and**
p-empty :: 'p **and**
p-select :: 'p ⇒ 'f **and**
p-add :: 'f ⇒ 'p ⇒ 'p **and**
p-remove :: 'f ⇒ 'p ⇒ 'p **and**
p-felems :: 'p ⇒ 'f fset +
fixes
Prec-S :: 'f ⇒ 'f ⇒ bool (**infix** <S 50)
assumes
wf-Prec-S: minimal-element (<S) UNIV **and**
transp-Prec-S: transp (<S) **and**
countable-Inf-between: finite A ⇒ countable (no-labels.Inf-between A {C})
begin

lemma *trans-Prec-S*: trans {(x, y). x <S y}
using *transp-Prec-S transp-trans* **by** blast

lemma *irreflp-Prec-S*: irreflp (<S)
using minimal-element.wf wfp-imp-irreflp wf-Prec-S wfp-on-UNIV **by** blast

lemma *irrefl-Prec-S*: irrefl {(x, y). x <S y}
by (metis CollectD case-prod-conv irrefl-def irreflp-Prec-S irreflp-def)

13.2 Basic Definitions and Lemmas

abbreviation *todo-of* :: ('t, 'p, 'f) ZLf-state ⇒ 't **where**
todo-of St ≡ fst St

abbreviation *done-of* :: ('t, 'p, 'f) ZLf-state ⇒ 'f inference set **where**
done-of St ≡ fst (snd St)

abbreviation *passive-of* :: ('t, 'p, 'f) ZLf-state ⇒ 'p **where**
passive-of St ≡ fst (snd (snd St))

abbreviation *yy-of* :: ('t, 'p, 'f) ZLf-state ⇒ 'f option **where**
yy-of St ≡ fst (snd (snd (snd St)))

abbreviation *active-of* :: ('t, 'p, 'f) ZLf-state ⇒ 'f fset **where**
active-of St ≡ snd (snd (snd (snd St)))

abbreviation *all-formulas-of* :: ('t, 'p, 'f) ZLf-state ⇒ 'f set **where**
all-formulas-of St ≡ passive.elems (passive-of St) ∪ set-option (yy-of St) ∪ fset (active-of St)

fun *zl-fstate* :: ('t, 'p, 'f) ZLf-state ⇒ 'f inference set × ('f × DL-label) set **where**
zl-fstate (T, D, P, Y, A) = *zl-state* (t-llists T, D, passive.elems P, set-option Y, fset A)

lemma *zl-fstate-alt-def*:
zl-fstate St = *zl-state* (t-llists (fst St), fst (snd St), passive.elems (fst (snd (snd St))),
set-option (fst (snd (snd (snd St)))), fset (snd (snd (snd (snd St))))))
by (cases St) auto

definition

$Liminf\text{-}z\text{-}f\text{-}state :: ('t, 'p, 'f) \text{ZLf}\text{-}state \text{ llist} \Rightarrow 'f \text{ set} \times 'f \text{ set} \times 'f \text{ set}$

where

$Liminf\text{-}z\text{-}f\text{-}state \text{ Sts} =$
 $(Liminf\text{-}l\text{-}list (lmap (passive.\text{elems} \circ passive\text{-}of) \text{ Sts}),$
 $Liminf\text{-}l\text{-}list (lmap (set\text{-}option \circ yy\text{-}of) \text{ Sts}),$
 $Liminf\text{-}l\text{-}list (lmap (fset \circ active\text{-}of) \text{ Sts}))$

lemma *Liminf-zl-fstate-commute:*

$Liminf\text{-}l\text{-}list (lmap (snd \circ z\text{-}f\text{-}state) \text{ Sts}) = \text{labeled}\text{-}formulas\text{-}of (Liminf\text{-}z\text{-}f\text{-}state \text{ Sts})$

proof –

have $Liminf\text{-}l\text{-}list (lmap (snd \circ z\text{-}f\text{-}state) \text{ Sts}) =$
 $(\lambda C. (C, Passive)) \text{ ' } Liminf\text{-}l\text{-}list (lmap (passive.\text{elems} \circ passive\text{-}of) \text{ Sts}) \cup$
 $(\lambda C. (C, YY)) \text{ ' } Liminf\text{-}l\text{-}list (lmap (set\text{-}option \circ yy\text{-}of) \text{ Sts}) \cup$
 $(\lambda C. (C, Active)) \text{ ' } Liminf\text{-}l\text{-}list (lmap (fset \circ active\text{-}of) \text{ Sts})$
unfolding *z\text{-}f\text{-}state\text{-}alt\text{-}def z\text{-}l\text{-}state\text{-}alt\text{-}def*
apply *simp*
apply (*subst Liminf\text{-}l\text{-}list\text{-}lmap\text{-}union, fast*)
apply (*subst Liminf\text{-}l\text{-}list\text{-}lmap\text{-}image, simp add: inj\text{-}on\text{-}convol\text{-}ident*)
by *auto*
thus *?thesis*
unfolding *Liminf\text{-}z\text{-}f\text{-}state\text{-}def* **by** *fastforce*

qed**fun** *formulas\text{-}union :: 'f set \times 'f set \times 'f set \Rightarrow 'f set* **where**

$formulas\text{-}union (P, Y, A) = P \cup Y \cup A$

inductive

$fair\text{-}ZL :: ('t, 'p, 'f) \text{ZLf}\text{-}state \Rightarrow ('t, 'p, 'f) \text{ZLf}\text{-}state \Rightarrow \text{bool}$ (**infix** $\sim ZL$ 50)

where

compute\text{-}infer: $(\exists \iota s \in \# \text{ t}\text{-}llists \ T. \iota s \neq LNil) \Longrightarrow \text{ t}\text{-}pick\text{-}elem \ T = (\iota 0, T') \Longrightarrow$
 $\iota 0 \in \text{ no}\text{-}labels.\text{Red}\text{-}I (fset \ A \cup \{C\}) \Longrightarrow$
 $(T, D, P, None, A) \sim ZL (T', D \cup \{\iota 0\}, p\text{-}add \ C \ P, None, A)$

| *choose\text{-}p*: $P \neq p\text{-}empty \Longrightarrow$

$(T, D, P, None, A) \sim ZL (T, D, p\text{-}remove (p\text{-}select \ P) \ P, Some (p\text{-}select \ P), A)$

| *delete\text{-}fwd*: $C \in \text{ no}\text{-}labels.\text{Red}\text{-}F (fset \ A) \vee (\exists C' \in fset \ A. C' \preceq C) \Longrightarrow$

$(T, D, P, Some \ C, A) \sim ZL (T, D, P, None, A)$

| *simplify\text{-}fwd*: $C' \prec_S C \Longrightarrow C \in \text{ no}\text{-}labels.\text{Red}\text{-}F (fset \ A \cup \{C'\}) \Longrightarrow$

$(T, D, P, Some \ C, A) \sim ZL (T, D, P, Some \ C', A)$

| *delete\text{-}bwd*: $C' \not\in A \Longrightarrow C' \in \text{ no}\text{-}labels.\text{Red}\text{-}F \{C\} \vee C' \succ_C C \Longrightarrow$

$(T, D, P, Some \ C, A \cup \{C'\}) \sim ZL (T, D, P, Some \ C, A)$

| *simplify\text{-}bwd*: $C' \not\in A \Longrightarrow C'' \prec_S C' \Longrightarrow C' \in \text{ no}\text{-}labels.\text{Red}\text{-}F \{C, C''\} \Longrightarrow$

$(T, D, P, Some \ C, A \cup \{C'\}) \sim ZL (T, D, p\text{-}add \ C'' \ P, Some \ C, A)$

| *schedule\text{-}infer*: $\text{ flat}\text{-}inferences\text{-}of (mset \ \iota ss) = \text{ no}\text{-}labels.\text{Inf}\text{-}between (fset \ A) \{C\} \Longrightarrow$

$(T, D, P, Some \ C, A) \sim ZL$

$(\text{ fold } \text{ t}\text{-}add\text{-}l\text{-}list \ \iota ss \ T, D - \text{ flat}\text{-}inferences\text{-}of (mset \ \iota ss), P, None, A \cup \{C\})$

| *delete\text{-}orphan\text{-}infs*: $\iota s \in \# \text{ t}\text{-}llists \ T \Longrightarrow \text{ lset } \iota s \cap \text{ no}\text{-}labels.\text{Inf}\text{-}from (fset \ A) = \{\} \Longrightarrow$

$(T, D, P, Y, A) \sim ZL (\text{ t}\text{-}remove\text{-}l\text{-}list \ \iota s \ T, D \cup \text{ lset } \iota s, P, Y, A)$

inductive *compute\text{-}infer\text{-}step :: ('t, 'p, 'f) \text{ZLf}\text{-}state \Rightarrow ('t, 'p, 'f) \text{ZLf}\text{-}state \Rightarrow bool* **where**

$(\exists \iota s \in \# \text{ t}\text{-}llists \ T. \iota s \neq LNil) \Longrightarrow \text{ t}\text{-}pick\text{-}elem \ T = (\iota 0, T') \Longrightarrow$

$\iota 0 \in \text{ no}\text{-}labels.\text{Red}\text{-}I (fset \ A \cup \{C\}) \Longrightarrow$

$\text{ compute}\text{-}infer\text{-}step (T, D, P, None, A) (T', D \cup \{\iota 0\}, p\text{-}add \ C \ P, None, A)$

The step below is slightly more general than the corresponding step in ($\sim ZL$), in the way it

handles the D component. The extra generality simplifies an argument later, when we erase the D “ghost” component of the state.

inductive *choose-p-step* :: ('t, 'p, 'f) ZLf-state \Rightarrow ('t, 'p, 'f) ZLf-state \Rightarrow bool **where**
 $P \neq p\text{-empty} \implies$
choose-p-step (T, D, P, None, A) (T, D', p-remove (p-select P) P, Some (p-select P), A)

13.3 Initial State and Invariant

inductive *is-initial-ZLf-state* :: ('t, 'p, 'f) ZLf-state \Rightarrow bool **where**
flat-inferences-of (mset ι ss) = no-labels.Inf-from {} \implies
is-initial-ZLf-state (fold t-add-llist ι ss t-empty, {}, p-empty, None, {||})

inductive *ZLf-invariant* :: ('t, 'p, 'f) ZLf-state \Rightarrow bool **where**
flat-inferences-of (t-llists T) \subseteq Inf-F \implies *ZLf-invariant* (T, D, P, Y, A)

lemma *initial-ZLf-invariant*:

assumes *is-initial-ZLf-state* St
shows *ZLf-invariant* St
using *assms*

proof

fix ι ss

assume

st : St = (fold t-add-llist ι ss t-empty, {}, p-empty, None, {||}) **and**
 ι ss: *flat-inferences-of* (mset ι ss) = no-labels.Inf-from {}

have *flat-inferences-of* (t-llists (fold t-add-llist ι ss t-empty)) \subseteq Inf-F

using ι ss no-labels.Inf-if-Inf-from **by** force

thus *ZLf-invariant* St

unfolding st **using** *ZLf-invariant.intros* **by** blast

qed

lemma *step-ZLf-invariant*:

assumes

inv : *ZLf-invariant* St **and**

$step$: St \sim ZLf St'

shows *ZLf-invariant* St'

using $step$ inv

proof *cases*

case (*compute-infer* T ι 0 T' A C D P)

note $defs = this(1,2)$ **and** $has\text{-}el = this(3)$ **and** $pick = this(4)$

have t' : T' = snd (t-pick-elem T)

using $pick$ **by** simp

obtain $\iota s'$ **where**

ι 0 $\iota s'$ -in: LCons ι 0 $\iota s' \in \#$ t-llists T **and**

$lists\text{-}t'$: t-llists T' = t-llists T - {#LCons ι 0 $\iota s'\#$ } + {# $\iota s'\#$ }

using *todo.llists-pick-elem[OF has-el, folded t']* $pick$ **by** auto

let ?II = {lset ιs | $\iota s. \iota s \in \#$ t-llists T}

let ?I = \bigcup ?II

have \bigcup {lset ιs | $\iota s. \iota s \in \#$ t-llists T - {#LCons ι 0 $\iota s'\#$ } + {# $\iota s'\#$ }} =

(\bigcup {lset ιs | $\iota s. \iota s \in \#$ t-llists T - {#LCons ι 0 $\iota s'\#$ }}) \cup lset $\iota s'$

by auto

also have ... $\subseteq (\bigcup \{lset\ \iota s \mid \iota s. \iota s \in \# \text{ t-llists } T - \{\#LCons\ \iota 0\ \iota s'\#\}\}) \cup \{\iota 0\} \cup lset\ \iota s'$
unfolding *lists-t'*
by *auto*
also have ... $\subseteq ?I \cup \{\iota 0\} \cup lset\ \iota s'$
proof –
have $\bigcup \{lset\ \iota s \mid \iota s. \iota s \in \# \text{ t-llists } T - \{\#LCons\ \iota 0\ \iota s'\#\}\} \subseteq \bigcup \{lset\ \iota s \mid \iota s. \iota s \in \# \text{ t-llists } T\}$
using *Union-Setcompr-member-mset-mono*[of *t-llists* *T* – $\{\#LCons\ \iota 0\ \iota s'\#\}$ *t-llists* *T* *lset*]
by *auto*
thus *?thesis*
by *blast*
qed
also have ... $\subseteq ?I$
proof –
have $\iota 0 \in ?I$
using *todo.llists-pick-elem*[*OF* *has-el*, *folded* *t'*] *pick* **by** *auto*
moreover have $lset\ \iota s' \subseteq ?I$
using *todo.llists-pick-elem*[*OF* *has-el*, *folded* *t'*] *pick* $\iota 0 \iota s'$ -*in* **by** *auto*
ultimately show *?thesis*
by *blast*
qed
finally show *?thesis*
using *inv unfolding* *defs ZLf-invariant.simps* **by** (*simp add: lists-t'*)
next
case (*schedule-infer* $\iota ss\ A\ C\ T\ D\ P$)
note *defs = this(1,2)* **and** $\iota ss\text{-inf-betw} = \text{this}(3)$
have $\bigcup \{lset\ \iota \mid \iota. \iota \in \text{set } \iota ss\} \subseteq \text{Inf-}F$
using $\iota ss\text{-inf-betw}$ **unfolding** *no-labels.Inf-between-def* *no-labels.Inf-from-def* **by** *auto*
thus *?thesis*
using *inv unfolding* *defs ZLf-invariant.simps* **by** *simp blast*
next
case (*delete-orphan-infers* $\iota s\ T\ A\ D\ P\ Y$)
note *defs = this(1,2)*
have $\bigcup \{lset\ \iota \mid \iota. \iota \in \# \text{ t-llists } T - \{\#\iota s\#\}\} \subseteq \bigcup \{lset\ \iota \mid \iota. \iota \in \# \text{ t-llists } T\}$
using *Union-Setcompr-member-mset-mono*[of *t-llists* *T* – $\{\#\iota s\#\}$ *t-llists* *T* *lset*] **by** *auto*
thus *?thesis*
using *inv unfolding* *defs ZLf-invariant.simps* **by** *simp*
qed (*auto simp: ZLf-invariant.simps*)

lemma *chain-ZLf-invariant-lnth*:
assumes
chain: chain ($\rightsquigarrow ZLf$) *Sts* **and**
fair-hd: ZLf-invariant (*lhd* *Sts*) **and**
i-lt: enat $i < \text{llength } Sts$
shows *ZLf-invariant* (*lnth* *Sts* i)
using *i-lt*
proof (*induct* i)
case 0
thus *?case*
using *fair-hd lhd-conv-lnth zero-enat-def* **by** *fastforce*
next
case (*Suc* i)
note $ih = \text{this}(1)$ **and** $si\text{-lt} = \text{this}(2)$

have $\text{enat } i < \text{llength } Sts$
using *si-lt Suc-ile-eq nless-le* **by** *blast*

hence *inv-i*: *ZLf-invariant (lnth Sts i)*
by (*rule ih*)
have *step*: *lnth Sts i* \rightsquigarrow *ZLf lnth Sts (Suc i)*
using *chain chain-lnth-rel si-lt* **by** *blast*

show *?case*
by (*rule step-ZLf-invariant[OF inv-i step]*)
qed

lemma *chain-ZLf-invariant-llast*:
assumes
chain: *chain (\rightsquigarrow ZLf) Sts* **and**
fair-hd: *ZLf-invariant (lhd Sts)* **and**
fin: *lfinite Sts*
shows *ZLf-invariant (llast Sts)*

proof –
obtain *i* :: *nat* **where**
i: *llength Sts = enat i*
using *lfinite-llength-enat[OF fin]* **by** *blast*

have *im1-lt*: *enat (i - 1) < llength Sts*
using *i* **by** (*metis chain chain-length-pos diff-less enat-ord-simps(2) less-numeral-extra(1) zero-enat-def*)

show *?thesis*
using *chain-ZLf-invariant-lnth[OF chain fair-hd im1-lt]*
by (*metis Suc-diff-1 chain chain-length-pos eSuc-enat enat-ord-simps(2) i llast-conv-lnth zero-enat-def*)
qed

13.4 Final State

inductive *is-final-ZLf-state* :: (*'t, 'p, 'f*) *ZLf-state* \Rightarrow *bool* **where**
is-final-ZLf-state (t-empty, D, p-empty, None, A)

lemma *is-final-ZLf-state-iff-no-ZLf-step*:
assumes *inv*: *ZLf-invariant St*
shows *is-final-ZLf-state St* \longleftrightarrow ($\forall St'. \neg St \rightsquigarrow$ *ZLf St'*)

proof
assume *is-final-ZLf-state St*
then obtain *D* :: *'f inference set* **and** *A* :: *'f fset* **where**
st: *St = (t-empty, D, p-empty, None, A)*
by (*auto simp: is-final-ZLf-state.simps*)
show $\forall St'. \neg St \rightsquigarrow$ *ZLf St'*
unfolding *st*
proof (*intro allI notI*)
fix *St'*
assume (*t-empty, D, p-empty, None, A*) \rightsquigarrow *ZLf St'*
thus *False*
by *cases auto*
qed

next
assume *no-step*: $\forall St'. \neg St \rightsquigarrow$ *ZLf St'*
show *is-final-ZLf-state St*
proof (*rule ccontr*)
assume *not-fin*: \neg *is-final-ZLf-state St*


```

obtain  $T :: 't$  and  $D :: 'f$  inference set and  $P :: 'p$  and  $Y :: 'f$  option and
 $A :: 'f$  fset where
 $st: St = (T, D, P, Y, A)$ 
by (cases St)

have  $T \neq t\text{-empty} \vee P \neq p\text{-empty} \vee Y \neq None$ 
using not-fin unfolding st is-final-ZLf-state.simps by auto
moreover {
assume
 $t: T \neq t\text{-empty}$  and
 $y: Y = None$ 

have  $\exists St'. St \sim ZLf St'$ 
proof (cases todo.has-elim T)
case has-el: True

obtain  $\iota 0 :: 'f$  inference and  $T' :: 't$  where
 $pick: t\text{-pick-elim } T = (\iota 0, T')$ 
by fastforce

obtain  $\iota s'$  where
 $\iota 0 \iota s'\text{-in}: LCons \iota 0 \iota s' \in \# t\text{-llists } T$  and
 $lists\text{-}t': t\text{-llists } T' = t\text{-llists } T - \{\#LCons \iota 0 \iota s'\#\} + \{\#\iota s'\#\}$ 
using todo.llists-pick-elim[OF has-el] pick by auto

have  $\iota 0 \in \bigcup \{lset \iota \mid \iota. \iota \in \# t\text{-llists } T\}$ 
using  $\iota 0 \iota s'\text{-in}$  by auto
hence  $\iota 0 \in Inf\text{-}F$ 
using inv t unfolding st ZLf-invariant.simps by auto
hence  $\iota 0\text{-red}: \iota 0 \in no\text{-labels.Red-I-}\mathcal{G} (fset A \cup \{concl\text{-of } \iota 0\})$ 
by (simp add: no-labels.empty-ord.Red-I-of-Inf-to-N)

show ?thesis
using fair-ZL.compute-infer[OF has-el pick \iota 0-red] unfolding st y by blast
next
case has-no-el: False

have  $nil\text{-in}: LNil \in \# t\text{-llists } T$ 
by (metis has-no-el multiset-nonemptyE t todo.llists-not-empty)
have  $nil\text{-inter}: lset LNil \cap no\text{-labels.Inf-from } (fset A) = \{\}$ 
by simp

show ?thesis
using fair-ZL.delete-orphan-infers[OF nil-in nil-inter] unfolding st t y by fast
qed
}
moreover
{
assume
 $p: P \neq p\text{-empty}$  and
 $y: Y = None$ 

have  $\exists St'. St \sim ZLf St'$ 
using fair-ZL.choose-p[OF p] unfolding st p y by fast

```

```

}
moreover
{
  assume  $Y \neq \text{None}$ 
  then obtain  $C :: 'f$  where
     $y: Y = \text{Some } C$ 
    by blast

  obtain  $\iota s :: 'f$  inference llist where
     $\iota s: \text{flat-inferences-of } (\text{mset } [\iota s]) = \text{no-labels.Inf-between } (\text{fset } A) \{C\}$ 
    using countable-imp-lset[OF countable-Inf-between[OF finite-fset]] by force

  have  $\exists St'. St \sim_{\text{ZLf}} St'$ 
    using fair-ZL.schedule-infer[OF  $\iota s$ ] unfolding  $st\ y$  by fast
} ultimately show False
using no-step by force
qed
qed

```

13.5 Refinement

lemma *fair-ZL-step-imp-ZL-step*:

```

assumes  $zlf: (T, D, P, Y, A) \sim_{\text{ZLf}} (T', D', P', Y', A')$ 
shows  $zlfstate\ (T, D, P, Y, A) \sim_{\text{ZL}} zlfstate\ (T', D', P', Y', A')$ 
using zlf

```

proof *cases*

```

case (compute-infer  $\iota 0\ C$ )
note  $defs = \text{this}(1-5)$  and  $has-el = \text{this}(6)$  and  $pick = \text{this}(7)$  and  $\iota-red = \text{this}(8)$ 

```

obtain $\iota s'$ **where**

```

 $\iota 0 \iota s'$ -in:  $LCons\ \iota 0\ \iota s' \in \#$  t-llists  $T$  and
lists-t': t-llists  $T' = \text{t-llists } T - \{\#LCons\ \iota 0\ \iota s'\#\} + \{\#\iota s'\#\}$ 
using todo.llists-pick-elem[OF has-el] pick by auto

```

show *?thesis*

```

unfolding  $defs$  zlfstate-alt-def prod.sel option.set lists-t'
using  $ZL.\text{compute-infer}[OF\ \iota-red, \text{of } \text{t-llists } T - \{\#LCons\ \iota 0\ \iota s'\#\}\ \iota s'\ D\ \text{passive.elements } P]$ 
 $\iota 0 \iota s'$ -in
by auto

```

next

```

case choose-p
note  $defs = \text{this}(1-6)$  and  $p-nemp = \text{this}(7)$ 

```

have *elems-rem-sel-uni-sel*:

```

 $\text{passive.elements } (p\text{-remove } (p\text{-select } P)\ P) \cup \{p\text{-select } P\} = \text{passive.elements } P$ 
using  $p-nemp$  by force

```

show *?thesis*

```

unfolding  $defs$  zlfstate-alt-def prod.sel option.set
using  $ZL.\text{choose-p}[\text{of } \text{t-llists } T\ D\ \text{passive.elements } (p\text{-remove } (p\text{-select } P)\ P)\ p\text{-select } P]$ 
 $\text{fset } A]$ 
by (metis elems-rem-sel-uni-sel)

```

next

```

case (delete-fwd C)
note  $defs = \text{this}(1-6)$  and  $c-red = \text{this}(7)$ 
show ?thesis

```

```

    unfolding defs zl-fstate-alt-def using ZL.delete-fwd[OF c-red] by simp
next
case (simplify-fwd C' C)
note defs = this(1-6) and c-red = this(8)
show ?thesis
    unfolding defs zl-fstate-alt-def using ZL.simplify-fwd[OF c-red] by simp
next
case (delete-bwd C' C)
note defs = this(1-6) and c'-red = this(8)
show ?thesis
    unfolding defs zl-fstate-alt-def using ZL.delete-bwd[OF c'-red] by simp
next
case (simplify-bwd C' C'' C)
note defs = this(1-6) and c''-red = this(9)
show ?thesis
    unfolding defs zl-fstate-alt-def using ZL.simplify-bwd[OF c''-red] by simp
next
case (schedule-infer  $\iota$ ss C)
note defs = this(1-6) and  $\iota$ ss = this(7)
show ?thesis
    unfolding defs zl-fstate-alt-def prod.sel option.set
    using ZL.schedule-infer[OF  $\iota$ ss, of t-llists T D passive.elems P]
    by (simp add: Un-commute)
next
case (delete-orphan-infers  $\iota$ s)
note defs = this(1-5) and  $\iota$ s-in = this(6) and inter = this(7)

show ?thesis
    unfolding defs zl-fstate-alt-def todo.llist-remove prod.sel option.set
    using ZL.delete-orphan-infers[OF inter, of t-llists T - {# $\iota$ s#} D passive.elems P
        set-option Y]
         $\iota$ s-in
    by simp
qed

```

lemma *fair-ZL-step-imp-GC-step*:

```

( $T, D, P, Y, A$ )  $\rightsquigarrow$  ZLf ( $T', D', P', Y', A'$ )  $\implies$ 
zl-fstate ( $T, D, P, Y, A$ )  $\rightsquigarrow$  LGC zl-fstate ( $T', D', P', Y', A'$ )
by (rule ZL-step-imp-LGC-step[OF fair-ZL-step-imp-ZL-step])

```

13.6 Completeness

```

fun mset-of-zl-fstate :: ('t, 'p, 'f) ZLf-state  $\Rightarrow$  'f multiset where
mset-of-zl-fstate ( $T, D, P, Y, A$ ) =
mset-set (passive.elems P) + mset-set (set-option Y) + mset-set (fset A)

```

```

abbreviation Precprec-S :: 'f multiset  $\Rightarrow$  'f multiset  $\Rightarrow$  bool (infix  $\prec\prec S$  50) where
( $\prec\prec S$ )  $\equiv$  multp ( $\prec S$ )

```

```

lemma wfP-Precprec-S: wfP ( $\prec\prec S$ )
using minimal-element-def wfP-multp wf-Prec-S wfp-on-UNIV by blast

```

```

definition Less-state :: ('t, 'p, 'f) ZLf-state  $\Rightarrow$  ('t, 'p, 'f) ZLf-state  $\Rightarrow$  bool (infix  $\sqsubset$  50)
where

```

```

St'  $\sqsubset$  St  $\iff$ 
mset-of-zl-fstate St'  $\prec\prec S$  mset-of-zl-fstate St

```

$$\begin{aligned} & \vee (mset\text{-of}\text{-}z\text{-l}\text{-fstate } St' = mset\text{-of}\text{-}z\text{-l}\text{-fstate } St \\ & \wedge (mset\text{-set } (passive.\text{elems } (passive\text{-of } St')) \prec\prec S mset\text{-set } (passive.\text{elems } (passive\text{-of } St)) \\ & \quad \vee (passive.\text{elems } (passive\text{-of } St') = passive.\text{elems } (passive\text{-of } St) \\ & \quad \wedge (mset\text{-set } (set\text{-option } (yy\text{-of } St')) \prec\prec S mset\text{-set } (set\text{-option } (yy\text{-of } St)) \\ & \quad \quad \vee (mset\text{-set } (set\text{-option } (yy\text{-of } St')) = mset\text{-set } (set\text{-option } (yy\text{-of } St)) \\ & \quad \quad \wedge size (t\text{-llists } (todo\text{-of } St')) < size (t\text{-llists } (todo\text{-of } St)))))) \end{aligned}$$

lemma *wfP-Less-state*: *wfP* (\square)

proof –

let *?msetset* = $\{(M', M). M' \prec\prec S M\}$
let *?natset* = $\{(n', n :: nat). n' < n\}$
let *?quad-of* = $\lambda St. (mset\text{-of}\text{-}z\text{-l}\text{-fstate } St, mset\text{-set } (passive.\text{elems } (passive\text{-of } St)),$
 $mset\text{-set } (set\text{-option } (yy\text{-of } St)), size (t\text{-llists } (todo\text{-of } St)))$

have *wf-msetset*: *wf* *?msetset*

using *wfP-Precprec-S wfP-def* **by** *auto*

have *wf-natset*: *wf* *?natset*

by (*rule Wellfounded.wellorder-class.wf*)

have *wf-lex-prod*: *wf* (*?msetset* $\prec\prec$ **lex** *?msetset* $\prec\prec$ **lex** *?msetset* $\prec\prec$ **lex** *?natset*)

by (*rule wf-lex-prod[OF wf-msetset wf-lex-prod[OF wf-msetset*
 $wf\text{-lex-prod}[OF wf\text{-msetset } wf\text{-natset}]]]$)

have *Less-state-alt-def*: $\bigwedge St' St. St' \square St \longleftrightarrow$

$(?quad\text{-of } St', ?quad\text{-of } St) \in ?msetset \prec\prec *lex* ?msetset \prec\prec *lex* ?msetset \prec\prec *lex* ?natset$

unfolding *Less-state-def* **by** *auto*

show *?thesis*

unfolding *wfP-def Less-state-alt-def* **using** *wf-app[of - ?quad-of]* *wf-lex-prod* **by** *blast*

qed

lemma *non-compute-infer-ZLf-step-imp-Less-state*:

assumes

step: $St \rightsquigarrow ZLf St'$ **and**

non-ci: $\neg compute\text{-infer}\text{-step } St St'$

shows $St' \square St$

using *step*

proof *cases*

case (*compute-infer T ι0 ιs A C D P*)

hence *False*

using *non-ci[unfolded compute-infer-step.simps]* **by** *blast*

thus *?thesis*

by *blast*

next

case (*choose-p P T D A*)

note *defs* = *this*(1,2)

have *all*: $add\text{-mset } (p\text{-select } P) (mset\text{-set } (passive.\text{elems } P - \{p\text{-select } P\})) =$

$mset\text{-set } (passive.\text{elems } P)$

by (*metis finite-fset local.choose-p(3) mset-set.remove passive.select-in-felems*)

have *pas*: $mset\text{-set } (passive.\text{elems } P - \{p\text{-select } P\}) \prec\prec S mset\text{-set } (passive.\text{elems } P)$

by (*metis all multi-psub-of-add-self subset-implies-multp*)

show *?thesis*

unfolding *defs Less-state-def* **by** (*simp add: all pas*)

next

```

case (delete-fwd C A T D P)
note defs = this(1,2)
show ?thesis
  unfolding defs Less-state-def by (auto intro!: subset-implies-multp)
next
case (simplify-fwd C' C A T D P)
note defs = this(1,2) and prec = this(3)

let ?new-bef = mset-set (passive.elms P) + mset-set (fset A) + {#C#}
let ?new-aft = mset-set (passive.elms P) + mset-set (fset A) + {#C'#}

have ?new-aft <<S ?new-bef
  unfolding multp-def
proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
  show {#C'#} <<S {#C#}
    unfolding multp-def using prec by (auto intro: singletons-in-mult)
qed
thus ?thesis
  unfolding defs Less-state-def by simp
next
case (delete-bwd C' A C T D P)
note defs = this(1,2) and c-ni = this(3)
show ?thesis
  unfolding defs Less-state-def using c-ni
  by (auto intro!: subset-implies-multp)
next
case (simplify-bwd C' A C'' C T D P)
note defs = this(1,2) and c'-ni = this(3) and prec = this(4)

show ?thesis
proof (cases C'' ∈ passive.elms P)
  case c''-in: True
  show ?thesis
    unfolding defs Less-state-def using c'-ni
    by (auto simp: insert-absorb[OF c''-in] intro!: subset-implies-multp)
next
case c''-ni: False

have bef: add-mset C (mset-set (passive.elms P) + mset-set (insert C' (fset A))) =
  add-mset C (mset-set (passive.elms P) + mset-set (fset A)) + {#C'#}
  (is ?old-bef = ?new-bef)
  using c'-ni by auto
have aft: add-mset C (mset-set (insert C'' (passive.elms P)) + mset-set (fset A)) =
  add-mset C (mset-set (passive.elms P) + mset-set (fset A)) + {#C''#}
  (is ?old-aft = ?new-aft)
  using c''-ni by simp

have ?new-aft <<S ?new-bef
  unfolding multp-def
proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
  show {#C''#} <<S {#C'#}
    unfolding multp-def using prec by (auto intro: singletons-in-mult)
qed
thus ?thesis
  unfolding defs Less-state-def by (simp add: bef aft)

```

```

qed
next
case (schedule-infer  $\iota s$  A C T D P)
note defs = this(1,2)
show ?thesis
  unfolding defs Less-state-def
  by simp (metis finite-fset insert-absorb mset-set.insert multi-psub-of-add-self
    subset-implies-multip)
next
case (delete-orphan-infers  $\iota s$  T A D P Y)
note defs = this(1,2) and  $\iota s$  = this(3)
have size (t-llists T - {# $\iota s$ #}) < size (t-llists T)
  using  $\iota s$  by (simp add: size-Diff1-less)
thus ?thesis
  unfolding defs Less-state-def by simp
qed

```

```

lemma yy-nonempty-ZLf-step-imp-Less-state:
  assumes
    step:  $St \rightsquigarrow ZLf St'$  and
    yy:  $yy\text{-of } St \neq None$ 
  shows  $St' \sqsubset St$ 
proof -
  have  $\neg$  compute-infer-step  $St St'$ 
    using yy unfolding compute-infer-step.simps by auto
  thus ?thesis
    using non-compute-infer-ZLf-step-imp-Less-state[OF step] by blast
qed

```

```

lemma fair-ZL-Liminf-yy-empty:
  assumes
    len:  $length\ Sts = \infty$  and
    full: full-chain ( $\rightsquigarrow ZLf$ )  $Sts$  and
    inv: ZLf-invariant (lhd  $Sts$ )
  shows  $Liminf\text{-llist } (lmap (set\ option \circ yy\text{-of})\ Sts) = \{\}$ 
proof (rule ccontr)
  assume lim-nemp:  $Liminf\text{-llist } (lmap (set\ option \circ yy\text{-of})\ Sts) \neq \{\}$ 

```

```

  obtain  $i :: nat$  where
    i-lt:  $enat\ i < length\ Sts$  and
    inter-nemp:  $\bigcap ((set\ option \circ yy\text{-of} \circ lnth\ Sts) \text{ ' } \{j. i \leq j \wedge enat\ j < length\ Sts\}) \neq \{\}$ 
    using lim-nemp unfolding Liminf-llist-def by auto

```

```

  from inter-nemp obtain  $C :: 'f$  where
    c-in:  $\forall P \in lnth\ Sts \text{ ' } \{j. i \leq j \wedge enat\ j < length\ Sts\}. C \in set\ option (yy\text{-of } P)$ 
    by auto
  hence c-in':  $\forall j \geq i. enat\ j < length\ Sts \longrightarrow C \in set\ option (yy\text{-of } (lnth\ Sts\ j))$ 
    by auto

```

```

  have si-lt:  $enat (Suc\ i) < length\ Sts$ 
    unfolding len by auto

```

```

  have yy-j:  $yy\text{-of } (lnth\ Sts\ j) \neq None$  if  $j\text{-ge}: j \geq i$  for  $j$ 
    using c-in' len  $j\text{-ge}$  by auto
  have step:  $lnth\ Sts\ j \rightsquigarrow ZLf\ lnth\ Sts (Suc\ j)$  if  $j\text{-ge}: j \geq i$  for  $j$ 

```

```

using full-chain-imp-chain[OF full] infinite-chain-lnth-rel len llength-eq-infty-conv-lfinite
by blast

have lnth Sts (Suc j)  $\sqsubseteq$  lnth Sts j if j-ge:  $j \geq i$  for j
  using yy-nonempty-ZLf-step-imp-Less-state by (meson step j-ge yy-j)
hence  $(\sqsubseteq)^{-1-1}$  (lnth Sts j) (lnth Sts (Suc j))
  if j-ge:  $j \geq i$  for j
  using j-ge by blast
hence inf-down-chain: chain  $(\sqsubseteq)^{-1-1}$  (ldropn i Sts)
  by (simp add: chain-ldropnI si-lt)

have inf-i:  $\neg$  lfinite (ldropn i Sts)
  using len by (simp add: llength-eq-infty-conv-lfinite)

show False
  using inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of  $(\sqsubseteq)$ ] wfP-Less-state
  by metis
qed

lemma ZLf-step-imp-passive-queue-step:
  assumes  $St \rightsquigarrow_{ZLf} St'$ 
  shows passive.queue-step (passive-of St) (passive-of St')
  using assms
  by cases (auto intro: passive.queue-step-idleI passive.queue-step-addI
    passive.queue-step-removeI)

lemma choose-p-step-imp-select-passive-queue-step:
  assumes choose-p-step St St'
  shows passive.select-queue-step (passive-of St) (passive-of St')
  using assms
proof cases
  case (1 P T D A)
  note defs = this(1,2) and p-nemp = this(3)
  show ?thesis
  unfolding defs prod.sel by (rule passive.select-queue-stepI[OF p-nemp])
qed

lemma fair-ZL-Liminf-passive-empty:
  assumes
    len: llength Sts =  $\infty$  and
    full: full-chain ( $\rightsquigarrow_{ZLf}$ ) Sts and
    init: is-initial-ZLf-state (lhd Sts) and
    fair: infinitely-often compute-infer-step Sts  $\longrightarrow$  infinitely-often choose-p-step Sts
  shows Liminf-llist (lmap (passive.elms  $\circ$  passive-of) Sts) = {}
proof -
  have chain-step: chain passive.queue-step (lmap passive-of Sts)
  using ZLf-step-imp-passive-queue-step chain-lmap full-chain-imp-chain[OF full]
  by (metis (no-types, lifting))

  have inf-oft: infinitely-often passive.select-queue-step (lmap passive-of Sts)
proof
  assume finitely-often passive.select-queue-step (lmap passive-of Sts)
  hence fin-cp: finitely-often choose-p-step Sts
  unfolding finitely-often-def choose-p-step-imp-select-passive-queue-step
  by (smt choose-p-step-imp-select-passive-queue-step enat-ord-code(4) len llength-lmap

```

```

    lnth-lmap)
  hence fin-ci: finitely-often compute-infer-step Sts
    using fair by blast

  obtain i :: nat where
    i:  $\forall j \geq i. \neg \text{compute-infer-step } (\text{lnth Sts } j) (\text{lnth Sts } (\text{Suc } j))$ 
    using fin-ci len unfolding finitely-often-def by auto

  have si-lt: enat (Suc i) < llength Sts
    unfolding len by auto

  have not-ci:  $\neg \text{compute-infer-step } (\text{lnth Sts } j) (\text{lnth Sts } (\text{Suc } j))$  if j-ge:  $j \geq i$  for j
    using i j-ge by auto

  have step:  $\text{lnth Sts } j \rightsquigarrow \text{ZLf lnth Sts } (\text{Suc } j)$  if j-ge:  $j \geq i$  for j
    by (simp add: full-chain-lnth-rel[OF full] len)

  have lnth Sts (Suc j)  $\sqsubset$  lnth Sts j if j-ge:  $j \geq i$  for j
    by (rule non-compute-infer-ZLf-step-imp-Less-state[OF step[OF j-ge] not-ci[OF j-ge]])
  hence  $(\sqsubset)^{-1-1} (\text{lnth Sts } j) (\text{lnth Sts } (\text{Suc } j))$  if j-ge:  $j \geq i$  for j
    using j-ge by blast
  hence inf-down-chain: chain  $(\sqsubset)^{-1-1} (\text{ldropn } i \text{ Sts})$ 
    using chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id] by simp
  have inf-i:  $\neg \text{lfinite } (\text{ldropn } i \text{ Sts})$ 
    using len lfinite-ldropn llength-eq-infty-conv-lfinite by blast
  show False
    using inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of  $(\sqsubset)$ ] wfP-Less-state
    by blast
qed

  have hd-emp:  $\text{lhd } (\text{lmap passive-of Sts}) = p\text{-empty}$ 
    using init full full-chain-not-lnull unfolding is-initial-ZLf-state.simps by fastforce

  have Liminf-llist  $(\text{lmap passive.elems } (\text{lmap passive-of Sts})) = \{\}$ 
    by (rule passive.fair[OF chain-step inf-oft hd-emp])
  thus ?thesis
    by (simp add: llist.map-comp)
qed

lemma ZLf-step-imp-todo-queue-step:
  assumes  $St \rightsquigarrow \text{ZLf } St'$ 
  shows  $\text{todo.lqueue-step } (\text{todo-of } St, \text{done-of } St) (\text{todo-of } St', \text{done-of } St')$ 
  using assms
proof cases
  case (compute-infer T  $\iota 0$  T' A C D P)
  note defs = this(1,2) and has-el = this(3) and pick = this(4)
  have t':  $T' = \text{snd } (t\text{-pick-elem } T)$ 
    using pick by simp
  show ?thesis
    unfolding defs prod.sel t' using todo.lqueue-step-pick-elemI[OF has-el] by (simp add: pick)
next
  case (schedule-infer  $\iota ss$  A C T D P)
  note defs = this(1,2) and betw = this(3)
  show ?thesis
    unfolding defs prod.sel using todo.lqueue-step-fold-add-llistI[of T D  $\iota ss$ ] by simp

```


qed (*auto intro: todo.lqueue-step-idleI todo.lqueue-step-fold-add-llistI
todo.lqueue-step-remove-llistI*)

lemma *fair-ZL-Liminf-todo-empty:*

assumes

len: llength Sts = ∞ and

full: full-chain (↪ZLf) Sts and

init: is-initial-ZLf-state (lhd Sts)

shows *Liminf-llist (lmap (λSt. flat-inferences-of (t-llists (todo-of St)) – done-of St) Sts) = {}*

proof –

define *Infs* **where**

Infs = lmap (λSt. flat-inferences-of (t-llists (todo-of St)) – done-of St) Sts

define *flat-Ts* **where**

flat-Ts = lmap (λSt. flat-inferences-of (t-llists (todo-of St))) Sts

define *TDs* **where**

TDs = lmap (λSt. (todo-of St, done-of St)) Sts

{

fix *i ι*

assume *ι-in-infs: ι ∈ lnth Infs i*

have *lt-sts: enat n < llength Sts for n*

by (*simp add: len*)

have *lt-tds: enat n < llength TDs for n*

by (*simp add: TDs-def len*)

have *chain-ts: chain todo.lqueue-step TDs*

proof –

have *fst-tds: lmap fst TDs = lmap todo-of Sts*

unfolding *TDs-def* **by** (*simp add: llist.map-comp*)

have *snd-tds: lmap snd TDs = lmap done-of Sts*

unfolding *TDs-def* **by** (*simp add: llist.map-comp*)

show *?thesis*

unfolding *fst-tds*

using *TDs-def ZLf-step-imp-todo-queue-step chain-lmap full full-chain-imp-chain*

by (*metis (lifting)*)

qed

have *inf-oft: infinitely-often todo.pick-lqueue-step TDs*

proof

assume *finitely-often todo.pick-lqueue-step TDs*

then obtain *i :: nat* **where**

no-pick: ∀ j ≥ i. ¬ todo.pick-lqueue-step (lnth TDs j) (lnth TDs (Suc j))

by (*metis infinitely-often-alt-def lt-tds*)

have *si-lt: enat (Suc i) < llength Sts*

unfolding *len* **by** *auto*

have *step: lnth Sts j ↪ZLf lnth Sts (Suc j) if j-ge: j ≥ i for j*

using *full-chain-imp-chain[OF full] infinite-chain-lnth-rel len*

length-eq-infnty-conv-lfinite

by *blast*

have *non-ci: ¬ compute-infer-step (lnth Sts j) (lnth Sts (Suc j)) if j-ge: j ≥ i for j*

```

proof –
{
  assume compute-infer-step (lnth Sts j) (lnth Sts (Suc j))
  hence  $\exists j \geq i$ . todo.pick-lqueue-step (lnth TDs j) (lnth TDs (Suc j))
    using assms
  proof cases
  case (1 T  $\iota 0$  T' A C D P)
  note sts-at-j = this(1) and sts-at-sj = this(2) and has-el = this(3) and pick = this(4)

  obtain  $\iota 0' :: 'f$  inference and  $\iota s :: 'f$  inference llist where
    cons-in0: LCons  $\iota 0' \iota s \in \#$  t-llists T and
    fst0: fst (t-pick-elem T) =  $\iota 0'$  and
    snd0: t-llists (snd (t-pick-elem T)) = t-llists T -  $\{\#LCons \iota 0' \iota s\# \}$  +  $\{\#\iota s\# \}$ 
    using todo.llists-pick-elem[OF has-el] by blast

  have  $\iota 0' : \iota 0' = \iota 0$ 
    using pick fst0 by auto

  have
    cons-in: LCons  $\iota 0 \iota s \in \#$  t-llists T and
    fst: fst (t-pick-elem T) =  $\iota 0$  and
    snd: t-llists (snd (t-pick-elem T)) = t-llists T -  $\{\#LCons \iota 0 \iota s\# \}$  +  $\{\#\iota s\# \}$ 
    unfolding  $\iota 0'$ [symmetric] by (auto simp: cons-in0 fst0 snd0)

  have td-at-j: lnth TDs j = (T, D)
    using sts-at-j TDs-def lt-tds by auto
  have td-at-sj: lnth TDs (Suc j) = (snd (t-pick-elem T), insert  $\iota 0$  D)
    using sts-at-sj TDs-def lt-tds pick by force

  have todo.pick-lqueue-step (lnth TDs j) (lnth TDs (Suc j))
    by (simp add: todo.pick-lqueue-step.simps todo.pick-lqueue-step-w-details.simps,
      rule exI[of -  $\iota s$ ], rule exI[of - T], rule exI[of - D],
      simp add: td-at-j td-at-sj cons-in fst snd)
  thus ?thesis
    using j-ge by blast
  qed
}
thus ?thesis
  using no-pick by blast
qed

have lnth Sts (Suc j)  $\sqsubset$  lnth Sts j if j-ge:  $j \geq i$  for j
  by (rule non-compute-infer-ZLf-step-imp-Less-state[OF step[OF j-ge] non-ci[OF j-ge]])
hence  $(\sqsubset)^{-1-1}$  (lnth Sts j) (lnth Sts (Suc j)) if j-ge:  $j \geq i$  for j
  using j-ge by blast
hence inf-down-chain: chain  $(\sqsubset)^{-1-1}$  (ldropn i Sts)
  using chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id] by simp

have inf-i:  $\neg$  lfinite (ldropn i Sts)
  using len lfinite-ldropn llength-eq-infnty-conv-lfinite by blast

show False
  using inf-i inf-down-chain wfP-iff-no-infinite-down-chain-llist[of  $(\sqsubset)$ ] wfP-Less-state
  by blast
qed

```

```

have  $\iota \in \text{lnth flat-Ts } i$ 
  using  $\iota\text{-in-infs unfolding Infs-def flat-Ts-def}$  by (simp add: lt-sts)
then obtain  $\iota s :: 'f \text{ inference llist where}$ 
   $\iota s\text{-in: } \iota s \in \# \text{ t-llists (fst (lnth TDs } i))$  and
   $\iota\text{-in-}\iota s: \iota \in \text{lset } \iota s$ 
  using  $\text{lnth-lmap}[OF \text{ lt-sts}]$  unfolding  $\text{flat-Ts-def TDs-def}$ 
  by (smt (verit, ccfv-SIG) Union-iff flat-inferences-of.simps fst-conv mem-Collect-eq)

obtain  $k :: \text{nat where}$ 
   $k\text{-lt: } \text{enat } k < \text{llength } \iota s$  and
   $at\text{-k: } \text{lnth } \iota s \ k = \iota$ 
  using  $\iota\text{-in-}\iota s$  by (meson in-lset-conv-lnth)

obtain  $j :: \text{nat where}$ 
   $j\text{-ge: } j \geq i$  and
   $\text{rem-or-pick-step: } (\exists k' \leq k. \exists \iota ss.$ 
     $\text{ldrop (enat } k') \ \iota s \in \text{set } \iota ss \wedge \text{todo.remove-lqueue-step-w-details (lnth TDs } j) \ \iota ss$ 
     $(\text{lnth TDs (Suc } j)))$ 
     $\vee \text{todo.pick-lqueue-step-w-details (lnth TDs } j) (\text{lnth } \iota s \ k) (\text{ldrop (enat (Suc } k)) \ \iota s)$ 
     $(\text{lnth TDs (Suc } j)))$ 
  using  $\text{todo.fair-strong}[OF \text{ chain-ts inf-oft } \iota s\text{-in } k\text{-lt}]$  by blast

have  $\exists j. j \geq i \wedge j < \text{llength Sts} \wedge \iota \notin \text{lnth Infs } j$ 
proof (rule exI[of - Suc j], intro conjI)
  {
    assume  $\exists k' \leq k. \exists \iota ss. \text{ldrop (enat } k') \ \iota s \in \text{set } \iota ss$ 
       $\wedge \text{todo.remove-lqueue-step-w-details (lnth TDs } j) \ \iota ss (\text{lnth TDs (Suc } j))$ 
    then obtain  $k' :: \text{nat and } \iota ss :: 'f \text{ inference llist list where}$ 
       $k'\text{-le: } k' \leq k$  and
       $\text{in-}\iota ss: \text{ldrop (enat } k') \ \iota s \in \text{set } \iota ss$  and
       $\text{rem-step: } \text{todo.remove-lqueue-step-w-details (lnth TDs } j) \ \iota ss (\text{lnth TDs (Suc } j))$ 
      by blast

    have  $\iota \notin \text{lnth Infs (Suc } j)$ 
      using rem-step
    proof cases
      case (remove-lqueue-step-w-detailsI Q D)
        note  $at\text{-j} = \text{this}(1)$  and  $at\text{-sj} = \text{this}(2)$ 

        have  $\text{don: } \text{done-of (lnth Sts (Suc } j)) = D \cup \bigcup \{\text{lset } \iota s \mid \iota s. \iota s \in \text{set } \iota ss\}$ 
          unfolding  $at\text{-sj}$  using  $\text{TDs-def at-sj len}$  by auto

        have  $\iota \in \text{lset (ldrop (enat } k') \ \iota s)$ 
          proof –
            have  $\text{nth-drop: } \text{lnth (ldrop (enat } k') \ \iota s) (k - k') = \iota$ 
              by (simp add: at-k k'-le k-lt)
            thus ?thesis
            using  $at\text{-k } k'\text{-le } k\text{-lt}$  by (smt (verit, del-insts) enat.distinct(1)
              enat-diff-cancel-left enat-minus-mono1 enat-ord-simps(1) idiff-enat-enat
              in-lset-conv-lnth llength-ldrop nless-le order-le-less-subst2)
          qed
        hence  $\iota \in \bigcup \{\text{lset } \iota s \mid \iota s. \iota s \in \text{set } \iota ss\}$ 
          using in-}\iota ss by blast
        thus ?thesis
  }

```

```

      unfolding Infs-def lnth-lmap[OF lt-sts] don by auto
    qed
  }
  moreover
  {
    assume todo.pick-lqueue-step-w-details (lnth TDs j) (lnth  $\iota$  s k) (ldrop (enat (Suc k))  $\iota$  s)
      (lnth TDs (Suc j))
    hence  $\iota \notin \text{lnth Infs (Suc j)}$ 
    proof cases
      case (pick-lqueue-step-w-detailsI Q D)
        note at-j = this(1) and at-sj = this(2)

        have don: done-of (lnth Sts (Suc j)) =  $D \cup \{\iota\}$ 
          using at-sj at-k by (simp add: TDs-def len)

        show ?thesis
          unfolding Infs-def lnth-lmap[OF lt-sts] don by auto
        qed
      }
    ultimately show  $\iota \notin \text{lnth Infs (Suc j)}$ 
      using rem-or-pick-step by blast
    qed (use j-ge lt-sts in auto)
  }
}
thus ?thesis
  unfolding Infs-def[symmetric] Liminf-llist-def
  by clarsimp (smt Infs-def Collect-empty-eq INT-iff Inf-set-def dual-order.refl llength-lmap
    mem-Collect-eq)
qed

```

theorem

assumes

full: full-chain (\rightsquigarrow ZLf) Sts **and**

init: is-initial-ZLf-state (lhd Sts) **and**

fair: infinitely-often compute-infer-step Sts \longrightarrow infinitely-often choose-p-step Sts

shows

fair-ZL-Liminf-saturated: saturated (labeled-formulas-of (Liminf-zl-fstate Sts)) **and**

fair-ZL-complete-Liminf: $B \in \text{Bot-F} \implies \text{passive elems (passive-of (lhd Sts))} \models_{\cap \mathcal{G}} \{B\} \implies$

$\exists B' \in \text{Bot-F}. B' \in \text{formulas-union (Liminf-zl-fstate Sts)}$ **and**

fair-ZL-complete: $B \in \text{Bot-F} \implies \text{passive elems (passive-of (lhd Sts))} \models_{\cap \mathcal{G}} \{B\} \implies$

$\exists i. \text{enat } i < \text{llength Sts} \wedge (\exists B' \in \text{Bot-F}. B' \in \text{all-formulas-of (lnth Sts } i))$

proof –

have chain: chain (\rightsquigarrow ZLf) Sts

by (rule full-chain-imp-chain[OF full])

have zl-chain: chain (\rightsquigarrow ZL) (lmap zl-fstate Sts)

using chain fair-ZL-step-imp-ZL-step chain-lmap **by** (smt (verit) zl-fstate.cases)

have inv: ZLf-invariant (lhd Sts)

using init initial-ZLf-invariant **by** auto

have nnul: $\neg \text{lnull Sts}$

using chain chain-not-lnull **by** blast

hence lhd-lmap: $\bigwedge f. \text{lhd (lmap } f \text{ Sts)} = f \text{ (lhd Sts)}$

by (rule llist.map-sel(1))

have active-of (lhd Sts) = $\{\|\}$

by (*metis is-initial-ZLf-state.cases init snd-conv*)
hence *act*: *active-subset* (*snd* (*lhd* (*lmap* *zl-fstate* *Sts*))) = {}
unfolding *active-subset-def lhd-lmap* **by** (*cases lhd Sts*) *auto*

have *pas-fml-and-t-inf*: *passive-subset* (*Liminf-llist* (*lmap* (*snd* \circ *zl-fstate*) *Sts*)) = {} \wedge
Liminf-llist (*lmap* (*fst* \circ *zl-fstate*) *Sts*) = {} (**is** *?pas-fml* \wedge *?t-inf*)
proof (*cases lfinite Sts*)
case *fin*: *True*

have *lim-fst*: *Liminf-llist* (*lmap* (*fst* \circ *zl-fstate*) *Sts*) = *fst* (*zl-fstate* (*llast* *Sts*)) **and**
lim-snd: *Liminf-llist* (*lmap* (*snd* \circ *zl-fstate*) *Sts*) = *snd* (*zl-fstate* (*llast* *Sts*))
using *lfinite-Liminf-llist fin nnul*
by (*metis comp-eq-dest-lhs lfinite-lmap llast-lmap llist.map-disc-iff*) $+$

have *last-inv*: *ZLf-invariant* (*llast* *Sts*)
by (*rule chain-ZLf-invariant-llast*[*OF chain inv fin*])

have $\forall St'. \neg llast Sts \rightsquigarrow ZLf St'$
using *full-chain-lnth-not-rel*[*OF full*] **by** (*metis fin full-chain-iff-chain full*)
hence *is-final-ZLf-state* (*llast* *Sts*)
unfolding *is-final-ZLf-state-iff-no-ZLf-step*[*OF last-inv*] .
then obtain *D* :: '*f* inference set **and** *A* :: '*f* fset **where**
at-l: *llast* *Sts* = (*t-empty*, *D*, *p-empty*, *None*, *A*)
unfolding *is-final-ZLf-state.simps* **by** *blast*

have *?pas-fml*
unfolding *passive-subset-def lim-snd at-l* **by** *auto*
moreover have *?t-inf*
unfolding *lim-fst at-l* **by** *simp*
ultimately show *?thesis*
by *blast*

next
case *False*
hence *len*: *llength* *Sts* = ∞
by (*simp add: not-lfinite-llength*)

have *?pas-fml*
unfolding *Liminf-zl-fstate-commute passive-subset-def Liminf-zl-fstate-def*
using *fair-ZL-Liminf-passive-empty*[*OF len full init fair*]
fair-ZL-Liminf-yy-empty[*OF len full inv*]
by *simp*
moreover have *?t-inf*
unfolding *zl-fstate-alt-def comp-def zl-state.simps prod.sel*
using *fair-ZL-Liminf-todo-empty*[*OF len full init*] .
ultimately show *?thesis*
by *blast*

qed
note *pas-fml* = *pas-fml-and-t-inf*[*THEN conjunct1*] **and**
t-inf = *pas-fml-and-t-inf*[*THEN conjunct2*]

obtain *iss* :: '*f* inference llist list **where**
hd: *lhd* *Sts* = (*fold t-add-llist iss t-empty*, {}, *p-empty*, *None*, {||}) **and**
infs: *flat-inferences-of* (*mset iss*) = { $\iota \in Inf-F$. *prems-of* ι = []}
using *init*[*unfolded is-initial-ZLf-state.simps no-labels.Inf-from-empty*] **by** *blast*

```

have hd': lhd (lmap zl-fstate Sts) =
  zl-fstate (fold t-add-llist iss t-empty, {}, p-empty, None, {||})
  using hd by (simp add: lhd-lmap)

have no-prems-init:  $\forall \iota \in \text{Inf-}F. \text{prems-of } \iota = [] \longrightarrow \iota \in \text{fst} (\text{lhd} (\text{lmap } \text{zl-fstate } \text{Sts}))$ 
  unfolding zl-fstate-alt-def hd' zl-state-alt-def prod.sel using infs by simp

show saturated (labeled-formulas-of (Liminf-zl-fstate Sts))
  using ZL-Liminf-saturated[of lmap zl-fstate Sts, unfolded llist.map-comp,
    OF zl-chain act pas-fml no-prems-init t-inf]
  unfolding Liminf-zl-fstate-commute .

{
assume
  bot:  $B \in \text{Bot-}F$  and
  unsat: passive elems (passive-of (lhd Sts))  $\models \cap \mathcal{G} \{B\}$ 

have unsat':  $\text{fst } \text{'snd} (\text{lhd} (\text{lmap } \text{zl-fstate } \text{Sts})) \models \cap \mathcal{G} \{B\}$ 
  using unsat unfolding lhd-lmap by (cases lhd Sts) (auto intro: no-labels-entails-mono-left)

have  $\exists BL \in \text{Bot-}FL. BL \in \text{Liminf-llist} (\text{lmap} (\text{snd} \circ \text{zl-fstate}) \text{Sts})$ 
  using ZL-complete-Liminf[of lmap zl-fstate Sts, unfolded llist.map-comp,
    OF zl-chain act pas-fml no-prems-init t-inf bot unsat'] .
thus  $\exists B' \in \text{Bot-}F. B' \in \text{formulas-union} (\text{Liminf-zl-fstate } \text{Sts})$ 
  unfolding Liminf-zl-fstate-def Liminf-zl-fstate-commute by auto
thus  $\exists i. \text{enat } i < \text{llength } \text{Sts} \wedge (\exists B' \in \text{Bot-}F. B' \in \text{all-formulas-of} (\text{lth } \text{Sts } i))$ 
  unfolding Liminf-zl-fstate-def Liminf-llist-def by auto
}
qed

end

end

```

14 Fair Zipperposition Loop without Ghosts

This version of the fair Zipperposition loop eliminates the ghost state component D , thus confirming that D is indeed a ghost.

```

theory Fair-Zipperposition-Loop-without-Ghosts
  imports Fair-Zipperposition-Loop
begin

```

14.1 Locale

```

type-synonym ('t, 'p, 'f) ZLf-wo-ghosts-state = 't  $\times$  'p  $\times$  'f option  $\times$  'f fset

```

```

locale fair-zipperposition-loop-wo-ghosts =
  w-ghosts?: fair-zipperposition-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q
  G-I-q Equiv-F Prec-F t-empty t-add-llist t-remove-llist t-pick-elem t-llists p-empty p-select
  p-add p-remove p-felems Prec-S
for
  Bot-F :: 'f set and
  Inf-F :: 'f inference set and
  Bot-G :: 'g set and

```

Q :: 'q set **and**
entails-q :: 'q ⇒ 'g set ⇒ 'g set ⇒ bool **and**
Inf-G-q :: 'q ⇒ 'g inference set **and**
Red-I-q :: 'q ⇒ 'g set ⇒ 'g inference set **and**
Red-F-q :: 'q ⇒ 'g set ⇒ 'g set **and**
G-F-q :: 'q ⇒ 'f ⇒ 'g set **and**
G-I-q :: 'q ⇒ 'f inference ⇒ 'g inference set option **and**
Equiv-F :: 'f ⇒ 'f ⇒ bool (**infix** ≐ 50) **and**
Prec-F :: 'f ⇒ 'f ⇒ bool (**infix** <· 50) **and**
t-empty :: 't **and**
t-add-llist :: 'f inference llist ⇒ 't ⇒ 't **and**
t-remove-llist :: 'f inference llist ⇒ 't ⇒ 't **and**
t-pick-elem :: 't ⇒ 'f inference × 't **and**
t-llists :: 't ⇒ 'f inference llist multiset **and**
p-empty :: 'p **and**
p-select :: 'p ⇒ 'f **and**
p-add :: 'f ⇒ 'p ⇒ 'p **and**
p-remove :: 'f ⇒ 'p ⇒ 'p **and**
p-felems :: 'p ⇒ 'f fset **and**
Prec-S :: 'f ⇒ 'f ⇒ bool (**infix** <S 50)

begin

fun *wo-ghosts-of* :: ('t, 'p, 'f) ZLf-state ⇒ ('t, 'p, 'f) ZLf-wo-ghosts-state **where**
wo-ghosts-of (T, D, P, Y, A) = (T, P, Y, A)

inductive

fair-ZL-wo-ghosts ::
('t, 'p, 'f) ZLf-wo-ghosts-state ⇒ ('t, 'p, 'f) ZLf-wo-ghosts-state ⇒ bool
(**infix** ~ ZLfw 50)
where
compute-infer: (∃ *ιs* ∈# *t-llists* T. *ιs* ≠ LNil) ⇒ *t-pick-elem* T = (*ι0*, T') ⇒
ι0 ∈ *no-labels.Red-I* (fset A ∪ {C}) ⇒
(T, P, None, A) ~ ZLfw (T', *p-add* C P, None, A)
| *choose-p*: P ≠ *p-empty* ⇒
(T, P, None, A) ~ ZLfw (T, *p-remove* (*p-select* P) P, *Some* (*p-select* P), A)
| *delete-fwd*: C ∈ *no-labels.Red-F* (fset A) ∨ (∃ C' ∈ fset A. C' ≼· C) ⇒
(T, P, *Some* C, A) ~ ZLfw (T, P, None, A)
| *simplify-fwd*: C' <S C ⇒ C ∈ *no-labels.Red-F* (fset A ∪ {C'}) ⇒
(T, P, *Some* C, A) ~ ZLfw (T, P, *Some* C', A)
| *delete-bwd*: C' |∉| A ⇒ C' ∈ *no-labels.Red-F* {C} ∨ C' ·> C ⇒
(T, P, *Some* C, A |∪| {|C'|}) ~ ZLfw (T, P, *Some* C, A)
| *simplify-bwd*: C' |∉| A ⇒ C'' <S C' ⇒ C' ∈ *no-labels.Red-F* {C, C''} ⇒
(T, P, *Some* C, A |∪| {|C'|}) ~ ZLfw (T, *p-add* C'' P, *Some* C, A)
| *schedule-infer*: *flat-inferences-of* (mset *ιss*) = *no-labels.Inf-between* (fset A) {C} ⇒
(T, P, *Some* C, A) ~ ZLfw (fold *t-add-llist* *ιss* T, P, None, A |∪| {|C|})
| *delete-orphan-infers*: *ιs* ∈# *t-llists* T ⇒ *lset* *ιs* ∩ *no-labels.Inf-from* (fset A) = {} ⇒
(T, P, Y, A) ~ ZLfw (*t-remove-llist* *ιs* T, P, Y, A)

inductive

compute-infer-step ::
('t, 'p, 'f) ZLf-wo-ghosts-state ⇒ ('t, 'p, 'f) ZLf-wo-ghosts-state ⇒ bool
where
(∃ *ιs* ∈# *t-llists* T. *ιs* ≠ LNil) ⇒ *t-pick-elem* T = (*ι0*, T') ⇒
ι0 ∈ *no-labels.Red-I* (fset A ∪ {C}) ⇒
compute-infer-step (T, P, None, A) (T', *p-add* C P, None, A)

inductive

choose-p-step :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow bool

where

$P \neq p\text{-empty} \implies$

choose-p-step (T, P, None, A) (T, p-remove (p-select P) P, Some (p-select P), A)

lemma *w-ghosts-compute-infer-step-imp-compute-infer-step*:

assumes *w-ghosts.compute-infer-step* St St'

shows *compute-infer-step* (wo-ghosts-of St) (wo-ghosts-of St')

using *assms* **by** *cases* (*simp* *add*: *compute-infer-step.intros*)

lemma *choose-p-step-imp-w-ghosts-choose-p-step*:

assumes *choose-p-step* (wo-ghosts-of St) (wo-ghosts-of St')

shows *w-ghosts.choose-p-step* St St'

using *assms*

proof *cases*

case (1 P T A)

note *wg-st* = *this*(1) **and** *wg-st'* = *this*(2) **and** *rest* = *this*(3)

have *st*: St = (T, done-of St, P, None, A)

using *wg-st* **by** (*smt* (*verit*) *fst-conv snd-conv wo-ghosts-of.elims*)

have *st'*: St' = (T, done-of St', p-remove (p-select P) P, Some (p-select P), A)

using *wg-st'* **by** (*smt* (*verit*) *fst-conv snd-conv wo-ghosts-of.elims*)

show *?thesis*

by (*subst st*, *subst st'*, *simp* *add*: *rest w-ghosts.choose-p-step.intros*)

qed

14.2 Basic Definitions and Lemmas

abbreviation *todo-of* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow 't **where**

todo-of St \equiv *fst* St

abbreviation *passive-of* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow 'p **where**

passive-of St \equiv *fst* (*snd* St)

abbreviation *yy-of* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow 'f option **where**

yy-of St \equiv *fst* (*snd* (*snd* St))

abbreviation *active-of* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow 'f fset **where**

active-of St \equiv *snd* (*snd* (*snd* St))

abbreviation *all-formulas-of* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow 'f set **where**

all-formulas-of St \equiv *passive.elims* (*passive-of* St) \cup *set-option* (*yy-of* St) \cup *fset* (*active-of* St)

definition

Liminf-zl-fstate :: ('t, 'p, 'f) ZLf-wo-ghosts-state llist \Rightarrow 'f set \times 'f set \times 'f set

where

Liminf-zl-fstate Sts =

(*Liminf-llist* (*lmap* (*passive.elims* \circ *passive-of*) Sts),

Liminf-llist (*lmap* (*set-option* \circ *yy-of*) Sts),

Liminf-llist (*lmap* (*fset* \circ *active-of*) Sts))

14.3 Initial States and Invariants

inductive *is-initial-ZLf-wo-ghosts-state* :: ('t, 'p, 'f) ZLf-wo-ghosts-state \Rightarrow bool **where**

flat-inferences-of (*mset* *lss*) = *no-labels.Inf-from* {} \implies

is-initial-ZLf-wo-ghosts-state (*fold* *t-add-llist* *lss* *t-empty*, *p-empty*, None, {||})

lemma *is-initial-ZLf-state-imp-is-initial-ZLf-wo-ghosts-state*:
assumes *is-initial-ZLf-state St*
shows *is-initial-ZLf-wo-ghosts-state (wo-ghosts-of St)*
using *assms by cases (auto intro: is-initial-ZLf-wo-ghosts-state.intros)*

lemma *is-initial-ZLf-wo-ghosts-state-imp-is-initial-ZLf-state*:
assumes
init: is-initial-ZLf-wo-ghosts-state (wo-ghosts-of St) and
don: done-of St = {}
shows *is-initial-ZLf-state St*
using *init*
by *cases (smt don is-initial-ZLf-state.simps prod.inject prod.exhaust-sel wo-ghosts-of.elims)*

end

14.4 Abstract Nonsense for Ghost–Ghostless Conversion

This subsection was originally contributed by Andrei Popescu.

locale *bisim* =
fixes *erase :: 'state0 ⇒ 'state*
and *R :: 'state ⇒ 'state ⇒ bool (infix ~ 60)*
and *R0 :: 'state0 ⇒ 'state0 ⇒ bool (infix ~ 0 60)*
assumes *simul: ∧ St0 St'. erase St0 ~ St' ⇒ ∃ St0'. erase St0' = St' ∧ St0 ~ 0 St0'*
begin

definition *lift :: 'state0 ⇒ 'state ⇒ 'state0 where*
lift St0 St' = (SOME St0'. erase St0' = St' ∧ St0 ~ 0 St0')

lemma *lift: erase St0 ~ St' ⇒ erase (lift St0 St') = St' ∧ St0 ~ 0 lift St0 St'*
by *(smt (verit) lift-def simul someI)*

lemmas *erase-lift = lift[THEN conjunct1]*
lemmas *R0-lift = lift[THEN conjunct2]*

primcorec *theSts0 :: 'state0 ⇒ 'state llist ⇒ 'state0 llist where*
theSts0 St0 Sts =
(case Sts of
LNil ⇒ LCons St0 LNil
| LCons St Sts' ⇒ LCons St0 (theSts0 (lift St0 St) Sts'))

lemma *theSts0-LNil[simp]: theSts0 St0 LNil = LCons St0 LNil*
by *(subst theSts0.code) auto*

lemma *theSts0-LCons[simp]: theSts0 St0 (LCons St Sts') = LCons St0 (theSts0 (lift St0 St) Sts')*
by *(subst theSts0.code) auto*

lemma *simul-chain0*:
assumes *chain: lnull Sts ∨ (chain (~) Sts ∧ erase St0 ~ lhd Sts)*
shows $\exists Sts0. \text{lhd } Sts0 = St0 \wedge \text{lmap } \text{erase } (\text{ltl } Sts0) = Sts \wedge \text{chain } (\sim 0) Sts0$
proof *(rule exI[of - theSts0 St0 Sts], safe)*
show *lhd (theSts0 St0 Sts) = St0*
by *(simp add: llist.case-eq-if)*

next
show *lmap erase (ltl (theSts0 St0 Sts)) = Sts*

```

using chain
apply (coinduction arbitrary: Sts St0)
using lift by (auto simp: llist.case-eq-if) (metis chain.simps eq-LConsD lnull-def)
next
{
  fix Sts'
  assume  $\exists St0 Sts. (lnull Sts \vee chain (\rightsquigarrow) Sts \wedge erase St0 \rightsquigarrow lhd Sts) \wedge Sts' = theSts0 St0 Sts$ 
  hence chain ( $\rightsquigarrow 0$ ) Sts'
  apply (coinduct rule: chain.coinduct)
  apply clarsimp
  apply (erule disjE)
  apply (metis lnull-def theSts0-LNil)
  by (smt (verit, ccfv-threshold) R0-lift chain.simps erase-lift lhd-LCons theSts0-LCons theSts0-LNil)
}
thus chain ( $\rightsquigarrow 0$ ) (theSts0 St0 Sts)
using assms by auto
qed

```

lemma simul-chain:

```

assumes
  chain: chain ( $\rightsquigarrow$ ) Sts and
  hd: lhd Sts = erase St0
shows  $\exists Sts0. lhd Sts0 = St0 \wedge lmap erase Sts0 = Sts \wedge chain (\rightsquigarrow 0) Sts0$ 
proof -
{
  assume nnul:  $\neg lnull (ltl Sts)$ 
  have chain ( $\rightsquigarrow$ ) (ltl Sts)  $\wedge$  erase St0  $\rightsquigarrow$  lhd (ltl Sts)
  (is ?thesis1  $\wedge$  ?thesis2)
  proof
  show ?thesis1
  by (simp add: nnul chain chain-ltl)
  next
  show ?thesis2
  by (metis chain chain-consE hd lhd-LCons-ltl lnull-def lnull-ltlI nnul)
}
qed
hence nil-or-chain:  $lnull (ltl Sts) \vee (chain (\rightsquigarrow) (ltl Sts) \wedge erase St0 \rightsquigarrow lhd (ltl Sts))$ 
by blast

```

obtain Sts0 **where**

```

hd-sts0: lhd Sts0 = St0 and
erase-tl-sts0: lmap erase (ltl Sts0) = ltl Sts and
chain-sts0: chain ( $\rightsquigarrow 0$ ) Sts0
using simul-chain0[OF nil-or-chain] by blast

```

```

have erase-hd-sts0: erase (lhd Sts0) = lhd Sts
by (simp add: hd hd-sts0)

```

```

have erase-sts0: lmap erase Sts0 = Sts

```

```

proof (cases Sts0 rule: llist.exhaust-sel)

```

```

  case LNil

```

```

  hence False

```

```

  using chain-LNil chain-sts0 by blast

```

```

thus ?thesis

```

```

    by blast
next
case LCons
note sts0 = this
show ?thesis
proof (cases Sts rule: llist.exhaust-sel)
  case LNil
  hence False
  using chain chain-LNil by blast
  thus ?thesis
  by blast
next
case LCons
note sts = this
show ?thesis
  by (subst sts0, subst sts, simp add: erase-hd-sts0 erase-tl-sts0)
qed
qed

show ?thesis
  by (rule exI[of - Sts0]) (use hd-sts0 erase-sts0 chain-sts0 in blast)
qed

end

```

14.5 Ghost–Ghostless Conversions, the Concrete Version

context *fair-zipperposition-loop-wo-ghosts*

begin

lemma

todo-of-wo-ghosts-of[simp]: todo-of (wo-ghosts-of St) = w-ghosts.todo-of St and
passive-of-wo-ghosts-of[simp]: passive-of (wo-ghosts-of St) = w-ghosts.passive-of St and
yy-of-wo-ghosts-of[simp]: yy-of (wo-ghosts-of St) = w-ghosts.yy-of St and
active-of-wo-ghosts-of[simp]: active-of (wo-ghosts-of St) = w-ghosts.active-of St
 by (cases St; simp)+

lemma *fair-ZL-step-imp-fair-ZL-wo-ghosts-step:*

assumes $St \rightsquigarrow_{ZLf} St'$
 shows $wo\text{-ghosts-of } St \rightsquigarrow_{ZLfw} wo\text{-ghosts-of } St'$
 using *assms* by cases (use *fair-ZL-wo-ghosts.intros* in *auto*)

lemma *fair-ZL-wo-ghosts-step-imp-fair-ZL-step:*

assumes $wo\text{-ghosts-of } St0 \rightsquigarrow_{ZLfw} St'$
 shows $\exists St0'. wo\text{-ghosts-of } St0' = St' \wedge St0 \rightsquigarrow_{ZLf} St0'$
 using *assms*

proof cases

case (compute-infer $T \iota0 T' A C P$)
 note $wo\text{-st0} = this(1)$ and $st' = this(2)$ and $rest = this(3-5)$

define $D :: 'f$ inference set where

$D = done\text{-of } St0$

define $St0' :: ('t, 'p, 'f)$ ZLf-state where

$St0' = (T', D \cup \{\iota0\}, p\text{-add } C P, None, A)$

have $wo\text{-st0}' : wo\text{-ghosts-of } St0' = St'$

```

unfolding  $St0'$ -def  $st'$  by simp

have  $st0$ :  $St0 = (T, D, P, None, A)$ 
  using  $wo-st0$  by (smt (verit)  $D$ -def fst-conv snd-conv wo-ghosts-of.elims)
have  $step0$ :  $St0 \rightsquigarrow_{ZLf} St0'$ 
  unfolding  $st0$   $St0'$ -def by (rule fair-ZL.compute-infer[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use  $wo-st0'$   $step0$  in blast)
next
case (choose-p  $P$   $T$   $A$ )
note  $wo-st0 = this(1)$  and  $st' = this(2)$  and  $rest = this(3)$ 

define  $D$  :: 'f inference set where
   $D = done-of$   $St0$ 
define  $St0'$  :: ('t, 'p, 'f) ZLf-state where
   $St0' = (T, D, p-remove (p-select P) P, Some (p-select P), A)$ 

have  $wo-st0'$ : wo-ghosts-of  $St0' = St'$ 
  unfolding  $St0'$ -def  $st'$  by simp

have  $st0$ :  $St0 = (T, D, P, None, A)$ 
  using  $wo-st0$  by (smt (verit)  $D$ -def fst-conv snd-conv wo-ghosts-of.elims)
have  $step0$ :  $St0 \rightsquigarrow_{ZLf} St0'$ 
  unfolding  $st0$   $St0'$ -def by (rule fair-ZL.choose-p[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use  $wo-st0'$   $step0$  in blast)
next
case (delete-fwd  $C$   $A$   $T$   $P$ )
note  $wo-st0 = this(1)$  and  $st' = this(2)$  and  $rest = this(3)$ 

define  $D$  :: 'f inference set where
   $D = done-of$   $St0$ 
define  $St0'$  :: ('t, 'p, 'f) ZLf-state where
   $St0' = (T, D, P, None, A)$ 

have  $wo-st0'$ : wo-ghosts-of  $St0' = St'$ 
  unfolding  $St0'$ -def  $st'$  by simp

have  $st0$ :  $St0 = (T, D, P, Some C, A)$ 
  using  $wo-st0$  by (smt (verit)  $D$ -def fst-conv snd-conv wo-ghosts-of.elims)
have  $step0$ :  $St0 \rightsquigarrow_{ZLf} St0'$ 
  unfolding  $st0$   $St0'$ -def by (rule fair-ZL.delete-fwd[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use  $wo-st0'$   $step0$  in blast)
next
case (simplify-fwd  $C'$   $C$   $A$   $T$   $P$ )
note  $wo-st0 = this(1)$  and  $st' = this(2)$  and  $rest = this(3,4)$ 

define  $D$  :: 'f inference set where
   $D = done-of$   $St0$ 
define  $St0'$  :: ('t, 'p, 'f) ZLf-state where
   $St0' = (T, D, P, Some C', A)$ 

```

```

have wo-st0': wo-ghosts-of St0' = St'
  unfolding St0'-def st' by simp

have st0: St0 = (T, D, P, Some C, A)
  using wo-st0 by (smt (verit) D-def fst-conv snd-conv wo-ghosts-of.elims)
have step0: St0 ~>ZLf St0'
  unfolding st0 St0'-def by (rule fair-ZL.simplify-fwd[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use wo-st0' step0 in blast)
next
case (delete-bwd C' A C T P)
note wo-st0 = this(1) and st' = this(2) and rest = this(3,4)

define D :: 'f inference set where
  D = done-of St0
define St0' :: ('t, 'p', 'f') ZLf-state where
  St0' = (T, D, P, Some C, A)

have wo-st0': wo-ghosts-of St0' = St'
  unfolding St0'-def st' by simp

have st0: St0 = (T, D, P, Some C, A |∪| {|C'|})
  using wo-st0 by (smt (verit) D-def fst-conv snd-conv wo-ghosts-of.elims)
have step0: St0 ~>ZLf St0'
  unfolding st0 St0'-def by (rule fair-ZL.delete-bwd[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use wo-st0' step0 in blast)
next
case (simplify-bwd C' A C'' C T P)
note wo-st0 = this(1) and st' = this(2) and rest = this(3-5)

define D :: 'f inference set where
  D = done-of St0
define St0' :: ('t, 'p', 'f') ZLf-state where
  St0' = (T, D, p-add C'' P, Some C, A)

have wo-st0': wo-ghosts-of St0' = St'
  unfolding St0'-def st' by simp

have st0: St0 = (T, D, P, Some C, A |∪| {|C'|})
  using wo-st0 by (smt (verit) D-def fst-conv snd-conv wo-ghosts-of.elims)
have step0: St0 ~>ZLf St0'
  unfolding st0 St0'-def by (rule fair-ZL.simplify-bwd[OF rest])

show ?thesis
  by (rule exI[of - St0']) (use wo-st0' step0 in blast)
next
case (schedule-infer lss A C T P)
note wo-st0 = this(1) and st' = this(2) and rest = this(3)

define D :: 'f inference set where
  D = done-of St0

```

```

define  $St0' :: ('t, 'p, 'f)$  ZLf-state where
   $St0' = (fold\ t\text{-add}\text{-l}\text{list}\ \iota s\ T, D - flat\text{-inferences}\text{-of}\ (mset\ \iota s), P, None, A \cup \{C\})$ 

have  $wo\text{-}st0' : wo\text{-ghosts}\text{-of}\ St0' = St'$ 
  unfolding  $St0'\text{-def}\ st'$  by simp

have  $st0 : St0 = (T, D, P, Some\ C, A)$ 
  using  $wo\text{-}st0$  by (smt (verit)  $D\text{-def}\ fst\text{-conv}\ snd\text{-conv}\ wo\text{-ghosts}\text{-of}\ elims$ )
have  $step0 : St0 \rightsquigarrow_{ZLf} St0'$ 
  unfolding  $st0\ St0'\text{-def}$  by (rule fair-ZL.schedule-infer[OF rest])

show ?thesis
  by (rule exI[of -  $St0'$ ]) (use  $wo\text{-}st0'\ step0$  in blast)
next
case (delete-orphan-infers  $\iota s\ T\ A\ P\ Y$ )
note  $wo\text{-}st0 = this(1)$  and  $st' = this(2)$  and  $rest = this(3,4)$ 

define  $D :: 'f$  inference set where
   $D = done\text{-of}\ St0$ 
define  $St0' :: ('t, 'p, 'f)$  ZLf-state where
   $St0' = (t\text{-remove}\text{-l}\text{list}\ \iota s\ T, D \cup lset\ \iota s, P, Y, A)$ 

have  $wo\text{-}st0' : wo\text{-ghosts}\text{-of}\ St0' = St'$ 
  unfolding  $St0'\text{-def}\ st'$  by simp

have  $st0 : St0 = (T, D, P, Y, A)$ 
  using  $wo\text{-}st0$  by (smt (verit)  $D\text{-def}\ fst\text{-conv}\ snd\text{-conv}\ wo\text{-ghosts}\text{-of}\ elims$ )
have  $step0 : St0 \rightsquigarrow_{ZLf} St0'$ 
  unfolding  $st0\ St0'\text{-def}$  by (rule fair-ZL.delete-orphan-infers[OF rest])

show ?thesis
  by (rule exI[of -  $St0'$ ]) (use  $wo\text{-}st0'\ step0$  in blast)
qed

interpretation bisim: bisim  $wo\text{-ghosts}\text{-of}\ (\rightsquigarrow_{ZLfw})\ (\rightsquigarrow_{ZLf})$ 
proof qed (fact fair-ZL-wo-ghosts-step-imp-fair-ZL-step)

lemma chain-fair-ZL-step-wo-ghosts-imp-chain-fair-ZL-step:
  assumes chain: chain  $(\rightsquigarrow_{ZLfw})\ Sts$ 
  shows  $\exists Sts0. lmap\ wo\text{-ghosts}\text{-of}\ Sts0 = Sts \wedge chain\ (\rightsquigarrow_{ZLf})\ Sts0 \wedge done\text{-of}\ (lhd\ Sts0) = \{\}$ 
proof -
  define  $St0 :: ('t, 'p, 'f)$  ZLf-state where
     $St0 = (todo\text{-of}\ (lhd\ Sts), \{\}, passive\text{-of}\ (lhd\ Sts), yy\text{-of}\ (lhd\ Sts), active\text{-of}\ (lhd\ Sts))$ 

  have  $hd : lhd\ Sts = wo\text{-ghosts}\text{-of}\ St0$ 
    unfolding  $St0\text{-def}$  by (cases  $lhd\ Sts$ ) auto

  obtain  $Sts0$  where
     $wog0 : lmap\ wo\text{-ghosts}\text{-of}\ Sts0 = Sts$  and
     $chain0 : chain\ (\rightsquigarrow_{ZLf})\ Sts0$  and
     $hd0 : lhd\ Sts0 = St0$ 
    using bisim.simul-chain[OF chain hd] by blast

  have  $don0 : done\text{-of}\ (lhd\ Sts0) = \{\}$ 
    unfolding  $hd0\ St0\text{-def}$  by simp

```

show *?thesis*
using *wog0 chain0 don0* **by** *blast*
qed

lemma *full-chain-fair-ZL-step-wo-ghosts-imp-full-chain-fair-ZL-step*:
assumes *full-chain* (\rightsquigarrow ZLfw) *Sts*
shows \exists *Sts0*. *Sts* = *lmap wo-ghosts-of Sts0* \wedge *full-chain* (\rightsquigarrow ZLf) *Sts0* \wedge *done-of* (*lhd Sts0*) = {}
by (*smt* (*verit*) *assms chain-fair-ZL-step-wo-ghosts-imp-chain-fair-ZL-step empty-def*
fair-ZL-step-imp-fair-ZL-wo-ghosts-step full-chain-iff-chain full-chain-not-lnull lfinite-lmap
llast-lmap llist.map-disc-iff passive.felems-empty todo.llists-empty)

14.6 Completeness

theorem

assumes

full: *full-chain* (\rightsquigarrow ZLfw) *Sts* **and**

init: *is-initial-ZLf-wo-ghosts-state* (*lhd Sts*) **and**

fair: *infinitely-often compute-infer-step Sts* \longrightarrow *infinitely-often choose-p-step Sts*

shows

fair-ZL-wo-ghosts-Liminf-saturated: *saturated* (*labeled-formulas-of* (*Liminf-zl-fstate Sts*)) **and**

fair-ZL-wo-ghosts-complete-Liminf: $B \in \text{Bot-F} \implies$

passive elems (*passive-of* (*lhd Sts*)) $\models \cap \mathcal{G} \{B\} \implies$

$\exists B' \in \text{Bot-F}$. $B' \in \text{formulas-union}$ (*Liminf-zl-fstate Sts*) **and**

fair-ZL-wo-ghosts-complete: $B \in \text{Bot-F} \implies \text{passive elems}$ (*passive-of* (*lhd Sts*)) $\models \cap \mathcal{G} \{B\} \implies$

$\exists i$. *enat* $i < \text{llength Sts} \wedge (\exists B \in \text{Bot-F}$. $B \in \text{all-formulas-of}$ (*lth Sts* i))

proof –

obtain *Sts0* :: (*t*, *p*, *f*) *ZLf-state* *llist* **where**

full0: *full-chain* (\rightsquigarrow ZLf) *Sts0* **and**

sts0: *lmap wo-ghosts-of Sts0* = *Sts* **and**

don0: *done-of* (*lhd Sts0*) = {}

using *full-chain-fair-ZL-step-wo-ghosts-imp-full-chain-fair-ZL-step*[*OF full*] **by** *blast*

have *init0*: *is-initial-ZLf-state* (*lhd Sts0*)

proof –

have *hd*: *lhd* (*lmap wo-ghosts-of Sts0*) = *wo-ghosts-of* (*lhd Sts0*)

using *full0 full-chain-not-lnull llist.map-sel*(1) **by** *blast*

show *?thesis*

by (*rule is-initial-ZLf-wo-ghosts-state-imp-is-initial-ZLf-state*[*OF*
init[*unfolded sts0*[*symmetric*] *hd*] *don0*])

qed

have *fair0*: *infinitely-often w-ghosts.compute-infer-step Sts0* \longrightarrow

infinitely-often w-ghosts.choose-p-step Sts0

proof

assume *inf-ci0*: *infinitely-often w-ghosts.compute-infer-step Sts0*

have *infinitely-often compute-infer-step Sts*

unfolding *sts0*[*symmetric*]

by (*rule infinitely-often-lifting*[*of* - λx . x , *unfolded llist.map-ident*, *OF* - *inf-ci0*])

(*use w-ghosts.compute-infer-step-imp-compute-infer-step in auto*)

hence *inf-cp*: *infinitely-often choose-p-step Sts*

by (*simp add: fair*)

show *infinitely-often w-ghosts.choose-p-step Sts0*

by (*rule infinitely-often-lifting*[*of* - - - λx . x , *unfolded llist.map-ident*,

```

    OF - inf-cp[unfolded sts0[symmetric]])
    (use choose-p-step-imp-w-ghosts-choose-p-step in auto)
qed

have saturated (labeled-formulas-of (w-ghosts.Liminf-zl-fstate Sts0))
  using fair-ZL-Liminf-saturated[OF full0 init0 fair0] .
thus saturated (labeled-formulas-of (Liminf-zl-fstate Sts))
  unfolding w-ghosts.Liminf-zl-fstate-def Liminf-zl-fstate-def sts0[symmetric]
  by (simp add: llist.map-comp)

{
  assume
    bot: B ∈ Bot-F and
    unsat: passive.elems (passive-of (lhd Sts)) ⊨ $\cap\mathcal{G}$  {B}

  have unsat0: passive.elems (w-ghosts.passive-of (lhd Sts0)) ⊨ $\cap\mathcal{G}$  {B}
  proof -
    have lhd (lmap wo-ghosts-of Sts0) = wo-ghosts-of (lhd Sts0)
      using full0 full-chain-not-lnull llist.map-sel(1) by blast
    hence passive-of (lhd (lmap wo-ghosts-of Sts0)) = w-ghosts.passive-of (lhd Sts0)
      by simp
    thus ?thesis
      using unsat unfolding sts0[symmetric] by auto
  qed

  have  $\exists B' \in \text{Bot-F}. B' \in \text{formulas-union (w-ghosts.Liminf-zl-fstate Sts0)}$ 
    by (rule fair-ZL-complete-Liminf[OF full0 init0 fair0 bot unsat0])
  thus  $\exists B' \in \text{Bot-F}. B' \in \text{formulas-union (Liminf-zl-fstate Sts)}$ 
    unfolding w-ghosts.Liminf-zl-fstate-def Liminf-zl-fstate-def sts0[symmetric]
    by (simp add: llist.map-comp)
  thus  $\exists i. \text{enat } i < \text{llength Sts} \wedge (\exists B \in \text{Bot-F}. B \in \text{all-formulas-of (lth Sts } i))$ 
    unfolding Liminf-zl-fstate-def Liminf-llist-def by auto
}
qed
end

```

14.7 Specialization with FIFO Queue

As a proof of concept, we specialize the passive set to use a FIFO queue, thereby eliminating the locale assumptions about the passive set.

```

locale fifo-zipperposition-loop =
  discount-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q Equiv-F Prec-F
for
  Bot-F :: 'f set and
  Inf-F :: 'f inference set and
  Bot-G :: 'g set and
  Q :: 'q set and
  entails-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set  $\Rightarrow$  bool and
  Inf-G-q :: 'q  $\Rightarrow$  'g inference set and
  Red-I-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g inference set and
  Red-F-q :: 'q  $\Rightarrow$  'g set  $\Rightarrow$  'g set and
  G-F-q :: 'q  $\Rightarrow$  'f  $\Rightarrow$  'g set and
  G-I-q :: 'q  $\Rightarrow$  'f inference  $\Rightarrow$  'g inference set option and
  Equiv-F :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool (infix <math>\Leftarrow> 50) and

```



```

  Prec-F :: 'f ⇒ 'f ⇒ bool (infix <<·> 50) +
fixes
  Prec-S :: 'f ⇒ 'f ⇒ bool (infix <S 50)
assumes
  wf-Prec-S: minimal-element (<S) UNIV and
  transp-Prec-S: transp (<S) and
  countable-Inf-between: finite A ⇒ countable (no-labels.Inf-between A {C})
begin

sublocale fifo-prover-queue
  .

sublocale fifo-prover-lazy-list-queue
  .

sublocale fair-zipperposition-loop Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q Red-F-q G-F-q G-I-q
  Equiv-F Prec-F empty add-llist remove-llist pick-elem llists [] hd
  λy xs. if y ∈ set xs then xs else xs @ [y] removeAll fset-of-list Prec-S
proof
  show po-on (<S) UNIV
    using wf-Prec-S minimal-element.po by blast
next
  show wfp-on (<S) UNIV
    using wf-Prec-S minimal-element.wf by blast
next
  show transp (<S)
    by (rule transp-Prec-S)
next
  show ∧ A C. finite A ⇒ countable (no-labels.Inf-between A {C})
    by (fact countable-Inf-between)
qed

end

end

```

15 Given Clause Loops

This section imports all the theory files of the given clause procedure formalization.

```

theory Given-Clause-Loops
imports
  Fair-DISCOUNT-Loop
  Fair-Otter-Loop-Complete
  Fair-Zipperposition-Loop-without-Ghosts
begin
end

```