# Given Clause Loops

Jasmin Blanchette Qi Qiu Sophie Tourret

March 17, 2025

**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

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** $(\bigwedge y.\ y < b \Longrightarrow P\ y) \Longrightarrow P\ b$ **for** $b$ :: *bool*
  **hence** $\bigwedge q.\ q \leq b \Longrightarrow P\ q$
    **using** *less-bool-def* **by** *presburger*
  **then show** $P\ b$
    **by** *auto*
**qed**

**lemma** *finite-imp-set-eq*:
  **assumes** *fin*: *finite* $A$
  **shows** $\exists xs.\ set\ xs = A$
  **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 ⊆# Q
  shows ⋃ {f x |x. x ∈# P} ⊆ ⋃ {f x |x. x ∈# Q}
proof −
  have {f x |x. x ∈# P} ⊆ {f x |x. x ∈# 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) ∈ R ⟹ ({#x#}, {#y#}) ∈ mult1 R
  by (metis add-mset-add-single insert-DiffM mult1I single-eq-add-mset)

lemma singletons-in-mult: (x, y) ∈ R ⟹ ({#x#}, {#y#}) ∈ 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 ∩# 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
    ∀ i < llength Xs. lnth Xs i ⊆ lnth Ys i
  shows Liminf-llist Xs ⊆ Liminf-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 ∃ as. lset as = A
proof (cases finite A)
  case fin: True
  have ∃ 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-llist (from-nat-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 ∉ set (drop (Suc i) xs)
  by (metis Cons-nth-drop-Suc assms distinct.simps(2) distinct-drop)

lemma distinct-set-drop-removeAll-hd:
  assumes
    distinct xs
    xs ≠ []
  shows set (drop n (removeAll (hd xs) xs)) = set (drop (Suc n) xs)
  using assms
  by (metis distinct.simps(2) drop-Suc list.exhaust-sel removeAll.simps(2) removeAll-id)

lemma set-drop-removeAll: set (drop n (removeAll y xs)) ⊆ set (drop n 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: set (drop k (fold removeAll ys xs)) ⊆ set (drop k xs)
proof (induct ys arbitrary: xs)
  case (Cons y ys)
  note ih = this(1)

  have set (drop k (fold removeAll ys (removeAll y xs))) ⊆ set (drop k (removeAll y xs))
    using ih[of removeAll y xs] .
  also have ... ⊆ set (drop k xs)
    by (meson set-drop-removeAll)
  finally show ?case
    by simp
qed simp

lemma set-drop-append-subseteq: set (drop n (xs @ ys)) ⊆ set (drop n xs) ∪ 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-yxs: distinct (removeAll y xs)
    using dist-xs by (simp add: distinct-removeAll)

  show ?case
    by simp (rule ih[OF dist-yxs])
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*: *Inf-from* $\{\} = \{\iota \in \mathit{Inf}.\ \mathit{prems\text{-}of}\ \iota = []\}$
  **using** *Inf-from-def* **by** *auto*

**end**

## 2.2 Given Clause Procedure Basis

**context** *given-clause-basis*
**begin**

**lemma** *no-labels-entails-mono-left*: $M \subseteq N \Longrightarrow M \models\cap\mathcal{G}\ P \Longrightarrow N \models\cap\mathcal{G}\ P$
  **using** *no-labels.entails-trans no-labels.subset-entailed* **by** *blast*

**lemma** *no-labels-Red-F-imp-Red-F*:
  **assumes** $C \in$ *no-labels.Red-F* (*fst* ‘ $\mathcal{N}$)
  **shows** $(C,\ l) \in$ *Red-F* $\mathcal{N}$
**proof** −
  **let** *?N* = *fst* ‘ $\mathcal{N}$
  **have** *c-in-red-f-g-q*: $\forall\, q \in Q.\ C \in$ *no-labels.Red-F-$\mathcal{G}$-q q ?N*
    **using** *no-labels.Red-F-def assms* **by** *auto*
  **moreover have** *redfgq-eq-redfeq*:
    $\forall\, q \in Q.$ *no-labels.Red-F-$\mathcal{G}$-q q ?N* = *no-labels.Red-F-$\mathcal{G}$-empty-q q ?N*
    **using** *no-labels.Red-F-$\mathcal{G}$-empty-q-def no-labels.Red-F-$\mathcal{G}$-q-def* **by** *auto*
  **ultimately have** $\forall\, q \in Q.\ C \in$ *no-labels.Red-F-$\mathcal{G}$-empty-q q ?N*
    **by** *simp*
  **then have** $\forall\, q \in Q.$ *$\mathcal{G}$-F-q q C* $\subseteq$ *Red-F-q q* (*no-labels.$\mathcal{G}$-Fset-q q ?N*)
    **using** *redfgq-eq-redfeq no-labels.Red-F-$\mathcal{G}$-q-def* **by** *auto*
  **moreover have** $\forall\, q \in Q.$ *$\mathcal{G}$-F-L-q q* $(C,\ l)$ = *$\mathcal{G}$-F-q q C*
    **by** *simp*
  **moreover have** $\forall\, q \in Q.$ *no-labels.$\mathcal{G}$-Fset-q q ?N* = *$\mathcal{G}$-Fset-q q* $\mathcal{N}$
    **by** *auto*
  **ultimately have** $\forall\, q \in Q.$ *$\mathcal{G}$-F-L-q q* $(C,\ l) \subseteq$ *Red-F-q q* (*$\mathcal{G}$-Fset-L-q q* $\mathcal{N}$)
    **by** *auto*
  **then have** $\forall\, q \in Q.\ (C,\ l) \in$ *Red-F-$\mathcal{G}$-q q* $\mathcal{N}$
    **using** *c-in-red-f-g-q Red-F-$\mathcal{G}$-q-def* **by** *force*
  **then show** $(C,\ l) \in$ *Red-F* $\mathcal{N}$
    **using** *Red-F-def* **by** *simp*
**qed**

**lemma** *succ-F-imp-Red-F*:
  **assumes**
    $C' \in$ *fst* ‘ $\mathcal{N}$ **and**
    $C' \prec\cdot C$
  **shows** $(C,\ l) \in$ *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: ∀ q ∈ Q. 𝒢-F-q q C ⊆ 𝒢-F-q q C′
    using assms prec-F-grounding by simp
  ultimately have ∀ q ∈ Q. 𝒢-F-L-q q (C, l) ⊆ 𝒢-F-L-q q (C, l)
    by auto
  then have ∀ q ∈ Q. (C, l) ∈ Red-F-𝒢-q q 𝒩
    using c′-l′-in c′-l′-ls-c-l g-f-q-included Red-F-𝒢-q-def by fastforce
  thus (C, l) ∈ Red-F 𝒩
    using Red-F-def by auto
qed

lemma succ-L-imp-Red-F:
  assumes
    (C′, l′) ∈ 𝒩 and
    C′ ≼· C and
    l′ ⊏L l
  shows (C, l) ∈ Red-F 𝒩
proof −
  have c′-l′-ls-c-l: (C′, l′) ⊏ (C, l)
    using Prec-FL-def assms by auto
  have c′-le-c: C′ ≼· C
    using assms by simp
  then show (C, l) ∈ Red-F 𝒩
  proof
    assume c′-ls-c: C′ ≺· C
    have C′ ∈ fst ‘ 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′ ≐ C
    have c-eq-c′: C ≐ C′
      using c′-eq-c equiv-equiv-F equivp-symp by force
    have ∀ q ∈ Q. 𝒢-F-q q C′ = 𝒢-F-q q C
      using c′-eq-c c-eq-c′ equiv-F-grounding subset-antisym by auto
    then have ∀ q ∈ Q. 𝒢-F-L-q q (C, l) = 𝒢-F-L-q q (C′, l′) by auto
    then have ∀ q ∈ Q. (C, l) ∈ Red-F-𝒢-q q 𝒩
      using assms(1) c′-l′-ls-c-l Red-F-𝒢-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]: fst ‘ (ℳ ∪ 𝒩) = fst ‘ ℳ ∪ fst ‘ 𝒩
  by (rule Set.image-Un)

lemma prj-labeledN-eq-N [simp]: fst ‘ {(C, l) | C. C ∈ N} = N
proof −
  let ?𝒩 = {(C, l) | C. C ∈ N}
  have fst‘ ?𝒩 = N
    proof
      show fst‘ ?𝒩 ⊆ N
        by fastforce
    next
      show fst‘ ?𝒩 ⊇ N
```

```
    proof
      fix x
      assume x ∈ N
      then have (x, l) ∈ ?𝒩
        by auto
      then show x ∈ fst‘ ?𝒩
        by force
    qed
  qed
  then show fst‘ ?𝒩 = N
    by simp
qed

end
```

## 2.3 Given Clause Procedure

```
context given-clause
begin

lemma remove-redundant:
  assumes (C, l) ∈ Red-F 𝒩
  shows 𝒩 ∪ {(C, l)} ⤳GC 𝒩
proof −
  have {(C, l)} ⊆ Red-F (𝒩 ∪ {})
    using assms by simp
  moreover have active-subset {} = {}
    using active-subset-def by simp
  ultimately show 𝒩 ∪ {(C, l)} ⤳GC 𝒩
    by (metis process sup-bot-right)
qed

lemma remove-redundant-no-label:
  assumes  C ∈ no-labels.Red-F (fst ‘ 𝒩)
  shows 𝒩 ∪ {(C, l)} ⤳GC 𝒩
proof −
  have (C, l) ∈ Red-F 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 ≠ active
  shows 𝒩 ⤳GC 𝒩 ∪ {(C, l)}
proof −
  have active-subset-C-l: active-subset {(C, l)} = {}
    using active-subset-def assms by simp
  also have {} ⊆ Red-F (𝒩 ∪ {(C, l)})
    by simp
  finally show 𝒩 ⤳GC 𝒩 ∪ {(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** $\mathcal{N} \cup \{(C, l)\} \leadsto GC\ \mathcal{N}$
**proof** $-$
**have** $C' \in fst\ `\ \mathcal{N}$
**by** (*metis assms*(*1*) *fst-conv rev-image-eqI*)
**then have** $\{(C, l)\} \subseteq Red\text{-}F\ (\mathcal{N})$
**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** $\mathcal{N} \cup \{(C, l)\} \leadsto GC\ \mathcal{N}$
**proof** $-$
**have** $(C, l) \in Red\text{-}F\ \mathcal{N}$
**using** *assms succ-L-imp-Red-F* **by** *auto*
**then show** $\mathcal{N} \cup \{(C, l)\} \leadsto GC\ \mathcal{N}$
**using** *remove-redundant* **by** *auto*
**qed**

**lemma** *relabel-inactive*:
**assumes**
$l' \sqsubset L\ l$ **and**
$l' \neq active$
**shows** $\mathcal{N} \cup \{(C, l)\} \leadsto GC\ \mathcal{N} \cup \{(C, l')\}$
**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** $(C, l) \in Red\text{-}F\ (\mathcal{N} \cup \{(C, l')\})$
**using** *assms succ-L-imp-Red-F*[*of - -* $\mathcal{N} \cup \{(C, l')\}$] **by** *auto*
**then have** $\{(C, l)\} \subseteq Red\text{-}F\ (\mathcal{N} \cup \{(C, l')\})$
**by** *auto*

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

  **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* ‘ $\mathcal{N}$)
  **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* ‘ $\mathcal{N}$
    **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** $(T, \mathcal{N} \cup \{(C, l)\}) \leadsto LGC\ (T, \mathcal{N})$
**using** *remove-redundant* **by** *auto*
**qed**

**lemma** *relabel-inactive*:
**assumes**
$l' \sqsubset L\ l$ **and**
$l' \neq active$
**shows** $(T, \mathcal{N} \cup \{(C, l)\}) \leadsto LGC\ (T, \mathcal{N} \cup \{(C, l')\})$
**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** $(C, l) \in$ *Red-F* $(\mathcal{N} \cup \{(C, l')\})$
**using** *assms succ-L-imp-Red-F*[*of* - - $\mathcal{N} \cup \{(C, l')\}$] **by** *auto*
**then have** $\{(C, l)\} \subseteq$ *Red-F* $(\mathcal{N} \cup \{(C, l')\})$
**by** *auto*

**then show** $(T, \mathcal{N} \cup \{(C, l)\}) \leadsto LGC\ (T, \mathcal{N} \cup \{(C, l')\})$
**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* ⇒ *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* ⇒ *DL-label* ⇒ *bool* (**infix** ‹⊏L› *50*) **where**
*DL-Prec-L l l'* ⟷ *nat-of-DL-label l* < *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* × '*l*) *inference. Infer* (*map fst* (*prems-of* $\iota_{FL}$)) (*fst* (*concl-of* $\iota_{FL}$)) ∈ *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* ⇒ '*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** $\mathcal{G}$*-F-q* :: '*q* ⇒ '*f* ⇒ '*g set*
**and** $\mathcal{G}$*-I-q* :: '*q* ⇒ '*f inference* ⇒ '*g inference set option*
\+ **fixes**
*Equiv-F* :: '*f* ⇒ '*f* ⇒ *bool* (**infix** ‹≐› *50*) **and**
*Prec-F* :: '*f* ⇒ '*f* ⇒ *bool* (**infix** ‹≺·› *50*)
**assumes**
*equiv-equiv-F*: *equivp* (≐) **and**
*wf-prec-F*: *wfp* (≺·) *transp* (≺·) **and**
*compat-equiv-prec*: *C1* ≐ *D1* ⟹ *C2* ≐ *D2* ⟹ *C1* ≺· *C2* ⟹ *D1* ≺· *D2* **and**
*equiv-F-grounding*: *q* ∈ *Q* ⟹ *C1* ≐ *C2* ⟹ $\mathcal{G}$*-F-q q C1* ⊆ $\mathcal{G}$*-F-q q C2* **and**
*prec-F-grounding*: *q* ∈ *Q* ⟹ *C2* ≺· *C1* ⟹ $\mathcal{G}$*-F-q q C1* ⊆ $\mathcal{G}$*-F-q q C2* **and**
*static-ref-comp*: *statically-complete-calculus Bot-F Inf-F* (⊨∩$\mathcal{G}$)
*no-labels.Red-I-*$\mathcal{G}$ *no-labels.Red-F-*$\mathcal{G}$*-empty* **and**
*inf-have-prems*: ιF ∈ *Inf-F* ⟹ *prems-of* ιF ≠ []
**begin**

**lemma** *po-DL-Prec-L*: *transp* (⊏L) *asymp* (⊏L)
**unfolding** *DL-Prec-L-def transp-def*
**by** *auto*

**lemma** *wfp-DL-Prec-L*: *wfp* (⊏L)
**unfolding** *DL-Prec-L-def*
**by**(*simp add*: *wfP-app*)

**lemma** *Active-minimal*: *l2* ≠ *Active* ⟹ *Active* ⊏L *l2*
**by** (*cases l2*) (*auto simp*: *DL-Prec-L-def*)

**lemma** *at-least-two-labels*: ∃ *l2*. *Active* ⊏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*
**proof** *unfold-locales*
**show** ⋀*B N*. ⟦*B* ∈ *Bot-F*; *no-labels.empty-ord.saturated N*; *N* ⊨∩$\mathcal{G}$ {*B*}⟧ ⟹ ∃ *B'*∈*Bot-F*. *B'* ∈ *N*
**using** *static-ref-comp statically-complete-calculus.statically-complete*
**by** *meson*
**qed** (*simp-all add*: *equiv-equiv-F wf-prec-F po-DL-Prec-L wfp-DL-Prec-L compat-equiv-prec*
*prec-F-grounding Active-minimal at-least-two-labels equiv-F-grounding*)

**notation** *lgc.step* (**infix** ‹⇝LGC› *50*)

## 3.2 Basic Definitions and Lemmas

**abbreviation** *c-dot-succ* :: '*f* ⇒ '*f* ⇒ *bool* (**infix** ‹·≻› *50*) **where**
*C* ·≻ *C'* ≡ *C'* ≺· *C*

**abbreviation** *sqsupset* :: *DL-label* ⇒ *DL-label* ⇒ *bool* (**infix** ‹⊐L› *50*) **where**
$l \sqsupset L\ l' \equiv l' \sqsubset L\ l$

**fun** *labeled-formulas-of* :: $'f$ *set* × $'f$ *set* × $'f$ *set* ⇒ ($'f$ × *DL-label*) *set* **where**
*labeled-formulas-of* $(P, Y, A) = \{(C, Passive) \mid C.\ C \in P\} \cup \{(C, YY) \mid C.\ C \in Y\} \cup \{(C, Active) \mid C.\ C \in A\}$

**lemma** *labeled-formulas-of-alt-def*:
*labeled-formulas-of* $(P, Y, A) =$
$(\lambda C.\ (C, Passive))$ ‘ $P \cup (\lambda C.\ (C, YY))$ ‘ $Y \cup (\lambda C.\ (C, Active))$ ‘ $A$
**by** *auto*

**fun**
*state* :: $'f$ *inference set* × $'f$ *set* × $'f$ *set* × $'f$ *set* ⇒ $'f$ *inference set* × ($'f$ × *DL-label*) *set*
**where**
*state* $(T, P, Y, A) = (T,$ *labeled-formulas-of* $(P, Y, A))$

**lemma** *state-alt-def*:
*state* $(T, P, Y, A) = (T, (\lambda C.\ (C, Passive))$ ‘ $P \cup (\lambda C.\ (C, YY))$ ‘ $Y \cup (\lambda C.\ (C, Active))$ ‘ $A)$
**by** *auto*

**inductive**
*DL* :: $'f$ *inference set* × ($'f$ × *DL-label*) *set* ⇒ $'f$ *inference set* × ($'f$ × *DL-label*) *set* ⇒ *bool*
(**infix** ‹⇝*DL*› *50*)
**where**
*compute-infer*: $\iota \in$ *no-labels.Red-I* $(A \cup \{C\}) \Longrightarrow$
*state* $(T \cup \{\iota\}, P, \{\}, A) \leadsto DL$ *state* $(T, P, \{C\}, A)$
| *choose-p*: *state* $(T, P \cup \{C\}, \{\}, A) \leadsto DL$ *state* $(T, P, \{C\}, A)$
| *delete-fwd*: $C \in$ *no-labels.Red-F* $A \vee (\exists C' \in A.\ C' \preceq\cdot\ C) \Longrightarrow$
*state* $(T, P, \{C\}, A) \leadsto DL$ *state* $(T, P, \{\}, A)$
| *simplify-fwd*: $C \in$ *no-labels.Red-F* $(A \cup \{C'\}) \Longrightarrow$
*state* $(T, P, \{C\}, A) \leadsto DL$ *state* $(T, P, \{C'\}, A)$
| *delete-bwd*: $C' \in$ *no-labels.Red-F* $\{C\} \vee C' \cdot\succ C \Longrightarrow$
*state* $(T, P, \{C\}, A \cup \{C'\}) \leadsto DL$ *state* $(T, P, \{C\}, A)$
| *simplify-bwd*: $C' \in$ *no-labels.Red-F* $\{C, C''\} \Longrightarrow$
*state* $(T, P, \{C\}, A \cup \{C'\}) \leadsto DL$ *state* $(T, P \cup \{C''\}, \{C\}, A)$
| *schedule-infer*: $T' =$ *no-labels.Inf-between* $A\ \{C\} \Longrightarrow$
*state* $(T, P, \{C\}, A) \leadsto DL$ *state* $(T \cup T', P, \{\}, A \cup \{C\})$
| *delete-orphan-infers*: $T' \cap$ *no-labels.Inf-from* $A = \{\} \Longrightarrow$
*state* $(T \cup T', P, Y, A) \leadsto DL$ *state* $(T, P, Y, A)$

**lemma** *If-f-in-A-then-fl-in-PYA*: $C' \in A \Longrightarrow (C', Active) \in$ *labeled-formulas-of* $(P, Y, A)$
**by** *auto*

**lemma** *PYA-add-passive-formula*[*simp*]:
*labeled-formulas-of* $(P, Y, A) \cup \{(C, Passive)\} =$ *labeled-formulas-of* $(P \cup \{C\}, Y, A)$
**by** *auto*

**lemma** *P0A-add-y-formula*[*simp*]:
*labeled-formulas-of* $(P, \{\}, A) \cup \{(C, YY)\} =$ *labeled-formulas-of* $(P, \{C\}, A)$
**by** *auto*

**lemma** *PYA-add-active-formula*[*simp*]:
*labeled-formulas-of* $(P, Y, A) \cup \{(C', Active)\} =$ *labeled-formulas-of* $(P, Y, A \cup \{C'\})$
**by** *auto*

**lemma** *prj-active-subset-of-state*: *fst* ‘ *active-subset* (*labeled-formulas-of* $(P,\ Y,\ A)) = A$
**proof** −
  **have** *active-subset* $\{(C,\ YY) \mid C.\ C \in Y\} = \{\}$ **and**
      *active-subset* $\{(C,\ Passive) \mid C.\ C \in P\} = \{\}$
    **using** *active-subset-def* **by** *auto*
  **moreover have** *active-subset* $\{(C,\ Active) \mid C.\ C \in A\} = \{(C,\ Active) \mid C.\ C \in A\}$
    **using** *active-subset-def* **by** *fastforce*
  **ultimately have** *fst* ‘ *active-subset* (*labeled-formulas-of* $(P,\ Y,\ A)) =$
   *fst* ‘ $(\{(C,\ Active) \mid C.\ C \in A\})$
    **by** *simp*
  **then show** *?thesis*
    **by** *simp*
**qed**

**lemma** *active-subset-of-setOfFormulasWithLabelDiffActive*:
  $l \neq Active \Longrightarrow$ *active-subset* $\{(C',\ l)\} = \{\}$
  **using** *active-subset-def* **by** *auto*

## 3.3 Refinement

**lemma** *dl-compute-infer-in-lgc*:
  **assumes** $\iota \in$ *no-labels.Red-I-𝒢* $(A \cup \{C\})$
  **shows** *state* $(T \cup \{\iota\},\ P,\ \{\},\ A) \rightsquigarrow LGC$ *state* $(T,\ P,\ \{C\},\ A)$
**proof** −
  **let** $?\mathcal{N}$ = *labeled-formulas-of* $(P,\ \{\},\ A)$
  **and** $?\mathcal{M} = \{(C,\ YY)\}$
  **have** $A \cup \{C\} \subseteq$ *fst*‘ (*labeled-formulas-of* $(P,\ \{\},\ A) \cup \{(C,\ YY)\})$
    **by** *auto*
  **then have** $\iota \in$ *no-labels.Red-I-𝒢* (*fst*‘ $(?\mathcal{N} \cup ?\mathcal{M})$)
    **by** (*meson assms no-labels.empty-ord.Red-I-of-subset subsetD*)
  **also have** *active-subset* $?\mathcal{M} = \{\}$
    **using** *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *auto*
  **then have** $(T \cup \{\iota\},\ ?\mathcal{N}) \rightsquigarrow LGC\ (T,\ ?\mathcal{N} \cup ?\mathcal{M})$
    **using** *calculation lgc.step.compute-infer* **by** *blast*
  **moreover have** $?\mathcal{N} \cup ?\mathcal{M}$ = *labeled-formulas-of* $(P,\ \{C\},\ A)$
    **by** *simp*
  **ultimately show** *?thesis*
    **by** *auto*
**qed**

**lemma** *dl-choose-p-in-lgc*: *state* $(T,\ P \cup \{C\},\ \{\},\ A) \rightsquigarrow LGC$ *state* $(T,\ P,\ \{C\},\ A)$
**proof** −
  **let** $?\mathcal{N}$ = *labeled-formulas-of* $(P,\ \{\},\ A)$
  **have** *Passive* $\sqsupset L\ YY$
    **by** (*simp add*: *DL-Prec-L-def*)
  **then have** $(T,\ ?\mathcal{N} \cup \{(C,\ Passive)\}) \rightsquigarrow LGC\ (T,\ ?\mathcal{N} \cup \{(C,\ YY)\})$
    **using** *relabel-inactive* **by** *blast*
  **then have** $(T,$ *labeled-formulas-of* $(P \cup \{C\},\ \{\},\ A)) \rightsquigarrow LGC\ (T,$ *labeled-formulas-of* $(P,\ \{C\},\ A))$
     **by** (*metis PYA-add-passive-formula P0A-add-y-formula*)
  **then show** *?thesis*
    **by** *auto*
**qed**

**lemma** *dl-delete-fwd-in-lgc*:
  **assumes**  $(C \in$ *no-labels.Red-F* $A) \vee (\exists\, C' {\in} A.\ C' \preceq\cdot\ C)$

**shows** *state* $(T, P, \{C\}, A)$ ⇝*LGC* *state* $(T, P, \{\}, A)$
**using** *assms*
**proof**
**assume** *c-in*: $C \in$ *no-labels.Red-F* $A$
**then have** $A \subseteq$ *fst* ‘ (*labeled-formulas-of* $(P, \{\}, A)$)
**by** *simp*
**then have** $C \in$ *no-labels.Red-F* (*fst* ‘ (*labeled-formulas-of* $(P, \{\}, A)$))
**by** (*metis* (*no-types*, *lifting*) *c-in in-mono no-labels.Red-F-of-subset*)
**then show** *?thesis*
**using** *remove-redundant-no-label* **by** *auto*
**next**
**assume** $\exists C' \in A.\ C' \preceq\cdot C$
**then obtain** $C'$ **where** *c′-in-and-c′-ls-c*: $C' \in A \wedge C' \preceq\cdot C$
**by** *auto*
**then have** $(C', Active) \in$ *labeled-formulas-of* $(P, \{\}, A)$
**by** *auto*
**then have** $YY \sqsupset L$ *Active*
**by** (*simp add*: *DL-Prec-L-def*)
**then show** *?thesis*
**by** (*metis 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** $C \in$ *no-labels.Red-F-$\mathcal{G}$* $(A \cup \{C'\})$
**shows** *state* $(T, P, \{C\}, A)$ ⇝*LGC* *state* $(T, P, \{C'\}, A)$
**proof** −
**let** $?\mathcal{N}$ = *labeled-formulas-of* $(P, \{\}, A)$
**and** $?\mathcal{M} = \{(C, YY)\}$
**and** $?\mathcal{M}' = \{(C', YY)\}$
**have** $A \cup \{C'\} \subseteq$ *fst*‘ $(?\mathcal{N} \cup ?\mathcal{M}')$
**by** *auto*
**then have** $C \in$ *no-labels.Red-F-$\mathcal{G}$* (*fst*‘ $(?\mathcal{N} \cup ?\mathcal{M}')$)
**by** (*smt* (*verit*, *ccfv-threshold*) *assms no-labels.Red-F-of-subset subset-iff*)
**then have** $(C, YY) \in$ *Red-F* $(?\mathcal{N} \cup ?\mathcal{M}')$
**using** *no-labels-Red-F-imp-Red-F* **by** *simp*
**then have** $?\mathcal{M} \subseteq$ *Red-F-$\mathcal{G}$* $(?\mathcal{N} \cup ?\mathcal{M}')$
**by** *simp*
**moreover have** *active-subset* $?\mathcal{M}' = \{\}$
**using** *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *blast*
**ultimately have** $(T,$ *labeled-formulas-of* $(P, \{\}, A) \cup \{(C, YY)\})$ ⇝*LGC* $(T,$ *labeled-formulas-of* $(P, \{\}, A) \cup \{(C', YY)\})$
**using** *process*[*of* - - $?\mathcal{M}$ - $?\mathcal{M}'$] **by** *auto*
**then show** *?thesis*
**by** *simp*
**qed**

**lemma** *dl-delete-bwd-in-lgc*:
**assumes** $C' \in$ *no-labels.Red-F-$\mathcal{G}$* $\{C\} \vee C' \cdot\succ C$
**shows** *state* $(T, P, \{C\}, A \cup \{C'\})$ ⇝*LGC* *state* $(T, P, \{C\}, A)$
**using** *assms*
**proof**
**let** $?\mathcal{N}$ = *labeled-formulas-of* $(P, \{C\}, A)$
**assume** *c′-in*: $C' \in$ *no-labels.Red-F-$\mathcal{G}$* $\{C\}$
**have** $\{C\} \subseteq$ *fst* ‘ $?\mathcal{N}$

**by** *simp*
**then have** $C' \in$ *no-labels.Red-F-*$\mathcal{G}$ (*fst*' *?*$\mathcal{N}$)
**by** (*metis* (*no-types*, *lifting*) *c'-in insert-Diff insert-subset no-labels.Red-F-of-subset*)
**then have** (*T*, *?*$\mathcal{N}$ ∪ {(*C'*, *Active*)}) ⇝*LGC* (*T*, *?*$\mathcal{N}$)
**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*) ∈ *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** $C' \in$ *no-labels.Red-F-*$\mathcal{G}$ {*C*, *C''*}
**shows** *state* (*T*, *P*, {*C*}, *A* ∪ {*C'*}) ⇝*LGC state* (*T*, *P* ∪ {*C''*}, {*C*}, *A*)
**proof** −
**let** *?*$\mathcal{M}$ = {(*C'*, *Active*)}
**and** *?*$\mathcal{M}'$ = {(*C''*, *Passive*)}
**and** *?*$\mathcal{N}$ = *labeled-formulas-of* (*P*, {*C*}, *A*)

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

**lemma** *dl-schedule-infer-in-lgc*:
**assumes** *T'* = *no-labels.Inf-between A* {*C*}
**shows** *state* (*T*, *P*, {*C*}, *A*) ⇝*LGC state* (*T* ∪ *T'*, *P*, {}, *A* ∪ {*C*})
**proof** −
**let** *?*$\mathcal{N}$ = *labeled-formulas-of* (*P*, {}, *A*)
**have** *fst* ' (*active-subset ?*$\mathcal{N}$) = *A*
**using** *prj-active-subset-of-state* **by** *blast*
**then have** *T'* = *no-labels.Inf-between* (*fst* ' (*active-subset ?*$\mathcal{N}$)) {*C*}
**using** *assms* **by** *auto*
**then have** (*T*, *labeled-formulas-of* (*P*, {}, *A*) ∪ {(*C*, *YY*)}) ⇝*LGC*
(*T* ∪ *T'*, *labeled-formulas-of* (*P*, {}, *A*) ∪ {(*C*, *Active*)})
**using** *lgc.step.schedule-infer* **by** *blast*
**then show** *?thesis*
**by** (*metis state.simps P0A-add-y-formula PYA-add-active-formula*)
**qed**

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

**theorem** *DL-step-imp-LGC-step*: $T\mathcal{M} \leadsto DL\ T\mathcal{M}' \Longrightarrow T\mathcal{M} \leadsto LGC\ T\mathcal{M}'$
**proof** (*induction rule*: *DL.induct*)
  **case** (*compute-infer* $\iota$ $A$ $C$ $T$ $P$)
  **then show** *?case*
    **using** *dl-compute-infer-in-lgc* **by** *blast*
**next**
  **case** (*choose-p* $T$ $P$ $C$ $A$)
  **then show** *?case*
    **using** *dl-choose-p-in-lgc* **by** *auto*
**next**
  **case** (*delete-fwd* $C$ $A$ $T$ $P$)
  **then show** *?case*
    **using** *dl-delete-fwd-in-lgc* **by** *auto*
**next**
  **case** (*simplify-fwd* $C$ $A$ $C'$ $T$ $P$)
  **then show** *?case*
    **using** *dl-simplify-fwd-in-lgc* **by** *blast*
**next**
  **case** (*delete-bwd* $C'$ $C$ $T$ $P$ $A$)
  **then show** *?case*
    **using** *dl-delete-bwd-in-lgc* **by** *blast*
**next**
  **case** (*simplify-bwd* $C'$ $C$ $C''$ $T$ $P$ $A$)
  **then show** *?case*
    **using** *dl-simplify-bwd-in-lgc* **by** *blast*
**next**
  **case** (*schedule-infer* $T'$ $A$ $C$ $T$ $P$)
  **then show** *?case*
    **using** *dl-schedule-infer-in-lgc* **by** *blast*
**next**
  **case** (*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* $(\leadsto DL)$ *Sts* **and**

*act*: *active-subset* (*snd* (*lhd Sts*)) = {} **and**
*pas*: *passive-subset* (*Liminf-llist* (*lmap snd Sts*)) = {} **and**
*no-prems-init*: ∀ ι ∈ *Inf-F*. *prems-of* ι = [] ⟶ ι ∈ *fst* (*lhd Sts*) **and**
*final-sched*: *Liminf-llist* (*lmap fst Sts*) = {}
**shows**
*DL-Liminf-saturated*: *saturated* (*Liminf-llist* (*lmap snd Sts*)) **and**
*DL-complete-Liminf*: *B* ∈ *Bot-F* ⟹ *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*B*} ⟹
∃ *BL* ∈ *Bot-FL*. *BL* ∈ *Liminf-llist* (*lmap snd Sts*) **and**
*DL-complete*: *B* ∈ *Bot-F* ⟹ *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*B*} ⟹
∃ *i*. *enat i* < *llength Sts* ∧ (∃ *BL* ∈ *Bot-FL*. *BL* ∈ *snd* (*lnth Sts i*))
**proof** −
**have** *lgc-chain*: *chain* (↝*LGC*) *Sts*
**using** *dl-chain DL-step-imp-LGC-step chain-mono* **by** *blast*

**show** *saturated* (*Liminf-llist* (*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* ∈ *Bot-F* **and**
*unsat*: *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*B*}

**show** ∃ *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*])
**then show** ∃ *i*. *enat i* < *llength Sts* ∧ (∃ *BL* ∈ *Bot-FL*. *BL* ∈ *snd* (*lnth Sts i*))
**unfolding** *Liminf-llist-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* (λ*y 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** $y \in$ *set xs*
**shows** *fold* ($\lambda y$ *xs. if* $y \in$ *set xs then xs else xs @* [$y$]) (*removeAll y ys*) *xs* =
*fold* ($\lambda y$ *xs. if* $y \in$ *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* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *llist* $\Rightarrow$ *bool* **where**
*finitely-often R xs* $\longleftrightarrow$
($\exists\, i.\ \forall\, j.\ i \leq j \longrightarrow$ *enat* (*Suc j*) < *llength xs* $\longrightarrow \neg$ *R* (*lnth xs j*) (*lnth xs* (*Suc j*)))

**abbreviation** *infinitely-often* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *llist* $\Rightarrow$ *bool* **where**
*infinitely-often R xs* $\equiv \neg$ *finitely-often R xs*

**lemma** *infinitely-often-alt-def*:
*infinitely-often R xs* $\longleftrightarrow$
($\forall\, i.\ \exists\, j.\ i \leq j \wedge$ *enat* (*Suc j*) < *llength xs* $\wedge$ *R* (*lnth xs j*) (*lnth xs* (*Suc j*)))
**unfolding** *finitely-often-def* **by** *blast*

**lemma** *infinitely-often-lifting*:
**assumes**
*r-imp-s*: $\forall\, x\, x'.$ *R* (*f x*) (*f x'*) $\longrightarrow$ *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* :: $'q \Rightarrow 'e$ **and**
*add* :: $'e \Rightarrow 'q \Rightarrow 'q$ **and**
*remove* :: $'e \Rightarrow 'q \Rightarrow 'q$ **and**
*felems* :: $'q \Rightarrow 'e$ *fset*
**assumes**
*felems-empty*[*simp*]: *felems empty* = {||} **and**
*felems-not-empty*: $Q \neq$ *empty* $\Longrightarrow$ *felems Q* $\neq$ {||} **and**
*select-in-felems*[*simp*]: $Q \neq$ *empty* $\Longrightarrow$ *select Q* |∈| *felems Q* **and**
*felems-add*[*simp*]: *felems* (*add e Q*) = {|*e*|} |∪| *felems Q* **and**
*felems-remove*[*simp*]: *felems* (*remove e Q*) = *felems Q* |−| {|*e*|} **and**
*add-again*: *e* |∈| *felems Q* $\Longrightarrow$ *add e Q* = *Q*

**begin**

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

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

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

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

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

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

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

**lemma** *queue-step-idleI*: *queue-step* $Q$ $Q$
**using** *queue-step-fold-addI*[*of* - [], *simplified*] **.**

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

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

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

**end**

**locale** *fair-prover-queue* $=$ *prover-queue empty select add remove felems*
**for**
*empty* :: $'q$ **and**
*select* :: $'q \Rightarrow 'e$ **and**
*add* :: $'e \Rightarrow 'q \Rightarrow 'q$ **and**
*remove* :: $'e \Rightarrow 'q \Rightarrow 'q$ **and**
*felems* :: $'q \Rightarrow 'e$ *fset* +
**assumes** *fair*: *chain queue-step Qs* $\Longrightarrow$ *infinitely-often select-queue-step Qs* $\Longrightarrow$
*lhd Qs* $=$ *empty* $\Longrightarrow$ *Liminf-llist* (*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** $\bigwedge Q.\ Q \neq [] \Longrightarrow$ *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.prems 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.prems])
  then show ?case
    using queue-step-preserves-distinct ih by blast
qed

sublocale fair-prover-queue [] hd λy xs. if y ∈ 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) ≠ {}

    obtain i :: nat where
      i-lt: enat i < llength Qs and
      inter-nemp: ⋂ ((set ∘ lnth Qs) ` {j. i ≤ j ∧ enat j < llength Qs}) ≠ {}
      using lim-nemp unfolding Liminf-llist-def by auto

    from inter-nemp obtain e :: 'e where
      ∀ Q ∈ lnth Qs ` {j. i ≤ j ∧ enat j < llength Qs}. e ∈ set Q
      by auto
    hence c-in: ∀ j ≥ i. enat j < llength Qs ⟶ e ∈ set (lnth Qs j)
      by auto

    have ps-inf: llength Qs = ∞
    proof (rule ccontr)
      assume llength Qs ≠ ∞
      obtain n :: nat where
        n: enat n = llength Qs
        using ‹llength Qs ≠ ∞› 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': ∀ j ≥ i. e ∈ set (lnth Qs j)
      by (simp add: c-in ps-inf)
```

**then obtain** *k* :: *nat* **where**
  *k-lt*: *k* < *length* (*lnth Qs i*) **and**
  *at-k*: *lnth Qs i* ! *k* = *e*
  **by** (*meson in-set-conv-nth le-refl*)

**have** *dist*: *distinct* (*lnth Qs i*)
  **by** (*simp add*: *chain-queue-step-preserves-distinct hd-emp i-lt chain*)

**have** $\forall k' \leq k + 1.\ \exists i' \geq i.\ e \notin set\ (drop\ k'\ (lnth\ Qs\ i'))$
**proof** −
  **have** $\exists i' \geq i.\ e \notin set\ (drop\ (k + 1 - l)\ (lnth\ Qs\ i'))$ **for** *l*
  **proof** (*induct l*)
    **case** *0*
    **have** $e \notin set\ (drop\ (k + 1)\ (lnth\ Qs\ i))$
      **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'*: $e \notin set\ (drop\ (k + 1 - l)\ (lnth\ Qs\ i'))$
      **by** *blast*

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

    **have** *c-ni-i'-i''*: $e \notin set\ (drop\ (k + 1 - l)\ (lnth\ Qs\ j))$
      **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.prems*(*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.prems*(*2*) *linorder-not-less* **by** *blast*
        **next**
          **case** *False*
          **hence** *d-le*: $d \leq i''$
            **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* (λ*y 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* (λ*y 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* (λ*y 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**
    *wfp-Prec-S*: *wfp* (≺*S*) **and**
    *transp-Prec-S*: *transp* (≺*S*) **and**
    *finite-Inf-between*: *finite A* ⟹ *finite* (*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*)

**by** (*simp add*: *wfp-Prec-S wfp-imp-irreflp*)

**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* ⇒ '*p* **where**
*passive-of St* ≡ *fst St*
**abbreviation** *yy-of* :: ('*p*, '*f*) *DLf-state* ⇒ '*f option* **where**
*yy-of St* ≡ *fst* (*snd St*)
**abbreviation** *active-of* :: ('*p*, '*f*) *DLf-state* ⇒ '*f fset* **where**
*active-of St* ≡ *snd* (*snd St*)

**definition** *passive-inferences-of* :: '*p* ⇒ '*f inference set* **where**
*passive-inferences-of P* = {ι. *Passive-Inference* ι ∈ *elems P*}
**definition** *passive-formulas-of* :: '*p* ⇒ '*f set* **where**
*passive-formulas-of P* = {*C*. *Passive-Formula C* ∈ *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* ⇒ '*f set* **where**
*all-formulas-of St* ≡ *passive-formulas-of* (*passive-of St*) ∪ *set-option* (*yy-of St*) ∪
*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*) = {ι} ∪ *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* $\iota$) *P*) = *passive-inferences-of* *P* − {$\iota$}
**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*) $\iota$*s* *P*) = *passive-inferences-of* *P* − *set* $\iota$*s*
**by** (*induct* $\iota$*s* *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* $\iota$) *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*) $\iota$*s* *P*) = *passive-formulas-of* *P*
**by** (*induct* $\iota$*s* *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* $\iota$) *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* ∘ *Passive-Inference*) $\iota$*s* *P*) = *passive-formulas-of* *P*
**by** (*induct* $\iota$*s* *arbitrary*: *P*) *auto*

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

**fun** *fstate* :: (′*p*, ′*f*) *DLf-state* ⇒ ′*f inference set* × (′*f* × *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* ⇒ '*f set* × '*f set* × '*f set* **where**
*Liminf-fstate Sts* =
(*Liminf-llist* (*lmap* (*passive-formulas-of* ∘ *passive-of*) *Sts*),
*Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*),
*Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*))

**lemma** *Liminf-fstate-commute*:
*Liminf-llist* (*lmap* (*snd* ∘ *fstate*) *Sts*) = *labeled-formulas-of* (*Liminf-fstate Sts*)
**proof** −
**have** *Liminf-llist* (*lmap* (*snd* ∘ *fstate*) *Sts*) =
(λ*C*. (*C*, *Passive*)) ‘ *Liminf-llist* (*lmap* (*passive-formulas-of* ∘ *passive-of*) *Sts*) ∪
(λ*C*. (*C*, *YY*)) ‘ *Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*) ∪
(λ*C*. (*C*, *Active*)) ‘ *Liminf-llist* (*lmap* (*fset* ∘ *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* × '*f set* × '*f set* ⇒ '*f set* **where**
*formulas-union* (*P*, *Y*, *A*) = *P* ∪ *Y* ∪ *A*

**inductive** *fair-DL* :: ('*p*, '*f*) *DLf-state* ⇒ ('*p*, '*f*) *DLf-state* ⇒ *bool* (**infix** ‹⇝*DLf*› *50*) **where**
*compute-infer*: *P* ≠ *empty* ⟹ *select P* = *Passive-Inference* ι ⟹
ι ∈ *no-labels.Red-I* (*fset A* ∪ {*C*}) ⟹
(*P*, *None*, *A*) ⇝*DLf* (*remove* (*select P*) *P*, *Some C*, *A*)
| *choose-p*: *P* ≠ *empty* ⟹ *select P* = *Passive-Formula C* ⟹
(*P*, *None*, *A*) ⇝*DLf* (*remove* (*select P*) *P*, *Some C*, *A*)
| *delete-fwd*: *C* ∈ *no-labels.Red-F* (*fset A*) ∨ (∃ *C'* ∈ *fset A*. *C'* ⪯· *C*) ⟹
(*P*, *Some C*, *A*) ⇝*DLf* (*P*, *None*, *A*)
| *simplify-fwd*: *C'* ≺*S C* ⟹ *C* ∈ *no-labels.Red-F* (*fset A* ∪ {*C'*}) ⟹
(*P*, *Some C*, *A*) ⇝*DLf* (*P*, *Some C'*, *A*)
| *delete-bwd*: *C'* |∉| *A* ⟹ *C'* ∈ *no-labels.Red-F* {*C*} ∨ *C'* ·≻ *C* ⟹
(*P*, *Some C*, *A* |∪| {|*C'*|}) ⇝*DLf* (*P*, *Some C*, *A*)
| *simplify-bwd*: *C'* |∉| *A* ⟹ *C''* ≺*S C'* ⟹ *C'* ∈ *no-labels.Red-F* {*C*, *C''*} ⟹
(*P*, *Some C*, *A* |∪| {|*C'*|}) ⇝*DLf* (*add* (*Passive-Formula C''*) *P*, *Some C*, *A*)
| *schedule-infer*: *set* ι*s* = *no-labels.Inf-between* (*fset A*) {*C*} ⟹
(*P*, *Some C*, *A*) ⇝*DLf* (*fold* (*add* ∘ *Passive-Inference*) ι*s P*, *None*, *A* |∪| {|*C*|})
| *delete-orphan-infers*: ι*s* ≠ [] ⟹ *set* ι*s* ⊆ *passive-inferences-of P* ⟹
*set* ι*s* ∩ *no-labels.Inf-from* (*fset A*) = {} ⟹
(*P*, *Y*, *A*) ⇝*DLf* (*fold* (*remove* ∘ *Passive-Inference*) ι*s P*, *Y*, *A*)

## 5.3 Initial State and Invariant

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

```
inductive DLf-invariant :: ('p, 'f) DLf-state ⇒ bool where
  passive-inferences-of P ⊆ Inf-F ⟹ DLf-invariant (P, Y, A)

lemma initial-DLf-invariant: is-initial-DLf-state St ⟹ 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 ↝DLf St'
  shows DLf-invariant St'
  using step inv
proof cases
  case (schedule-infer ιs A C P)
  note defs = this(1,2) and ιs-inf-betw = this(3)
  have set ιs ⊆ Inf-F
    using ιs-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 (↝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 ↝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 (↝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* ⇒ *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* ⟷ (∀ *St'*. ¬ *St* ⇝*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** ∀ *St'*. ¬ *St* ⇝*DLf St'*
**unfolding** *st*
**proof** (*intro allI notI*)
**fix** *St'*
**assume** (*empty*, *None*, *A*) ⇝*DLf St'*
**thus** *False*
**by** *cases auto*
**qed**
**next**
**assume** *no-step*: ∀ *St'*. ¬ *St* ⇝*DLf St'*
**show** *is-final-DLf-state St*
**proof** (*rule ccontr*)
**assume** *not-fin*: ¬ *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* ≠ *empty* ∨ *Y* ≠ *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-ιs-uni-ιs*)
**qed**

**lemma** *fair-DL-step-imp-GC-step*:
  (*P*, *Y*, *A*) ⇝*DLf* (*P′*, *Y′*, *A′*) ⟹ *fstate* (*P*, *Y*, *A*) ⇝*LGC 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* :: (′*p*, ′*f*) *DLf-state* ⇒ ′*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 :: 'f multiset ⇒ 'f multiset ⇒ bool (infix ‹≺≺S› 50) where
  (≺≺S) ≡ multp (≺S)

lemma wfP-Precprec-S: wfP (≺≺S)
  by (simp add: wfp-Prec-S wfp-multp)

definition Less-state :: ('p, 'f) DLf-state ⇒ ('p, 'f) DLf-state ⇒ bool (infix ‹⊏› 50) where
  St' ⊏ St ⟷
   (yy-of St' = None ∧ yy-of St ≠ None)
   ∨ ((yy-of St' = None ⟷ yy-of St = None) ∧ mset-of-fstate St' ≺≺S mset-of-fstate St)

lemma wfP-Less-state: wfP (⊏)
proof −
  let ?boolset = {(b', b :: bool). b' < b}
  let ?msetset = {(M', M). M' ≺≺S M}
  let ?pair-of = λSt. (yy-of St ≠ None, mset-of-fstate St)

  have wf-boolset: wf ?boolset
    by (rule Wellfounded.wellorder-class.wf)
  have wf-msetset: wf ?msetset
    using wfP-Precprec-S wfp-def by auto
  have wf-lex-prod: wf (?boolset <*lex*> ?msetset)
    by (rule wf-lex-prod[OF wf-boolset wf-msetset])

  have Less-state-alt-def:
    ⋀St' St. St' ⊏ St ⟷ (?pair-of St', ?pair-of St) ∈ ?boolset <*lex*> ?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 ↝DLf St' and
    yy: yy-of St ≠ None ∨ yy-of St' = None
  shows St' ⊏ St
  using step
proof cases
  case (compute-infer P ι 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-multp)
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-multp)
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-multp)
  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* $\prec\prec S$ *?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**
**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* $\leadsto$*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* ($\leadsto$*DLf*) *Sts* **and**
*inv*: *DLf-invariant* (*lhd Sts*)
**shows** *Liminf-llist* (*lmap* (*set-option* $\circ$ *yy-of*) *Sts*) = {}
**proof** (*rule ccontr*)
**assume** *lim-nemp*: *Liminf-llist* (*lmap* (*set-option* $\circ$ *yy-of*) *Sts*) $\neq$ {}

**obtain** *i* :: *nat* **where**
*i-lt*: *enat i* < *llength Sts* **and**
*inter-nemp*: $\bigcap$ ((*set-option* $\circ$ *yy-of* $\circ$ *lnth Sts*) ‘ {*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*: ∀ *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** *si-lt*: *enat* (*Suc i*) < *llength Sts*
**unfolding** *len* **by** *auto*

**have** *yy-j*: *yy-of* (*lnth Sts j*) ≠ *None* **if** *j-ge*: *j* ≥ *i* **for** *j*
**using** *c-in′ len j-ge* **by** *auto*
**hence** *yy-sj*: *yy-of* (*lnth Sts* (*Suc j*)) ≠ *None* **if** *j-ge*: *j* ≥ *i* **for** *j*
**using** *le-Suc-eq that* **by** *presburger*
**have** *step*: *lnth Sts j* ⇝*DLf lnth Sts* (*Suc j*) **if** *j-ge*: *j* ≥ *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*) ⊏ *lnth Sts j* **if** *j-ge*: *j* ≥ *i* **for** *j*
**using** *yy-nonempty-DLf-step-imp-Less-state* **by** (*meson step j-ge yy-j yy-sj*)
**hence** $(\sqsubset)^{-1-1}$ (*lnth Sts j*) (*lnth Sts* (*Suc j*)) **if** *j-ge*: *j* ≥ *i* **for** *j*
**using** *j-ge* **by** *blast*
**hence** *inf-down-chain*: *chain* $(\sqsubset)^{-1-1}$ (*ldropn i Sts*)
**by** (*simp add*: *chain-ldropnI si-lt*)

**have** *inf-i*: ¬ *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* (⊏)] *wfP-Less-state*
**by** *metis*
**qed**

**lemma** *DLf-step-imp-queue-step*:
**assumes** *St* ⇝*DLf 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*: *llength Sts* = ∞ **and**
*full*: *full-chain* (⇝*DLf*) *Sts* **and**
*init*: *is-initial-DLf-state* (*lhd Sts*)
**shows** *Liminf-llist* (*lmap* (*elems* ∘ *passive-of*) *Sts*) = {}
**proof** −
**have** *chain-step*: *chain queue-step* (*lmap passive-of 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*:
∀ *j* ≥ *i*. ¬ *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 ⇝DLf lnth Sts (Suc j) if j-ge: j ≥ 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) ≠ None ∨ yy-of (lnth Sts (Suc j)) = None if j-ge: j ≥ i for j
      using step[OF j-ge]
    proof cases
      case (compute-infer P ι 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) ⊏ lnth Sts j if j-ge: j ≥ 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 (⊏)⁻¹⁻¹ (lnth Sts j) (lnth Sts (Suc j)) if j-ge: j ≥ i for j
      using j-ge by blast
    hence inf-down-chain: chain (⊏)⁻¹⁻¹ (ldropn i Sts)
      using chain-ldropn-lmapI[OF - si-lt, of - id, simplified llist.map-id] by simp

    have inf-i: ¬ 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 (⊏)] 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 (⇝DLf) Sts and
```

*init*: *is-initial-DLf-state* (*lhd Sts*)
**shows** *Liminf-llist* (*lmap* (*passive-formulas-of* ∘ *passive-of*) *Sts*) = {}
**proof** −
**have** *lim-filt*: *Liminf-llist* (*lmap* (*Set.filter is-passive-formula* ∘ *elems* ∘ *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* ∘ *elems* ∘ *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*) ⊆ *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* (λ*x*. *passive-formula* ‘
(*Set.filter is-passive-formula* ∘ *elems* ∘ *passive-of*) *x*) *Sts*) = {}
**using** *Liminf-llist-lmap-image*[*OF inj-pi*] *lim-filt* **by** *simp*

**have** *Liminf-llist* (*lmap* (λ*St*. {*C*. *Passive-Formula C* ∈ *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* (⇝*DLf*) *Sts* **and**
*init*: *is-initial-DLf-state* (*lhd Sts*)
**shows** *Liminf-llist* (*lmap* (*passive-inferences-of* ∘ *passive-of*) *Sts*) = {}
**proof** −
**have** *lim-filt*: *Liminf-llist* (*lmap* (*Set.filter is-passive-inference* ∘ *elems* ∘ *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* ∘ *elems* ∘ *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*) ⊆ *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*: *Liminf-llist* (*lmap* (λ*x*. *passive-inference* ‘
(*Set.filter is-passive-inference* ∘ *elems* ∘ *passive-of*) *x*) *Sts*) = {}
**using** *Liminf-llist-lmap-image*[*OF inj-pi*] *lim-filt* **by** *simp*

**have** *Liminf-llist* (*lmap* (λ*St*. {ι. *Passive-Inference* ι ∈ *elems* (*passive-of St*)}) *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* ($\rightsquigarrow$*DLf*) *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*: $B \in$ *Bot-F* $\Longrightarrow$ *passive-formulas-of* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists B' \in$ *Bot-F*. $B' \in$ *formulas-union* (*Liminf-fstate Sts*) **and**
*fair-DL-complete*: $B \in$ *Bot-F* $\Longrightarrow$ *passive-formulas-of* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists i$. *enat i* < *llength Sts* $\wedge$ ($\exists B' \in$ *Bot-F*. $B' \in$ *all-formulas-of* (*lnth Sts i*))
**proof** −
**have** *chain*: *chain* ($\rightsquigarrow$*DLf*) *Sts*
**by** (*rule full-chain-imp-chain*[*OF full*])
**hence** *dl-chain*: *chain* ($\rightsquigarrow$*DL*) (*lmap fstate 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*: ¬ *lnull Sts*
**using** *chain chain-not-lnull* **by** *blast*
**hence** *lhd-lmap*: $\bigwedge f$. *lhd* (*lmap f Sts*) = *f* (*lhd 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* ∘ *fstate*) *Sts*)) = {} $\wedge$
*Liminf-llist* (*lmap* (*fst* ∘ *fstate*) *Sts*) = {} (**is** *?pas-fml* $\wedge$ *?t-inf*)
**proof** (*cases lfinite Sts*)
**case** *fin*: *True*

**have** *lim-fst*: *Liminf-llist* (*lmap* (*fst* ∘ *fstate*) *Sts*) = *fst* (*fstate* (*llast Sts*)) **and**
*lim-snd*: *Liminf-llist* (*lmap* (*snd* ∘ *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** $\forall St'$. ¬ *llast Sts* $\rightsquigarrow$*DLf 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)) ⊨∩𝒢 {B}

    have unsat': fst ` snd (lhd (lmap fstate Sts)) ⊨∩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 ∃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
```

### 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 $\mathcal{G}$-F-q $\mathcal{G}$-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* ⇒ *'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**
    *$\mathcal{G}$-F-q* :: *'q* ⇒ *'f* ⇒ *'g set* **and**
    *$\mathcal{G}$-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*) +
  **fixes**
    *Prec-S* :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ‹≺S› *50*)
  **assumes**
    *wfp-Prec-S*: *wfp* (≺S) **and**
    *transp-Prec-S*: *transp* (≺S) **and**
    *finite-Inf-between*: *finite A* ⟹ *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 $\mathcal{G}$-F-q $\mathcal{G}$-I-q*
  *Equiv-F Prec-F* [] *hd* λ*y xs. if y* ∈ *set xs then xs else xs* @ [*y*] *removeAll fset-of-list Prec-S*
**proof**
  **show** *wfp* (≺S)
    **using** *wfp-Prec-S* **.**
**next**
  **show** *transp* (≺S)
    **by** (*rule transp-Prec-S*)
**next**
  **show** ⋀*A C*. *finite A* ⟹ *finite* (*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-label* =
*New* | *XX* | *Passive* | *YY* | *Active*

**primrec** *nat-of-OL-label* :: *OL-label* ⇒ *nat* **where**
*nat-of-OL-label New* = *4*
| *nat-of-OL-label XX* = *3*
| *nat-of-OL-label Passive* = *2*
| *nat-of-OL-label YY* = *1*
| *nat-of-OL-label Active* = *0*

**definition** *OL-Prec-L* :: *OL-label* ⇒ *OL-label* ⇒ *bool* (**infix** ‹⊏L› *50*) **where**
*OL-Prec-L l l′* ⟷ *nat-of-OL-label l* < *nat-of-OL-label l′*

**locale** *otter-loop* = *labeled-lifting-intersection Bot-F Inf-F Bot-G Q entails-q Inf-G-q Red-I-q*
*Red-F-q 𝒢-F-q 𝒢-I-q*
{$\iota_{FL}$ :: (′*f* × *OL-label*) *inference*. *Infer* (*map fst* (*prems-of* $\iota_{FL}$)) (*fst* (*concl-of* $\iota_{FL}$)) ∈ *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* ⇒ ′*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*
+ **fixes**
*Equiv-F* :: ′*f* ⇒ ′*f* ⇒ *bool* (**infix** ‹≐› *50*) **and**
*Prec-F* :: ′*f* ⇒ ′*f* ⇒ *bool* (**infix** ‹≺·› *50*)
**assumes**
*equiv-equiv-F*: *equivp* (≐) **and**
*wfp-prec-F*: *wfp* (≺·) *transp* (≺·) **and**
*compat-equiv-prec*: *C1* ≐ *D1* ⟹ *C2* ≐ *D2* ⟹ *C1* ≺· *C2* ⟹ *D1* ≺· *D2* **and**
*equiv-F-grounding*: *q* ∈ *Q* ⟹ *C1* ≐ *C2* ⟹ *𝒢-F-q q C1* ⊆ *𝒢-F-q q C2* **and**
*prec-F-grounding*: *q* ∈ *Q* ⟹ *C2* ≺· *C1* ⟹ *𝒢-F-q q C1* ⊆ *𝒢-F-q q C2* **and**
*static-ref-comp*: *statically-complete-calculus Bot-F Inf-F* (⊨∩𝒢)
*no-labels.Red-I-𝒢 no-labels.Red-F-𝒢-empty* **and**
*inf-have-prems*: *ιF* ∈ *Inf-F* ⟹ *prems-of ιF* ≠ []
**begin**

**lemma** *transp-OL-Prec-L*: *transp* (⊏L)
**unfolding** *OL-Prec-L-def transp-def* **by** *auto*

**lemma** *wfp-OL-Prec-L*: *wfp* (⊏L)
**unfolding** *OL-Prec-L-def* **by** (*simp add*: *wfP-app*)

**lemma** *Active-minimal*: *l2* ≠ *Active* ⟹ *Active* ⊏L *l2*
**by** (*cases l2*) (*auto simp*: *OL-Prec-L-def*)

**lemma** *at-least-two-labels*: ∃ *l2*. *Active* ⊏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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-I-q Equiv-F Prec-F OL-Prec-L Active*
**proof** *unfold-locales*
  **show** $\bigwedge B\ N.\ [\![B \in \textit{Bot-F};\ \textit{no-labels.empty-ord.saturated}\ N;\ N \models\cap\mathcal{G}\ \{B\}]\!] \Longrightarrow \exists\, B' \in \textit{Bot-F}.\ B' \in N$
    **using** *static-ref-comp statically-complete-calculus.statically-complete* **by** *fastforce*
**qed** (*simp-all add*: *equiv-equiv-F wfp-prec-F wfp-OL-Prec-L transp-OL-Prec-L compat-equiv-prec equiv-F-grounding prec-F-grounding Active-minimal at-least-two-labels inf-have-prems*)

**notation** *gc.step* (**infix** ‹$\leadsto$*GC*› *50*)

## 6.1 Basic Definitions and Lemmas

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

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

**inductive** *OL* :: $('f \times \textit{OL-label})\ set \Rightarrow ('f \times \textit{OL-label})\ set \Rightarrow \textit{bool}$ (**infix** ‹$\leadsto$*OL*› *50*) **where**
  *choose-n*: $C \notin N \Longrightarrow \textit{state}\ (N \cup \{C\},\ \{\},\ P,\ \{\},\ A) \leadsto OL\ \textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A)$
| *delete-fwd*: $C \in \textit{no-labels.Red-F}\ (P \cup A) \vee (\exists\, C' \in P \cup A.\ C' \preceq\cdot\ C) \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A) \leadsto OL\ \textit{state}\ (N,\ \{\},\ P,\ \{\},\ A)$
| *simplify-fwd*: $C \in \textit{no-labels.Red-F}\ (P \cup A \cup \{C'\}) \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A) \leadsto OL\ \textit{state}\ (N,\ \{C'\},\ P,\ \{\},\ A)$
| *delete-bwd-p*: $C' \in \textit{no-labels.Red-F}\ \{C\} \vee C \prec\cdot\ C' \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P \cup \{C'\},\ \{\},\ A) \leadsto OL\ \textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A)$
| *simplify-bwd-p*: $C' \in \textit{no-labels.Red-F}\ \{C,\ C''\} \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P \cup \{C'\},\ \{\},\ A) \leadsto OL\ \textit{state}\ (N \cup \{C''\},\ \{C\},\ P,\ \{\},\ A)$
| *delete-bwd-a*: $C' \in \textit{no-labels.Red-F}\ \{C\} \vee C \prec\cdot\ C' \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A \cup \{C'\}) \leadsto OL\ \textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A)$
| *simplify-bwd-a*: $C' \in \textit{no-labels.Red-F}\ (\{C,\ C''\}) \Longrightarrow$
    $\textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A \cup \{C'\}) \leadsto OL\ \textit{state}\ (N \cup \{C''\},\ \{C\},\ P,\ \{\},\ A)$
| *transfer*: $\textit{state}\ (N,\ \{C\},\ P,\ \{\},\ A) \leadsto OL\ \textit{state}\ (N,\ \{\},\ P \cup \{C\},\ \{\},\ A)$
| *choose-p*: $C \notin P \Longrightarrow \textit{state}\ (\{\},\ \{\},\ P \cup \{C\},\ \{\},\ A) \leadsto OL\ \textit{state}\ (\{\},\ \{\},\ P,\ \{C\},\ A)$
| *infer*: $\textit{no-labels.Inf-between}\ A\ \{C\} \subseteq \textit{no-labels.Red-I}\ (A \cup \{C\} \cup M) \Longrightarrow$
    $\textit{state}\ (\{\},\ \{\},\ P,\ \{C\},\ A) \leadsto OL\ \textit{state}\ (M,\ \{\},\ P,\ \{\},\ A \cup \{C\})$

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

**lemma** *active-subset-of-setOfFormulasWithLabelDiffActive*:
  $l \neq \textit{Active} \Longrightarrow \textit{active-subset}\ \{(C',\ l)\} = \{\}$
  **by** (*simp add*: *active-subset-def*)

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

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

**lemma** *state-add-C-Passive*: *state* $(N, X, P, Y, A) \cup \{(C, \mathit{Passive})\} =$ *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* $(N, X, P, Y, A) \cup \{(C, \mathit{Active})\} =$ *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* $(N, \{\}, P, \{\}, A) \cup \{(C, \mathit{New})\} \rightsquigarrow GC$ *state* $(N, \{\}, P, \{\}, A) \cup \{(C, XX)\}$
  **using** *relabel-inactive* **by** *blast*
 **have** *rewrite-left*: *state* $(N, \{\}, P, \{\}, A) \cup \{(C, \mathit{New})\} =$ *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* $(P \cup A) \vee (\exists C' \in P \cup A.\ C' \preceq\cdot C)$
 **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-labels.Red-F* $(P \cup A) \subseteq$ *no-labels.Red-F* $(N \cup \{\} \cup P \cup \{\} \cup A)$
  **using** *no-labels.Red-F-of-subset* **by** *simp*
 **then have** *c-in-redf-NPA*: $C \in$ *no-labels.Red-F* $(N \cup \{\} \cup P \cup \{\} \cup A)$
  **using** *c-in-redf-PA* **by** *auto*
 **have** *NPA-eq-prj-state-NPA*: $N \cup \{\} \cup P \cup \{\} \cup A =$ *fst* ' *state* $(N, \{\}, P, \{\}, A)$
  **using** *prj-state-union-sets* **by** *simp*
 **have** $C \in$ *no-labels.Red-F* (*fst* ' *state* $(N, \{\}, P, \{\}, A))$
  **using** *c-in-redf-NPA NPA-eq-prj-state-NPA* **by** *fastforce*
 **then show** *?thesis*
  **using** *remove-redundant-no-label* **by** *auto*
**next**
 **assume** $\exists C' \in P \cup A.\ C' \preceq\cdot C$
 **then obtain** $C'$ **where** $C' \in P \cup A$ **and** *c′-le-c*: $C' \preceq\cdot C$
  **by** *auto*
 **then have** $C' \in P \vee C' \in A$
  **by** *blast*
 **then show** *?thesis*
 **proof**
  **assume** $C' \in P$
  **then have** *c′-Passive-in*: $(C', \mathit{Passive}) \in$ *state* $(N, \{\}, P, \{\}, A)$

by *simp*
have *Passive* $\sqsubset L$ *XX*
by (*simp add*: *OL-Prec-L-def*)
then have *state* (*N*, {}, *P*, {}, *A*) ∪ {(*C*, *XX*)} $\rightsquigarrow$*GC state* (*N*, {}, *P*, {}, *A*)
using *remove-succ-L c′-le-c c′-Passive-in* by *blast*
then show *?thesis*
by *auto*
next
assume $C'$ ∈ *A*
then have *c′-Active-in-state-NPA*: ($C'$, *Active*) ∈ *state* (*N*, {}, *P*, {}, *A*)
by *simp*
also have *Active-ls-x*: *Active* $\sqsubset L$ *XX*
using *Active-minimal* by *simp*
then have *state* (*N*, {}, *P*, {}, *A*) ∪ {(*C*, *XX*)} $\rightsquigarrow$*GC state* (*N*, {}, *P*, {}, *A*)
using *remove-succ-L c′-le-c Active-ls-x c′-Active-in-state-NPA* by *blast*
then show *?thesis*
by *auto*
qed
qed

lemma *simplifyFwd-in-GC*:
*C* ∈ *no-labels.Red-F* (*P* ∪ *A* ∪ {$C'$}) $\Longrightarrow$
*state* (*N*, {*C*}, *P*, {}, *A*) $\rightsquigarrow$*GC state* (*N*, {$C'$}, *P*, {}, *A*)
proof −
assume *c-in*: *C* ∈ *no-labels.Red-F* (*P* ∪ *A* ∪ {$C'$})
let *?𝒩* = *state* (*N*, {}, *P*, {}, *A*)
and *?ℳ* = {(*C*, *XX*)} and *?ℳ′* = {($C'$, *XX*)}

have *P* ∪ *A* ∪ {$C'$} ⊆ *fst‘* (*?𝒩* ∪ *?ℳ′*)
by *auto*
then have *no-labels.Red-F* (*P* ∪ *A* ∪ {$C'$}) ⊆ *no-labels.Red-F* (*fst‘* (*?𝒩* ∪ *?ℳ′*))
using *no-labels.Red-F-of-subset* by *auto*
then have *C* ∈ *no-labels.Red-F* (*fst‘* (*?𝒩* ∪ *?ℳ′*))
using *c-in* by *auto*
then have *c-x-in*: (*C*, *XX*) ∈ *Red-F* (*?𝒩* ∪ *?ℳ′*)
using *no-labels-Red-F-imp-Red-F* by *auto*
then have *?ℳ* ⊆ *Red-F* (*?𝒩* ∪ *?ℳ′*)
by *auto*
then have *active-subset-of-m′*: *active-subset ?ℳ′* = {}
using *active-subset-of-setOfFormulasWithLabelDiffActive* by *auto*
show *?thesis*
using *c-x-in active-subset-of-m′ process*[*of - - ?ℳ - ?ℳ′*] by *auto*
qed

lemma *deleteBwdP-in-GC*:
assumes $C'$ ∈ *no-labels.Red-F* {*C*} ∨ *C* $\prec\cdot$ $C'$
shows *state* (*N*, {*C*}, *P* ∪ {$C'$}, {}, *A*) $\rightsquigarrow$*GC state* (*N*, {*C*}, *P*, {}, *A*)
using *assms*
proof
let *?𝒩* = *state* (*N*, {*C*}, *P*, {}, *A*)
assume *c-ls-c′*: *C* $\prec\cdot$ $C'$

have (*C*, *XX*) ∈ *state* (*N*, {*C*}, *P*, {}, *A*)
by *simp*
then have *?𝒩* ∪ {($C'$, *Passive*)} $\rightsquigarrow$*GC ?𝒩*

**using** *c-ls-c′ remove-succ-F* **by** *blast*
**also have** $?\mathcal{N} \cup \{(C', \textit{Passive})\} = \textit{state}\ (N, \{C\}, P \cup \{C'\}, \{\}, A)$
**by** *auto*
**finally show** *?thesis*
**by** *auto*
**next**
**let** $?\mathcal{N} = \textit{state}\ (N, \{C\}, P, \{\}, A)$
**assume** *c′-in-redf-c*: $C' \in \textit{no-labels.Red-F-}\mathcal{G}\ \{C\}$
**have** $\{C\} \subseteq \textit{fst}`\ ?\mathcal{N}$ **by** *auto*
**then have** $\textit{no-labels.Red-F}\ \{C\} \subseteq \textit{no-labels.Red-F}\ (\textit{fst}`\ ?\mathcal{N})$
**using** *no-labels.Red-F-of-subset* **by** *auto*
**then have** $C' \in \textit{no-labels.Red-F}\ (\textit{fst}`\ ?\mathcal{N})$
**using** *c′-in-redf-c* **by** *blast*
**then have** $?\mathcal{N} \cup \{(C', \textit{Passive})\} \leadsto GC\ ?\mathcal{N}$
**using** *remove-redundant-no-label* **by** *blast*
**then show** *?thesis*
**by** (*metis state-add-C-Passive*)
**qed**

**lemma** *simplifyBwdP-in-GC*:
**assumes** $C' \in \textit{no-labels.Red-F}\ \{C, C''\}$
**shows** $\textit{state}\ (N, \{C\}, P \cup \{C'\}, \{\}, A) \leadsto GC\ \textit{state}\ (N \cup \{C''\}, \{C\}, P, \{\}, A)$
**proof** –
**let** $?\mathcal{N} = \textit{state}\ (N, \{C\}, P, \{\}, A)$
**and** $?\mathcal{M} = \{(C', \textit{Passive})\}$
**and** $?\mathcal{M}' = \{(C'', \textit{New})\}$

**have** $\{C, C''\} \subseteq \textit{fst}`\ (?\mathcal{N} \cup ?\mathcal{M}')$
**by** (*smt* (*z3*) *Un-commute Un-empty-left Un-insert-right insert-absorb2 subset-Un-eq state-add-C-New prj-state-union-sets*)
**then have** $\textit{no-labels.Red-F}\ \{C, C''\} \subseteq \textit{no-labels.Red-F}\ (\textit{fst}`\ (?\mathcal{N} \cup ?\mathcal{M}'))$
**using** *no-labels.Red-F-of-subset* **by** *auto*
**then have** $C' \in \textit{no-labels.Red-F}\ (\textit{fst}`\ (?\mathcal{N} \cup ?\mathcal{M}'))$
**using** *assms* **by** *auto*
**then have** $(C', \textit{Passive}) \in \textit{Red-F}\ (?\mathcal{N} \cup ?\mathcal{M}')$
**using** *no-labels-Red-F-imp-Red-F* **by** *auto*
**then have** *$\mathcal{M}$-in-redf*: $?\mathcal{M} \subseteq \textit{Red-F}\ (?\mathcal{N} \cup ?\mathcal{M}')$ **by** *auto*

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

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

**lemma** *deleteBwdA-in-GC*:
**assumes** $C' \in \textit{no-labels.Red-F}\ \{C\} \lor C \prec\cdot C'$
**shows** $\textit{state}\ (N, \{C\}, P, \{\}, A \cup \{C'\}) \leadsto GC\ \textit{state}\ (N, \{C\}, P, \{\}, A)$
**using** *assms*

**proof**
  **let** *?𝒩* = *state* (*N*, {*C*}, *P*, {}, *A*)
  **assume** *c-ls-c′*: *C* ≺· *C′*

  **have** (*C*, *XX*) ∈ *state* (*N*, {*C*}, *P*, {}, *A*)
   **by** *simp*
  **then have** *?𝒩* ∪ {(*C′*, *Active*)} ⇝*GC* *?𝒩*
   **using** *c-ls-c′ remove-succ-F* **by** *blast*
  **also have** *?𝒩* ∪ {(*C′*, *Active*)} = *state* (*N*, {*C*}, *P*, {}, *A* ∪ {*C′*})
   **by** *auto*
  **finally show** *state* (*N*, {*C*}, *P*, {}, *A* ∪ {*C′*}) ⇝*GC* *state* (*N*, {*C*}, *P*, {}, *A*)
   **by** *auto*
**next**
  **let** *?𝒩* = *state* (*N*, {*C*}, *P*, {}, *A*)
  **assume** *c′-in-redf-c*: *C′* ∈ *no-labels.Red-F-𝒢* {*C*}

  **have** {*C*} ⊆ *fst‘ ?𝒩*
   **by** (*metis Un-commute Un-upper2 le-supI2 prj-state-union-sets*)
  **then have** *no-labels.Red-F* {*C*} ⊆ *no-labels.Red-F* (*fst‘ ?𝒩*)
   **using** *no-labels.Red-F-of-subset* **by** *auto*
  **then have** *C′* ∈ *no-labels.Red-F* (*fst‘ ?𝒩*)
   **using** *c′-in-redf-c* **by** *blast*
  **then have** *?𝒩* ∪ {(*C′*, *Active*)} ⇝*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′* ∈ *no-labels.Red-F* {*C*, *C″*}
 **shows** *state* (*N*, {*C*}, *P*, {}, *A* ∪ {*C′*}) ⇝*GC* *state* (*N* ∪ {*C″*}, {*C*}, *P*, {}, *A*)
**proof** −
 **let** *?𝒩* = *state* (*N*, {*C*}, *P*, {}, *A*) **and** *?ℳ* = {(*C′*, *Active*)} **and** *?ℳ′* = {(*C″*, *New*)}

 **have** {*C*, *C″*} ⊆ *fst‘* (*?𝒩* ∪ *?ℳ′*)
  **by** *simp*
 **then have** *no-labels.Red-F* {*C*, *C″*} ⊆ *no-labels.Red-F* (*fst‘* (*?𝒩* ∪ *?ℳ′*))
  **using** *no-labels.Red-F-of-subset* **by** *auto*
 **then have** *C′* ∈ *no-labels.Red-F* (*fst‘* (*?𝒩* ∪ *?ℳ′*))
  **using** *assms* **by** *auto*
 **then have** (*C′*, *Active*) ∈ *Red-F* (*?𝒩* ∪ *?ℳ′*)
  **using** *no-labels-Red-F-imp-Red-F* **by** *auto*
 **then have** *ℳ-included*: *?ℳ* ⊆ *Red-F* (*?𝒩* ∪ *?ℳ′*)
  **by** *auto*

 **have** *active-subset ?ℳ′* = {}
  **using** *active-subset-of-setOfFormulasWithLabelDiffActive* **by** *auto*
 **then have** *state* (*N*, {*C*}, *P*, {}, *A*) ∪ {(*C′*, *Active*)} ⇝*GC* *state* (*N*, {*C*}, *P*, {}, *A*) ∪ {(*C″*, *New*)}
  **using** *ℳ-included process*[**where** *?M*=*?ℳ* **and** *?M′*=*?ℳ′*] **by** *auto*
 **then show** *?thesis*
  **by** (*metis state-add-C-New state-add-C-Active*)
**qed**

**lemma** *transfer-in-GC*: *state* (*N*, {*C*}, *P*, {}, *A*) ⇝*GC* *state* (*N*, {}, *P* ∪ {*C*}, {}, *A*)
**proof** −

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

**have** *Passive* ⊏*L XX*
**by** (*simp add*: *OL-Prec-L-def*)
**then have** *?𝒩* ∪ {(*C*, *XX*)} ⤳*GC ?𝒩* ∪ {(*C*, *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*: *state* ({}, {}, *P* ∪ {*C*}, {}, *A*) ⤳*GC state* ({}, {}, *P*, {*C*}, *A*)
**proof** −
**let** *?𝒩* = *state* ({}, {}, *P*, {}, *A*)

**have** *YY* ⊏*L Passive*
**by** (*simp add*: *OL-Prec-L-def*)
**moreover have** *YY* ≠ *Active*
**by** *simp*
**ultimately have** *?𝒩* ∪ {(*C*, *Passive*)} ⤳*GC ?𝒩* ∪ {(*C*, *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*} ⊆ *no-labels.Red-I* (*A* ∪ {*C*} ∪ *M*)
**shows** *state* ({}, {}, *P*, {*C*}, *A*) ⤳*GC state* (*M*, {}, *P*, {}, *A* ∪ {*C*})
**proof** −
**let** *?ℳ* = {(*C′*, *New*) | *C′*. *C′* ∈ *M*}
**let** *?𝒩* = *state* ({}, {}, *P*, {}, *A*)

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

**have** *A* ∪ {*C*} ∪ *M* ⊆ (*fst‘ ?𝒩*) ∪ {*C*} ∪ (*fst‘ ?ℳ*)
**by** *fastforce*
**then have** *no-labels.Red-I* (*A* ∪ {*C*} ∪ *M*) ⊆ *no-labels.Red-I* ((*fst‘ ?𝒩*) ∪ {*C*} ∪ (*fst‘ ?ℳ*))
**using** *no-labels.empty-ord.Red-I-of-subset* **by** *auto*
**moreover have** *fst‘* (*active-subset ?𝒩*) = *A*
**using** *prj-ActiveSubset-of-state* **by** *blast*
**ultimately have** *no-labels.Inf-between* (*fst‘* (*active-subset ?𝒩*)) {*C*} ⊆
*no-labels.Red-I* ((*fst‘ ?𝒩*) ∪ {*C*} ∪ (*fst‘ ?ℳ*))
**using** *assms* **by** *auto*

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

**theorem** *OL-step-imp-GC-step*: *M* ⤳*OL M′* ⟹ *M* ⤳*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*: *chain* ($\leadsto$*OL*) *Sts* **and**
    *act*: *active-subset* (*lhd Sts*) = {} **and**
    *pas*: *passive-subset* (*Liminf-llist Sts*) = {}
  **shows**
    *OL-Liminf-saturated*: *saturated* (*Liminf-llist Sts*) **and**
    *OL-complete-Liminf*: $B \in$ *Bot-F* $\Longrightarrow$ *fst* ‘ *lhd Sts* $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
      $\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *Liminf-llist Sts* **and**
    *OL-complete*: $B \in$ *Bot-F* $\Longrightarrow$ *fst* ‘ *lhd Sts* $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
      $\exists$ *i*. *enat i* < *llength Sts* $\wedge$ ($\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *lnth Sts i*)
**proof** −

**have** *gc-chain*: *chain* (↝*GC*) *Sts*
  **using** *ol-chain OL-step-imp-GC-step chain-mono* **by** *blast*

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

**{**
  **assume**
    *bot*: $B \in$ *Bot-F* **and**
    *unsat*: *fst* ' *lhd Sts* $\models\cap\mathcal{G}$ $\{B\}$

  **show** $\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *Liminf-llist Sts*
    **by** (*rule gc-complete-Liminf*[*OF gc-chain act pas bot unsat*])
  **then show** $\exists$ *i*. *enat i* < *llength Sts* ∧ ($\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *lnth Sts i*)
    **unfolding** *Liminf-llist-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 fset* × '*f option* × '*p* × '*f option* × '*f 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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-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**
    $\mathcal{G}$*-F-q* :: '*q* ⇒ '*f* ⇒ '*g set* **and**
    $\mathcal{G}$*-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* **and**

*add* :: *'f* ⇒ *'p* ⇒ *'p* **and**
*remove* :: *'f* ⇒ *'p* ⇒ *'p* **and**
*felems* :: *'p* ⇒ *'f fset* +
**fixes**
*Prec-S* :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ‹≺*S*› *50*)
**assumes**
*wfp-Prec-S*: *wfp* (≺*S*) **and**
*transp-Prec-S*: *transp* (≺*S*) **and**
*finite-Inf-between*: *finite A* ⟹ *finite* (*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** *wfp-imp-irreflp wfp-Prec-S* **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*)

## 7.2 Basic Definitions and Lemmas

**abbreviation** *new-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'f fset* **where**
*new-of St* ≡ *fst St*
**abbreviation** *xx-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'f option* **where**
*xx-of St* ≡ *fst* (*snd St*)
**abbreviation** *passive-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'p* **where**
*passive-of St* ≡ *fst* (*snd* (*snd St*))
**abbreviation** *yy-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'f option* **where**
*yy-of St* ≡ *fst* (*snd* (*snd* (*snd St*)))
**abbreviation** *active-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'f fset* **where**
*active-of St* ≡ *snd* (*snd* (*snd* (*snd St*)))

**abbreviation** *all-formulas-of* :: (*'p*, *'f*) *OLf-state* ⇒ *'f set* **where**
*all-formulas-of St* ≡ *fset* (*new-of St*) ∪ *set-option* (*xx-of St*) ∪ *elems* (*passive-of St*) ∪
*set-option* (*yy-of St*) ∪ *fset* (*active-of St*)

**fun** *fstate* :: *'f fset* × *'f option* × *'p* × *'f option* × *'f fset* ⇒ (*'f* × *OL-label*) *set* **where**
*fstate* (*N*, *X*, *P*, *Y*, *A*) = *state* (*fset N*, *set-option X*, *elems P*, *set-option Y*, *fset A*)

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

**definition**
*Liminf-fstate* :: (*'p*, *'f*) *OLf-state llist* ⇒ *'f set* × *'f set* × *'f set* × *'f set* × *'f set*
**where**
*Liminf-fstate Sts* =
(*Liminf-llist* (*lmap* (*fset* ∘ *new-of*) *Sts*),
*Liminf-llist* (*lmap* (*set-option* ∘ *xx-of*) *Sts*),
*Liminf-llist* (*lmap* (*elems* ∘ *passive-of*) *Sts*),
*Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*),
*Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*))

**lemma** *Liminf-fstate-commute*: *Liminf-llist* (*lmap fstate Sts*) = *state* (*Liminf-fstate Sts*)
**proof** −
  **have** *Liminf-llist* (*lmap fstate Sts*) =
    ($\lambda C$. ($C$, *New*)) ‘ *Liminf-llist* (*lmap* (*fset* ∘ *new-of*) *Sts*) ∪
    ($\lambda C$. ($C$, *XX*)) ‘ *Liminf-llist* (*lmap* (*set-option* ∘ *xx-of*) *Sts*) ∪
    ($\lambda C$. ($C$, *Passive*)) ‘ *Liminf-llist* (*lmap* (*elems* ∘ *passive-of*) *Sts*) ∪
    ($\lambda C$. ($C$, *YY*)) ‘ *Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*) ∪
    ($\lambda C$. ($C$, *Active*)) ‘ *Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*)
    **unfolding** *fstate-alt-def state-alt-def*
    **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** *state-union* :: $'f$ *set* × $'f$ *set* × $'f$ *set* × $'f$ *set* × $'f$ *set* ⇒ $'f$ *set* **where**
  *state-union* ($N$, $X$, $P$, $Y$, $A$) = $N \cup X \cup P \cup Y \cup A$

**inductive** *fair-OL* :: ($'p$, $'f$) *OLf-state* ⇒ ($'p$, $'f$) *OLf-state* ⇒ *bool* (**infix** ‹⇝*OLf*› *50*) **where**
  *choose-n*: $C$ |∉| $N$ ⟹ ($N$ |∪| {|$C$|}, *None*, $P$, *None*, $A$) ⇝*OLf* ($N$, *Some* $C$, $P$, *None*, $A$)
| *delete-fwd*: $C$ ∈ *no-labels.Red-F* (*elems* $P$ ∪ *fset* $A$) ∨ (∃ $C'$ ∈ *elems* $P$ ∪ *fset* $A$. $C'$ ⪯· $C$) ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$) ⇝*OLf* ($N$, *None*, $P$, *None*, $A$)
| *simplify-fwd*: $C'$ ≺$S$ $C$ ⟹ $C$ ∈ *no-labels.Red-F* (*elems* $P$ ∪ *fset* $A$ ∪ {$C'$}) ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$) ⇝*OLf* ($N$, *Some* $C'$, $P$, *None*, $A$)
| *delete-bwd-p*: $C'$ ∈ *elems* $P$ ⟹ $C'$ ∈ *no-labels.Red-F* {$C$} ∨ $C$ ≺· $C'$ ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$) ⇝*OLf* ($N$, *Some* $C$, *remove* $C'$ $P$, *None*, $A$)
| *simplify-bwd-p*: $C''$ ≺$S$ $C'$ ⟹ $C'$ ∈ *elems* $P$ ⟹ $C'$ ∈ *no-labels.Red-F* {$C$, $C''$} ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$) ⇝*OLf* ($N$ |∪| {|$C''$|}, *Some* $C$, *remove* $C'$ $P$, *None*, $A$)
| *delete-bwd-a*: $C'$ |∉| $A$ ⟹ $C'$ ∈ *no-labels.Red-F* {$C$} ∨ $C$ ≺· $C'$ ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$ |∪| {|$C'$|}) ⇝*OLf* ($N$, *Some* $C$, $P$, *None*, $A$)
| *simplify-bwd-a*: $C''$ ≺$S$ $C'$ ⟹ $C'$ |∉| $A$ ⟹ $C'$ ∈ *no-labels.Red-F* {$C$, $C''$} ⟹
    ($N$, *Some* $C$, $P$, *None*, $A$ |∪| {|$C'$|}) ⇝*OLf* ($N$ |∪| {|$C''$|}, *Some* $C$, $P$, *None*, $A$)
| *transfer*: ($N$, *Some* $C$, $P$, *None*, $A$) ⇝*OLf* ($N$, *None*, *add* $C$ $P$, *None*, $A$)
| *choose-p*: $P$ ≠ *empty* ⟹
    ({||}, *None*, $P$, *None*, $A$) ⇝*OLf* ({||}, *None*, *remove* (*select* $P$) $P$, *Some* (*select* $P$), $A$)
| *infer*: *no-labels.Inf-between* (*fset* $A$) {$C$} ⊆ *no-labels.Red-I* (*fset* $A$ ∪ {$C$} ∪ *fset* $M$) ⟹
    ({||}, *None*, $P$, *Some* $C$, $A$) ⇝*OLf* ($M$, *None*, $P$, *None*, $A$ |∪| {|$C$|})

## 7.3 Initial State and Invariant

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

**inductive** *OLf-invariant* :: ($'p$, $'f$) *OLf-state* ⇒ *bool* **where**
  ($N$ = {||} ∧ $X$ = *None*) ∨ $Y$ = *None* ⟹ *OLf-invariant* ($N$, $X$, $P$, $Y$, $A$)

**lemma** *initial-OLf-invariant*: *is-initial-OLf-state St* ⟹ *OLf-invariant St*
  **unfolding** *is-initial-OLf-state.simps OLf-invariant.simps* **by** *auto*

**lemma** *step-OLf-invariant*:
  **assumes** *step*: *St* ⇝*OLf* *St'*
  **shows** *OLf-invariant St'*
  **using** *step* **by** *cases* (*auto intro*: *OLf-invariant.intros*)

**lemma** *chain-OLf-invariant-lnth*:

**assumes**
*chain*: *chain* (⇝*OLf*) *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-OLf-invariant* **by** *blast*
**qed**

**lemma** *chain-OLf-invariant-llast*:
**assumes**
*chain*: *chain* (⇝*OLf*) *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-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* ⇒ *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* ⟷ (∀ *St*′. ¬ *St* ⇝*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** ∀ *St*′. ¬ *St* ⇝*OLf St*′
**unfolding** *st*
**proof** (*intro allI notI*)
**fix** *St*′
**assume** ({||}, *None*, *empty*, *None*, *A*) ⇝*OLf St*′
**thus** *False*

**by** *cases auto*
**qed**
**next**
**assume** *no-step*: $\forall St'.\ \neg\ St \leadsto OLf\ St'$
**show** *is-final-OLf-state St*
**proof** (*rule ccontr*)
**assume** *not-fin*: ¬ *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* = {||} ∧ *X* = *None*) ∨ *Y* = *None*
**using** *inv st OLf-invariant.simps* **by** *simp*

**have** *N* ≠ {||} ∨ *X* ≠ *None* ∨ *P* ≠ *empty* ∨ *Y* ≠ *None*
**using** *not-fin* **unfolding** *st is-final-OLf-state.simps* **by** *auto*
**moreover {**
**assume**
*n*: *N* ≠ {||} **and**
*x*: *X* = *None*

**obtain** *N'* :: *'f fset* **and** *C* :: *'f* **where**
*n'*: *N* = *N'* |∪| {|*C*|} **and**
*c-ni*: *C* |∉| *N'*
**using** *n finsert-is-funion* **by** *blast*
**have** *y*: *Y* = *None*
**using** *n x inv'* **by** *meson*

**have** $\exists St'.\ St \leadsto OLf\ St'$
**using** *fair-OL.choose-n*[*OF c-ni*] **unfolding** *st n' x y* **by** *fast*
**} moreover {**
**assume** *X* ≠ *None*
**then obtain** *C* :: *'f* **where**
*x*: *X* = *Some C*
**by** *blast*

**have** *y*: *Y* = *None*
**using** *x inv'* **by** *auto*

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

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

**by** *blast*

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

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

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

**have** *inf-red*: *no-labels.Inf-between* (*fset A*) {*C*}
⊆ *no-labels.Red-I-$\mathcal{G}$* (*fset A* ∪ {*C*} ∪ *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** ∃ *St'*. *St* ⇝*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*) ⇝*OLf* (*N'*, *X'*, *P'*, *Y'*, *A'*)
**shows** *fstate* (*N*, *X*, *P*, *Y*, *A*) ⇝*OL fstate* (*N'*, *X'*, *P'*, *Y'*, *A'*)
**using** *olf*
**proof** *cases*
**case** (*choose-n C*)
**note** *defs* = *this*(*1*−*7*) **and** *c-ni* = *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*−*7*) **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*−*7*) **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*−*7*) **and** *c'-in-p* = *this*(*8*) **and** *c'-red* = *this*(*9*)

**have** *p-rm-c'-uni-c'*: *elems* (*remove C' P*) ∪ {*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(1−7) and c′-in-p = this(9) and c′-red = this(10)

  have p-rm-c′-uni-c′: elems (remove C′ P) ∪ {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(1−7) 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(1−7) 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(1−7)

  have p-uni-c: elems P ∪ {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 ∉ elems (remove (select P) P)
    unfolding felems-remove by auto

  have rm-sel-uni-sel: elems (remove (select P) P) ∪ {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*:
$(N, X, P, Y, A) \leadsto OLf\ (N', X', P', Y', A') \Longrightarrow$
$fstate\ (N, X, P, Y, A) \leadsto GC\ fstate\ (N', X', P', Y', A')$
**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* :: $('f \times OL\text{-}label)\ set \Rightarrow ('f \times OL\text{-}label)\ set \Rightarrow bool$ (**infix** ‹⇝IL› *50*)
**where**
*ol*: $St \leadsto OL\ St' \Longrightarrow St \leadsto IL\ St'$
| *red-by-children*: $C \in no\text{-}labels.Red\text{-}F\ (A \cup M) \lor (M = \{C'\} \land C' \prec\cdot C) \Longrightarrow$
$state\ (\{\}, \{\}, P, \{C\}, A) \leadsto IL\ state\ (M, \{\}, P, \{\}, A)$

## 8.2 Refinement

**lemma** *red-by-children-in-GC*:
**assumes** $C \in no\text{-}labels.Red\text{-}F\ (A \cup M) \lor (M = \{C'\} \land C' \prec\cdot C)$
**shows** $state\ (\{\}, \{\}, P, \{C\}, A) \leadsto GC\ state\ (M, \{\}, P, \{\}, A)$
**proof** −
**let** $?\mathcal{N} = state\ (\{\}, \{\}, P, \{\}, A)$
**and** $?St = \{(C, YY)\}$
**and** $?St' = \{(x, New)\ |x.\ x \in M\}$

**have** $(C, YY) \in Red\text{-}F\ (?\mathcal{N} \cup ?St')$
**using** *assms*
**proof**
**assume** *c-in*: $C \in no\text{-}labels.Red\text{-}F\ (A \cup M)$
**have** $A \cup M \subseteq A \cup M \cup P$ **by** *auto*
**also have** $fst\ `\ (?\mathcal{N} \cup ?St') = A \cup M \cup P$
**by** *auto*
**then have** $C \in no\text{-}labels.Red\text{-}F\ (fst\ `\ (?\mathcal{N} \cup ?St'))$
**by** (*metis* (*no-types*, *lifting*) *c-in calculation in-mono no-labels.Red-F-of-subset*)
**then show** $(C, YY) \in Red\text{-}F\ (?\mathcal{N} \cup ?St')$
**using** *no-labels-Red-F-imp-Red-F* **by** *blast*

```
  next
    assume assm: M = {C'} ∧ C' ≺· C
    then have C' ∈ fst ‘ (?𝒩 ∪ ?St')
      by simp
    then show (C, YY) ∈ Red-F (?𝒩 ∪ ?St')
      by (metis (mono-tags) assm succ-F-imp-Red-F)
  qed
  then have St-included-in: ?St ⊆ Red-F (?𝒩 ∪ ?St')
    by auto

  have prj-of-active-subset-of-St': fst ‘ (active-subset ?St') = {}
    by (simp add: active-subset-def)

  have ?𝒩 ∪ ?St ⇝GC ?𝒩 ∪ ?St'
    using process[of - ?𝒩 ?St - ?St'] St-included-in prj-of-active-subset-of-St' by auto
  moreover have ?𝒩 ∪ ?St = state ({}, {}, P, {C}, A)
    by simp
  moreover have ?𝒩 ∪ ?St' = state (M, {}, P, {}, A)
    by auto
  ultimately show state ({}, {}, P, {C}, A) ⇝GC state (M, {}, P, {}, A)
    by simp
qed

theorem IL-step-imp-GC-step: M ⇝IL M' ⟹ M ⇝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: chain (⇝IL) Sts and
    act: active-subset (lhd Sts) = {} and
    pas: passive-subset (Liminf-llist Sts) = {}
  shows
    IL-Liminf-saturated: saturated (Liminf-llist Sts) and
    IL-complete-Liminf: B ∈ Bot-F ⟹ fst ‘ lhd Sts ⊨∩𝒢 {B} ⟹
      ∃ BL ∈ Bot-FL. BL ∈ Liminf-llist Sts and
    IL-complete: B ∈ Bot-F ⟹ fst ‘ lhd Sts ⊨∩𝒢 {B} ⟹
      ∃ i. enat i < llength Sts ∧ (∃ BL ∈ Bot-FL. BL ∈ lnth Sts i)
proof −
  have gc-chain: chain (⇝GC) Sts
    using il-chain IL-step-imp-GC-step chain-mono by blast

  show saturated (Liminf-llist Sts)
    using gc.fair-implies-Liminf-saturated gc-chain gc-fair gc-to-red act pas by blast

  {
    assume
```

*bot*: $B \in$ *Bot-F* **and**
*unsat*: *fst* ‘ *lhd Sts* $\models\cap\mathcal{G}$ $\{B\}$

**show** $\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *Liminf-llist Sts*
**by** (*rule gc-complete-Liminf*[*OF gc-chain act pas bot unsat*])
**then show** $\exists$ *i*. *enat i* $<$ *llength Sts* $\wedge$ ($\exists$ *BL* $\in$ *Bot-FL*. *BL* $\in$ *lnth Sts i*)
**unfolding** *Liminf-llist-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** ‹$\leadsto$*ILf*› *50*) **where**
*ol*: *St* $\leadsto$*OLf St'* $\Longrightarrow$ *St* $\leadsto$*ILf St'*
| *red-by-children*: $C \in$ *no-labels.Red-F* (*fset A* $\cup$ *fset M*) $\vee$ *fset M* $= \{C'\} \wedge C' \prec\cdot C \Longrightarrow$
($\{||\}$, *None*, *P*, *Some C*, *A*) $\leadsto$*ILf* (*M*, *None*, *P*, *None*, *A*)

## 9.3 Initial State and Invariant

**lemma** *step-ILf-invariant*:
**assumes** *St* $\leadsto$*ILf St'*
**shows** *OLf-invariant St'*
**using** *assms*
**proof** *cases*
**case** *ol*
**then show** *?thesis*
**using** *step-OLf-invariant* **by** *auto*
**next**
**case** (*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 (↝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 (↝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 ⟷ (∀ St′. ¬ St ↝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 ∀ St′. ¬ St ↝ILf St′
    unfolding st
  proof (intro allI notI)
    fix St′
    assume ({||}, None, empty, None, A) ↝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 $\forall St'. \neg St \leadsto ILf\ St'$
hence $\forall St'. \neg St \leadsto OLf\ St'$
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*: $(N, X, P, Y, A) \leadsto ILf\ (N', X', P', Y', A')$
**shows** *fstate* $(N, X, P, Y, A) \leadsto IL$ *fstate* $(N', X', P', Y', A')$
**using** *ilf*
**proof** *cases*
**case** *ol*
**note** *olf* = *this*(*1*)
**have** *ol*: *fstate* $(N, X, P, Y, A) \leadsto OL$ *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*) $\leadsto IL$ *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*:
$(N, X, P, Y, A) \leadsto ILf\ (N', X', P', Y', A') \Longrightarrow$
*fstate* $(N, X, P, Y, A) \leadsto GC$ *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* ⇒ '*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* ⇒ '*f multiset* ⇒ *bool* (**infix** ‹≺≺*S*› *50*) **where**
(≺≺*S*) ≡ *multp* (≺*S*)

**lemma** *wfP-Precprec-S*: *wfP* (≺≺*S*)
**using** *wfp-multp wfp-Prec-S* **by** *blast*

**definition** *Less1-state* :: ('*p*, '*f*) *OLf-state* ⇒ ('*p*, '*f*) *OLf-state* ⇒ *bool* (**infix** ‹⊏*1*› *50*) **where**
*St'* ⊏*1* *St* ⟷
*mset-of-fstate St'* ≺≺*S* *mset-of-fstate St*
∨ (*mset-of-fstate St'* = *mset-of-fstate St*

∧ (*mset-set* (*fset* (*new-of St'*)) ≺≺*S* *mset-set* (*fset* (*new-of St*))
∨ (*mset-set* (*fset* (*new-of St'*)) = *mset-set* (*fset* (*new-of St*))
∧ *mset-set* (*set-option* (*xx-of St'*)) ≺≺*S* *mset-set* (*set-option* (*xx-of St*)))))

**lemma** *wfP-Less1-state*: *wfP* (⊏*1*)
**proof** −
**let** *?msetset* = {(*M'*, *M*). *M'* ≺≺*S* *M*}
**let** *?triple-of* =
λ*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* <∗*lex*∗> *?msetset* <∗*lex*∗> *?msetset*)
**by** (*rule wf-lex-prod*[*OF wf-msetset wf-lex-prod*[*OF wf-msetset wf-msetset*]])

**have** *Less1-state-alt-def*: ⋀*St' St*. *St'* ⊏*1 St* ⟷
(*?triple-of St'*, *?triple-of St*) ∈ *?msetset* <∗*lex*∗> *?msetset* <∗*lex*∗> *?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* ⇒ (*'p*, *'f*) *OLf-state* ⇒ *bool* (**infix** ‹⊏*2*› *50*) **where**
*St'* ⊏*2 St* ≡
*mset-set* (*set-option* (*yy-of St'*)) ≺≺*S* *mset-set* (*set-option* (*yy-of St*))
∨ (*mset-set* (*set-option* (*yy-of St'*)) = *mset-set* (*set-option* (*yy-of St*))
∧ *St'* ⊏*1 St*)

**lemma** *wfP-Less2-state*: *wfP* (⊏*2*)
**proof** −
**let** *?msetset* = {(*M'*, *M*). *M'* ≺≺*S* *M*}
**let** *?stateset* = {(*St'*, *St*). *St'* ⊏*1 St*}
**let** *?pair-of* = λ*St*. (*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'* ⊏*2 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 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* ⇝*ILf lnth Sts* (*Suc i*)
    **by** (*simp add*: *chain chain-lnth-rel si-lt*)
  **hence** ({||}, *None*, *P*, *Some C*, *A*) ⇝*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 ⇝OLf 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 (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 ≺≺S ?new-bef
    unfolding multp-def
  proof (subst mult-cancelL[OF trans-Prec-S irrefl-Prec-S], fold multp-def)
    show {#C'#} ≺≺S {#C#}
      by (simp add: multp-def prec singletons-in-mult)
  qed
  hence ?old-aft ≺≺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'}) ⊂# 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-multp)
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 ≺≺S ?old-bef
  proof (cases C'' ∈ fset N)
    case c''-in: True

    have mset-set (elems P − {C'}) ⊂# 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-multp*)
**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* ≺≺*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** *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-multp*)
**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 = ∞ and
    full: full-chain (⇝ILf) Sts and
    inv: OLf-invariant (lhd Sts)
  shows Liminf-llist (lmap (set-option ∘ xx-of) Sts) = {}
proof (rule ccontr)
  assume lim-nemp: Liminf-llist (lmap (set-option ∘ xx-of) Sts) ≠ {}

  obtain i :: nat where
    i-lt: enat i < llength Sts and
    inter-nemp: ⋂ ((set-option ∘ xx-of ∘ lnth Sts) ` {j. i ≤ j ∧ enat j < llength Sts}) ≠ {}
    using lim-nemp unfolding Liminf-llist-def by auto
```

**from** *inter-nemp* **obtain** $C :: 'f$ **where**
*c-in*: $\forall P \in$ *lnth Sts* ' $\{j.\ i \leq j \wedge$ *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* $j \leadsto$*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*: ($N$, *Some* $C$, $P$, $Y$, $A$) $\leadsto$*OLf* ($N'$, *Some* $C'$, $P'$, $Y'$, $A'$) (**is** *?bef* $\leadsto$*OLf ?aft*)
**shows** *?aft* $\sqsubset$*1 ?bef*
**proof** $-$
**have** *mset-of-fstate ?aft* $\prec\prec S$ *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* $\leadsto$*OLf St'* **and**
*yy*: *yy-of St* = *None* **and**
*yy'*: *yy-of St'* = *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: mset-of-fstate St′ = mset-of-fstate St
      unfolding defs using c-ni by fastforce
    have new-lt: mset-set (fset (new-of St′)) ≺≺S mset-set (fset (new-of St))
      unfolding defs using c-ni
      by (auto intro!: subset-implies-multp)

    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′ ≺≺S mset-of-fstate St
      unfolding defs by (auto intro: subset-implies-multp)
    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*: *llength Sts* = ∞ and
*full*: *full-chain* (⇝*ILf*) *Sts* and
*inv*: *OLf-invariant* (*lhd Sts*)
shows *Liminf-llist* (*lmap* (*fset* ∘ *new-of*) *Sts*) = {}
proof (*rule ccontr*)
assume *lim-nemp*: *Liminf-llist* (*lmap* (*fset* ∘ *new-of*) *Sts*) ≠ {}

obtain *i* :: *nat* where
*i-lt*: *enat i* < *llength Sts* and
*inter-nemp*: ⋂ ((*fset* ∘ *new-of* ∘ *lnth Sts*) ‘ {*j*. *i* ≤ *j* ∧ *enat j* < *llength Sts*}) ≠ {}
using *lim-nemp* unfolding *Liminf-llist-def* by *auto*

from *inter-nemp* obtain *C* :: *'f* where
*c-in*: ∀ *P* ∈ *lnth Sts* ‘ {*j*. *i* ≤ *j* ∧ *enat j* < *llength Sts*}. *C* ∈ *fset* (*new-of P*)
by *auto*
hence *c-in'*: ∀ *j* ≥ *i*. *enat j* < *llength Sts* ⟶ *C* ∈ *fset* (*new-of* (*lnth Sts j*))
by *auto*

have *si-lt*: *enat* (*Suc i*) < *llength Sts*
by (*simp add*: *len*)

have *new-j*: *new-of* (*lnth Sts j*) ≠ {||} if *j-ge*: *j* ≥ *i* for *j* using *c-in′ len that* **by** *fastforce*

**have** *yy*: *yy-of* (*lnth Sts j*) = *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′*: *yy-of* (*lnth Sts* (*Suc j*)) = *None* **if** *j-ge*: $j \geq i$ **for** *j*
**using** *j-ge* **by** *auto*
**have** *step*: *lnth Sts j* ⇝*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** *lnth Sts* (*Suc j*) ⊏*1 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** $(\sqsubset 1)^{-1-1}$ (*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* $(\sqsubset 1)^{-1-1}$ (*ldropn i Sts*)
**using** *chain-ldropn-lmapI*[*OF - si-lt*, *of - id*, *simplified llist.map-id*] **by** *simp*

**have** *inf-i*: ¬ *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* (⊏*1*)] *wfP-Less1-state*
**by** *blast*
**qed**

**lemma** *yy-empty-OLf-step-imp-Less2-state*:
**assumes** *step*: (*N*, *X*, *P*, *None*, *A*) ⇝*OLf* (*N′*, *X′*, *P′*, *None*, *A′*) (**is** *?bef* ⇝*OLf ?aft*)
**shows** *?aft* ⊏*2 ?bef*
**proof** −
**have** *?aft* ⊏*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* ⇝*OLf St′* **and**
*yy*: *yy-of St′* = *None*
**shows** *St′* ⊏*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* $St'$)) $\prec\prec S$ *mset-set* (*set-option* (*yy-of St*))
    **unfolding** *defs* **by** (*auto intro*!: *subset-implies-multp*)
  **thus** *?thesis*
    **unfolding** *Less2-state-def* **by** *blast*
**qed**

**lemma** *non-choose-p-ILf-step-imp-Less2-state*:
  **assumes**
    *step*: *St* $\leadsto$*ILf* $St'$ **and**
    *yy*: *yy-of* $St'$ = *None*
  **shows** $St'$ $\sqsubset 2$ *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-multp*)
**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* (⇝*ILf*) *Sts* **and**
*init*: *is-initial-OLf-state* (*lhd Sts*)
**shows** *Liminf-llist* (*lmap* (*elems* ∘ *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*:
$\forall j \geq i$. ¬ *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* ⇝*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*: ¬ *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* ($\leadsto$*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*: $B \in$ *Bot-F* $\Longrightarrow$ *fset* (*new-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists B' \in$ *Bot-F*. $B' \in$ *state-union* (*Liminf-fstate Sts*) **and**
*fair-IL-complete*: $B \in$ *Bot-F* $\Longrightarrow$ *fset* (*new-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists i$. *enat* $i <$ *llength Sts* $\wedge$ ($\exists B' \in$ *Bot-F*. $B' \in$ *all-formulas-of* (*lnth Sts i*))
**proof** −
**have** *chain*: *chain* ($\leadsto$*ILf*) *Sts*
**by** (*rule full-chain-imp-chain*[*OF full*])
**have** *il-chain*: *chain* ($\leadsto$*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*: ¬ *lnull Sts*
**using** *chain chain-not-lnull* **by** *blast*
**hence** *lhd-lmap*: $\bigwedge f$. *lhd* (*lmap f Sts*) = *f* (*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** ∀ *St′*. ¬ *llast Sts* ⇝*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*)) ⊨∩$\mathcal{G}$ {*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* (⇝*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* ∈ *Bot-F* ⟹ *fset* (*new-of* (*lhd Sts*)) ⊨∩𝒢 {*B*} ⟹
      ∃ *B′* ∈ *Bot-F*. *B′* ∈ *state-union* (*Liminf-fstate Sts*) **and**
    *fair-OL-complete*: *B* ∈ *Bot-F* ⟹ *fset* (*new-of* (*lhd Sts*)) ⊨∩𝒢 {*B*} ⟹
      ∃ *i*. *enat i* < *llength Sts* ∧ (∃ *B′* ∈ *Bot-F*. *B′* ∈ *all-formulas-of* (*lnth Sts i*))
**proof** −
  **have** *ilf-chain*: *chain* (⇝*ILf*) *Sts*
    **using** *Lazy-List-Chain.chain-mono fair-IL.ol full-chain-imp-chain full* **by** *blast*
  **hence** *ilf-full*: *full-chain* (⇝*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* ∈ *Bot-F* **and**

*unsat*: *fset* (*new-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$

**show** $\exists\, B' \in$ *Bot-F*. $B' \in$ *state-union* (*Liminf-fstate Sts*)
**by** (*rule fair-IL-complete-Liminf*[*OF ilf-full init bot unsat*])
**show** $\exists\, i$. *enat* $i <$ *llength Sts* $\wedge$ ($\exists\, B' \in$ *Bot-F*. $B' \in$ *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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-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* ⇒ *'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**
$\mathcal{G}$*-F-q* :: *'q* ⇒ *'f* ⇒ *'g set* **and**
$\mathcal{G}$*-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*) +
**fixes**
*Prec-S* :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ‹≺S› *50*)
**assumes**
*wfp-Prec-S*: *wfp* (≺S) **and**
*transp-Prec-S*: *transp* (≺S) **and**
*finite-Inf-between*: *finite A* ⟹ *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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-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** *wfp* (≺S)
**by** (*rule wfp-Prec-S*)
**next**
**show** *transp* (≺S)
**by** (*rule transp-Prec-S*)
**next**
**show** $\bigwedge A\ C$. *finite A* ⟹ *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* ⇒ *'f inference set* **where**
  *flat-inferences-of* $T = \bigcup$ {*lset* ι |ι. ι ∈# $T$}

**fun**
  *zl-state* :: *'f inference llist multiset* × *'f inference set* × *'f set* × *'f set* × *'f set* ⇒
    *'f inference set* × (*'f* × *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* $T$ − $D$, (λ$C$. ($C$, *Passive*)) ‘ $P$ ∪ (λ$C$. ($C$, *YY*)) ‘ $Y$ ∪ (λ$C$. ($C$, *Active*)) ‘ $A$)
  **by** *auto*

**inductive**
  *ZL* :: *'f inference set* × (*'f* × *DL-label*) *set* ⇒ *'f inference set* × (*'f* × *DL-label*) *set* ⇒ *bool*
  (**infix** ‹⇝*ZL*› *50*)
**where**
  *compute-infer*: *ι0* ∈ *no-labels.Red-I* ($A$ ∪ {$C$}) ⟹
    *zl-state* ($T$ + {#*LCons ι0 ιs*#}, $D$, $P$, {}, $A$) ⇝*ZL* *zl-state* ($T$ + {#*ιs*#}, $D$ ∪ {*ι0*}, $P$ ∪ {$C$}, {}, $A$)
| *choose-p*: *zl-state* ($T$, $D$, $P$ ∪ {$C$}, {}, $A$) ⇝*ZL* *zl-state* ($T$, $D$, $P$, {$C$}, $A$)
| *delete-fwd*: $C$ ∈ *no-labels.Red-F* $A$ ∨ (∃ $C'$ ∈ $A$. $C'$ ≼· $C$) ⟹
    *zl-state* ($T$, $D$, $P$, {$C$}, $A$) ⇝*ZL* *zl-state* ($T$, $D$, $P$, {}, $A$)
| *simplify-fwd*: $C$ ∈ *no-labels.Red-F* ($A$ ∪ {$C'$}) ⟹
    *zl-state* ($T$, $D$, $P$, {$C$}, $A$) ⇝*ZL* *zl-state* ($T$, $D$, $P$, {$C'$}, $A$)
| *delete-bwd*: $C'$ ∈ *no-labels.Red-F* {$C$} ∨ $C'$ ·≻ $C$ ⟹
    *zl-state* ($T$, $D$, $P$, {$C$}, $A$ ∪ {$C'$}) ⇝*ZL* *zl-state* ($T$, $D$, $P$, {$C$}, $A$)
| *simplify-bwd*: $C'$ ∈ *no-labels.Red-F* {$C$, $C''$} ⟹
    *zl-state* ($T$, $D$, $P$, {$C$}, $A$ ∪ {$C'$}) ⇝*ZL* *zl-state* ($T$, $D$, $P$ ∪ {$C''$}, {$C$}, $A$)
| *schedule-infer*: *flat-inferences-of* $T'$ = *no-labels.Inf-between* $A$ {$C$} ⟹
    *zl-state* ($T$, $D$, $P$, {$C$}, $A$) ⇝*ZL* *zl-state* ($T$ + $T'$, $D$ − *flat-inferences-of* $T'$, $P$, {}, $A$ ∪ {$C$})
| *delete-orphan-infers*: *lset ιs* ∩ *no-labels.Inf-from* $A$ = {} ⟹
    *zl-state* ($T$ + {#*ιs*#}, $D$, $P$, $Y$, $A$) ⇝*ZL* *zl-state* ($T$, $D$ ∪ *lset ιs*, $P$, $Y$, $A$)

## 11.2 Refinement

```
lemma zl-compute-infer-in-lgc:
  assumes ι0 ∈ no-labels.Red-I (A ∪ {C})
  shows zl-state (T + {#LCons ι0 ιs#}, D, P, {}, A) ⇝LGC
    zl-state (T + {#ιs#}, D ∪ {ι0}, P ∪ {C}, {}, A)
proof −
  show ?thesis
  proof (cases ι0 ∈ D)
    case True
    hence infs: flat-inferences-of (T + {#LCons ι0 ιs#}) − D =
      flat-inferences-of (T + {#ιs#}) − (D ∪ {ι0})
      by fastforce
    show ?thesis
      unfolding zl-state.simps infs
      by (rule step.process[of - labeled-formulas-of (P, {}, A) {} - {(C, Passive)}])
        (auto simp: active-subset-def)
  next
    case i0-ni: False
    show ?thesis
      unfolding zl-state.simps
    proof (rule step.compute-infer[of - - ι0 - - {(C, Passive)}])
      show flat-inferences-of (T + {#LCons ι0 ιs#}) − D =
        flat-inferences-of (T + {#ιs#}) − (D ∪ {ι0}) ∪ {ι0}
        using i0-ni by fastforce
    next
      show labeled-formulas-of (P ∪ {C}, {}, A) = labeled-formulas-of (P, {}, A) ∪ {(C, Passive)}
        by auto
    next
      show active-subset {(C, Passive)} = {}
        by (auto simp: active-subset-def)
    next
      show ι0 ∈ no-labels.Red-I-𝒢 (fst ‘ (labeled-formulas-of (P, {}, A) ∪ {(C, 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: zl-state (T, D, P ∪ {C}, {}, A) ⇝LGC zl-state (T, D, P, {C}, A)
proof −
  let ?𝒩 = labeled-formulas-of (P, {}, A)
  and ?𝒯 = flat-inferences-of T − D
  have Passive ⊐L YY
    by (simp add: DL-Prec-L-def)
  hence (?𝒯, ?𝒩 ∪ {(C, Passive)}) ⇝LGC (?𝒯, ?𝒩 ∪ {(C, YY)})
    using relabel-inactive by blast
  hence (?𝒯, labeled-formulas-of (P ∪ {C}, {}, A)) ⇝LGC (?𝒯, 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 ∈ no-labels.Red-F A ∨ (∃ C′ ∈ A. C′ ≼· C)
  shows zl-state (T, D, P, {C}, A) ⇝LGC zl-state (T, D, P, {}, A)
```

**using** *assms*
**proof**
**assume** *c-in*: $C \in$ *no-labels.Red-F A*
**hence** $A \subseteq$ *fst* ‘ *labeled-formulas-of* $(P, \{\}, A)$
**by** *simp*
**hence** $C \in$ *no-labels.Red-F* (*fst* ‘ *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** $\exists\, C' \in A.\ C' \preceq\cdot\ C$
**then obtain** $C'$ **where** *c′-in-and-c′-ls-c*: $C' \in A \wedge C' \preceq\cdot\ C$
**by** *auto*
**hence** ($C'$, *Active*) $\in$ *labeled-formulas-of* $(P, \{\}, A)$
**by** *auto*
**moreover have** *YY* $\sqsupset$*L 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** $C \in$ *no-labels.Red-F-*$\mathcal{G}$ $(A \cup \{C'\})$
**shows** *zl-state* $(T, D, P, \{C\}, A) \leadsto$*LGC zl-state* $(T, D, P, \{C'\}, A)$
**proof** −
**let** *?*$\mathcal{N}$ = *labeled-formulas-of* $(P, \{\}, A)$
**and** *?*$\mathcal{M}$ = $\{(C,\ YY)\}$
**and** *?*$\mathcal{M}'$= $\{(C',\ YY)\}$
**have** $A \cup \{C'\} \subseteq$ *fst* ‘ (*?*$\mathcal{N}$ $\cup$ *?*$\mathcal{M}'$)
**by** *auto*
**hence** $C \in$ *no-labels.Red-F-*$\mathcal{G}$ (*fst*‘ (*?*$\mathcal{N}$ $\cup$ *?*$\mathcal{M}'$))
**by** (*smt* (*verit*, *ccfv-SIG*) *assms no-labels.Red-F-of-subset subset-iff*)
**hence** $(C,\ YY) \in$ *Red-F* (*?*$\mathcal{N}$ $\cup$ *?*$\mathcal{M}'$)
**using** *no-labels-Red-F-imp-Red-F* **by** *simp*
**hence** *?*$\mathcal{M}$ $\subseteq$ *Red-F-*$\mathcal{G}$ (*?*$\mathcal{N}$ $\cup$ *?*$\mathcal{M}'$)
**by** *simp*
**moreover have** *active-subset ?*$\mathcal{M}'$ = $\{\}$
**using** *active-subset-def* **by** *auto*
**ultimately have** (*flat-inferences-of* $T - D$, *labeled-formulas-of* $(P, \{\}, A) \cup \{(C,\ YY)\}) \leadsto$*LGC*
(*flat-inferences-of* $T - D$, *labeled-formulas-of* $(P, \{\}, A) \cup \{(C',\ YY)\}$)
**using** *process*[*of - - ?*$\mathcal{M}$ *- ?*$\mathcal{M}'$] **by** *auto*
**thus** *?thesis*
**by** *simp*
**qed**

**lemma** *zl-delete-bwd-in-lgc*:
**assumes** $C' \in$ *no-labels.Red-F-*$\mathcal{G}$ $\{C\} \vee C' \cdot\!\succ C$
**shows** *zl-state* $(T, D, P, \{C\}, A \cup \{C'\}) \leadsto$*LGC zl-state* $(T, D, P, \{C\}, A)$
**using** *assms*
**proof**
**let** *?*$\mathcal{N}$ = *labeled-formulas-of* $(P, \{C\}, A)$
**assume** *c′-in*: $C' \in$ *no-labels.Red-F-*$\mathcal{G}$ $\{C\}$

**have** $\{C\} \subseteq$ *fst* ‘ *?*$\mathcal{N}$
**by** *simp*

```
  hence C′ ∈ no-labels.Red-F-𝒢 (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 ∪ {(C′, Active)}) ⇝LGC (flat-inferences-of T − D, ?𝒩)
    using remove-redundant-no-label by auto

  moreover have ?𝒩 ∪ {(C′, Active)} = labeled-formulas-of (P, {C}, A ∪ {C′})
    using PYA-add-active-formula by blast
  ultimately have (flat-inferences-of T − D, labeled-formulas-of (P, {C}, A ∪ {C′})) ⇝LGC
    zl-state (T, D, P, {C}, A)
    by simp
  thus ?thesis
    by auto
next
  assume C′ ·≻ C
  moreover have (C, YY) ∈ 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′ ∈ no-labels.Red-F-𝒢 {C, C′′}
  shows zl-state (T, D, P, {C}, A ∪ {C′}) ⇝LGC zl-state (T, D, P ∪ {C′′}, {C}, A)
proof −
  let ?ℳ = {(C′, Active)}
  and ?ℳ′ = {(C′′, Passive)}
  and ?𝒩 = labeled-formulas-of (P, {C}, A)
  have {C, C′′} ⊆ fst‘ (?𝒩 ∪ ?ℳ′)
    by simp
  hence C′ ∈ no-labels.Red-F-𝒢 (fst‘ (?𝒩 ∪ ?ℳ′))
    by (smt (z3) DiffI Diff-eq-empty-iff assms empty-iff no-labels.Red-F-of-subset)
  hence ℳ-included: ?ℳ ⊆ Red-F-𝒢 (?𝒩 ∪ ?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 ∪ ?ℳ) ⇝LGC (flat-inferences-of T − D, ?𝒩 ∪ ?ℳ′)
    using ℳ-included process[of - - ?ℳ - ?ℳ′] by auto
  moreover have ?𝒩 ∪ ?ℳ = labeled-formulas-of(P, {C}, A ∪ {C′}) and
    ?𝒩 ∪ ?ℳ′ = labeled-formulas-of(P ∪ {C′′}, {C}, A)
    by auto
  ultimately show ?thesis
    by auto
qed

lemma zl-schedule-infer-in-lgc:
  assumes flat-inferences-of T′ = no-labels.Inf-between A {C}
  shows zl-state (T, D, P, {C}, A) ⇝LGC
    zl-state (T + T′, D − flat-inferences-of T′, P, {}, A ∪ {C})
proof −
  let ?𝒩 = labeled-formulas-of (P, {}, A)
  have fst ‘ active-subset ?𝒩 = A
    by (meson prj-active-subset-of-state)
  hence infs: flat-inferences-of T′ = no-labels.Inf-between (fst ‘ active-subset ?𝒩) {C}
    using assms by simp
```

**have** *inf*: (*flat-inferences-of* $T - D$, *?𝒩* ∪ {(*C*, *YY*)}) ⇝*LGC*
((*flat-inferences-of* $T - D$) ∪ *flat-inferences-of* $T'$, *?𝒩* ∪ {(*C*, *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* ∪ {(*C*, *YY*)}
**by** *auto*
**have** *t-aft*: *flat-inferences-of* ($T + T'$) − (*D* − *flat-inferences-of* $T'$) =
(*flat-inferences-of* $T - D$) ∪ *flat-inferences-of* $T'$
**by** *auto*
**have** *m-aft*: *labeled-formulas-of* (*P*, {}, *A* ∪ {*C*}) = *?𝒩* ∪ {(*C*, *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*: *lset ιs* ∩ *no-labels.Inf-from A* = {}
**shows** *zl-state* (*T* + {#*ιs*#}, *D*, *P*, *Y*, *A*) ⇝*LGC zl-state* (*T*, *D* ∪ *lset ιs*, *P*, *Y*, *A*)
**proof** −
**let** *?𝒩* = *labeled-formulas-of* (*P*, *Y*, *A*)

**have** *inf*: (*flat-inferences-of T* ∪ *lset ιs* − *D*, *?𝒩*)
⇝*LGC* (*flat-inferences-of T* − (*D* ∪ *lset ιs*), *?𝒩*)
**by** (*rule step.delete-orphan-infers*[*of* - - *lset ιs* − *D*])
(*use inter prj-active-subset-of-state* **in** *auto*)

**have** *t-bef*: *flat-inferences-of* (*T* + {#*ιs*#}) − *D* = *flat-inferences-of T* ∪ *lset ιs* − *D*
**by** *auto*
**show** *?thesis*
**unfolding** *zl-state.simps t-bef* **using** *inf* **.**
**qed**

**theorem** *ZL-step-imp-LGC-step*: *St* ⇝*ZL* $St'$ ⟹ *St* ⇝*LGC* $St'$
**proof** (*induction rule*: *ZL.induct*)
**case** (*compute-infer ι0 A C T ιs 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 ιs A T D P Y*)
  **thus** *?case*
    **using** *zl-delete-orphan-infers-in-lgc* **by** *auto*
**qed**

## 11.3 Completeness

**theorem**
  **assumes**
    *zl-chain*: *chain* (↝*ZL*) *Sts* **and**
    *act*: *active-subset* (*snd* (*lhd Sts*)) = {} **and**
    *pas*: *passive-subset* (*Liminf-llist* (*lmap snd Sts*)) = {} **and**
    *no-prems-init*: ∀ *ι* ∈ *Inf-F*. *prems-of ι* = [] ⟶ *ι* ∈ *fst* (*lhd Sts*) **and**
    *final-sched*: *Liminf-llist* (*lmap fst Sts*) = {}
  **shows**
    *ZL-Liminf-saturated*: *saturated* (*Liminf-llist* (*lmap snd Sts*)) **and**
    *ZL-complete-Liminf*: *B* ∈ *Bot-F* ⟹ *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*B*} ⟹
      ∃ *BL* ∈ *Bot-FL*. *BL* ∈ *Liminf-llist* (*lmap snd Sts*) **and**
    *ZL-complete*: *B* ∈ *Bot-F* ⟹ *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*B*} ⟹
      ∃ *i*. *enat i* < *llength Sts* ∧ (∃ *BL* ∈ *Bot-FL*. *BL* ∈ *snd* (*lnth Sts i*))
**proof** −
  **have** *lgc-chain*: *chain* (↝*LGC*) *Sts*
    **using** *zl-chain ZL-step-imp-LGC-step chain-mono* **by** *blast*

  **show** *saturated* (*Liminf-llist* (*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* ∈ *Bot-F* **and**
      *unsat*: *fst* ‘ *snd* (*lhd Sts*) ⊨∩𝒢 {*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-xxs = 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-xxs z-ne-y)
  next
    case z-ne-x: False

    have z-in-take-xs: z ∈ set (take m xs)
      using z-in-take-xxs 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-xxs)
      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-xxs 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* ∪ *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* × *'e set* ⇒ *'e* ⇒ *'e llist* ⇒ *'q* × *'e set* ⇒ *bool* **where**
  *pick-lqueue-step-w-detailsI*: *LCons e es* ∈# *llists Q* ⟹ *fst* (*pick-elem Q*) = *e* ⟹
    *llists* (*snd* (*pick-elem Q*)) = *llists Q* − {#*LCons e es*#} + {#*es*#} ⟹
    *pick-lqueue-step-w-details* (*Q*, *D*) *e es* (*snd* (*pick-elem Q*), *D* ∪ {*e*})

**inductive** *pick-lqueue-step* :: *'q* × *'e set* ⇒ *'q* × *'e set* ⇒ *bool* **where**
  *pick-lqueue-stepI*: *pick-lqueue-step-w-details QD e es QD'* ⟹ *pick-lqueue-step QD QD'*

**inductive**
  *remove-lqueue-step-w-details* :: *'q* × *'e set* ⇒ *'e llist list* ⇒ *'q* × *'e set* ⇒ *bool*
**where**

*remove-lqueue-step-w-detailsI*:
*remove-lqueue-step-w-details* (*Q*, *D*) *ess*
(*fold remove-llist ess Q*, *D* ∪ ⋃ {*lset es* |*es*. *es* ∈ *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* ⇒ *'q* ⇒ *'q* **and**
*remove-llist* :: *'e llist* ⇒ *'q* ⇒ *'q* **and**
*pick-elem* :: *'q* ⇒ *'e* × *'q* **and**
*llists* :: *'q* ⇒ *'e llist multiset* +
**assumes** *fair*: *chain lqueue-step QDs* ⟹ *infinitely-often pick-lqueue-step QDs* ⟹
*LCons e es* ∈# *llists* (*fst* (*lnth QDs i*)) ⟹
∃ *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*))
**begin**

**lemma** *fair-strong*:
**assumes**
*chain*: *chain lqueue-step QDs* **and**
*inf*: *infinitely-often pick-lqueue-step QDs* **and**
*es-in*: *es* ∈# *llists* (*fst* (*lnth QDs i*)) **and**
*k-lt*: *enat k* < *llength es*
**shows** ∃ *j* ≥ *i*.
(∃ *k'* ≤ *k*. ∃ *ess*. *ldrop 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* (*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 ⇒ 'e fifo ⇒ '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 ⇒ 'e fifo ⇒ '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 ⇒ 'e × 'e fifo where
  pick-elem (-, []) = undefined
| pick-elem (num-nils, (e, es) # ps) =
   (e,
    (case es of
      LNil ⇒ (num-nils + 1, ps)
    | LCons e' es' ⇒ (num-nils, ps @ [(e', es')])))

fun llists :: 'e fifo ⇒ '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 ≠ empty
  thus llists Q ≠ {#}
  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) ∈# mset ps#} ∩# {#LNil#} = {#}
        by auto
      show ?thesis
      proof (cases num-nils)
        case num-nils: 0
        have nil-ni: LNil ∉# {#LCons x y. (x, y) ∈# 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

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: ¬ (∃j ≥ i. ∃ess. LCons e es ∈ set ess
    ∧ remove-lqueue-step-w-details (lnth QDs j) ess (lnth QDs (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 ∀k' ≤ k. ∃i' ≥ i. (e, es) ∈ set (take (Suc k') (snd (fst (lnth QDs i'))))
  proof −
    have ∃i' ≥ i. (e, es) ∈ set (take (k + 1 − l) (snd (fst (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) ∈ set (take (Suc k) (snd (fst (lnth QDs i))))
          by (simp add: at-k k-lt take-Suc-conv-app-nth)
      qed
    next
      case (Suc l)
      note ih = this(1) and sl-le = this(2)

      have l-le-k: l ≤ k
        using sl-le by linarith
      note ih = ih[OF l-le-k]

      obtain i' :: nat where
        i'-ge: i' ≥ i and
        cons-at-i': (e, es) ∈ set (take (k + 1 − l) (snd (fst (lnth QDs i'))))
        using ih by blast
      then obtain j0 :: nat where
        j0 ≥ i' and
        pick-lqueue-step (lnth QDs j0) (lnth QDs (Suc j0))
        using inf-pick unfolding infinitely-often-alt-def by auto
      then obtain j :: nat where
        j-ge: j ≥ i' and
```

*pick-step*: *pick-lqueue-step* (*lnth QDs j*) (*lnth QDs* (*Suc j*)) **and**
*pick-step-min*:
$\forall j'.\ j' \geq i' \longrightarrow j' < j \longrightarrow \neg$ *pick-lqueue-step* (*lnth QDs j'*) (*lnth QDs* (*Suc j'*))
**using** *wfP-exists-minimal*[*OF wfp-on-less*, *of*
$\lambda j.\ j \geq i' \wedge$ *pick-lqueue-step* (*lnth QDs j*) (*lnth QDs* (*Suc j*)) *j0* $\lambda j.\ j$]
**by** *blast*

**have** *cons-at-le-j*: (*e*, *es*) $\in$ *set* (*take* (*k* + *1* − *l*) (*snd* (*fst* (*lnth QDs j'*))))
**if** *j'-ge*: $j' \geq i'$ **and** *j'-le*: $j' \leq j$ **for** *j'*
**proof** −
**have** (*e*, *es*) $\in$ *set* (*take* (*k* + *1* − *l*) (*snd* (*fst* (*lnth 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** *cons-at-i'* **by** *fastforce*
**next**
**case** (*Suc m*)
**note** *ih* = *this*(*1*) **and** *i'sm-le* = *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** *step*: *lqueue-step* (*lnth QDs* (*i'* + *m*)) (*lnth QDs* (*i'* + *Suc m*))
**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** *in-set-fold-add*: (*e*, *es*) $\in$ *set* (*take n* (*snd* (*fold add-llist ess Q*)))
**if** (*e*, *es*) $\in$ *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*) $\in$ *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 ∃ e es. 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 ¬ 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 ≤ j
      using i'-ge j-ge le-trans by blast
  next
    show (e, es) ∈ 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 ≤ Suc j
      using i'-ge j-ge by linarith
  next
    obtain Q :: 'e fifo and D :: 'e set and e' :: 'e and es' :: 'e llist where
      at-j: lnth QDs j = (Q, D) and
      at-sj: lnth QDs (Suc j) = (snd (pick-elem Q), D ∪ {e'}) and
      pair-in: LCons e' es' ∈# 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) ∈ set (take (k + 1 − l) (snd (fst (lnth QDs j))))
      using cons-at-le-j[of j] j-ge by blast

    show (e, es) ∈ set (take (k + 1 − Suc l) (snd (fst (lnth QDs (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: hd (snd (fst (lnth 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
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′ ≥ i and
  cons-at-i′: (e, es) ∈ set (take 1 (snd (fst (lnth QDs i′))))
  by auto
then obtain j0 :: nat where
  j0 ≥ i′ and
  pick-lqueue-step (lnth QDs j0) (lnth QDs (Suc j0))
  using inf-pick unfolding infinitely-often-alt-def by auto
then obtain j :: nat where
  j-ge: j ≥ i′ and
  pick-step: pick-lqueue-step (lnth QDs j) (lnth QDs (Suc j)) and
  pick-step-min:
    ∀ j′. j′ ≥ i′ ⟶ j′ < j ⟶ ¬ pick-lqueue-step (lnth QDs j′) (lnth QDs (Suc j′))
  using wfP-exists-minimal[OF wfp-on-less, of
    λj. j ≥ i′ ∧ pick-lqueue-step (lnth QDs j) (lnth QDs (Suc j)) j0 λj. j]
  by blast
hence pick-step-det: ∃ e es. pick-lqueue-step-w-details (lnth QDs j) e es (lnth QDs (Suc j))
  unfolding pick-lqueue-step.simps by simp
have pick-lqueue-step-w-details (lnth QDs j) e es (lnth QDs (Suc j))
proof −
  have cons-at-j: (e, es) ∈ set (take 1 (snd (fst (lnth QDs j))))
  proof −
    have (e, es) ∈ set (take 1 (snd (fst (lnth QDs (i′ + l))))) if i′l-le: i′ + l ≤ j for l
      using i′l-le
    proof (induct l)
      case (Suc l)
      note ih = this(1) and i′sl-le = this(2)

      have i′l-lt: i′ + l < j
        using i′sl-le by linarith
      have i′l-le: i′ + l ≤ j
        using i′sl-le by linarith
      note ih = ih[OF i′l-le]

      have step: lqueue-step (lnth QDs (i′ + l)) (lnth QDs (i′ + 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*: *length* (*snd Q*) ≥ *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*: *take* (*Suc 0*) (*snd* (*fold add-llist ess Q*)) = *take* (*Suc 0*) (*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* = *this*(*1*) **and** *len-q* = *this*(*2*)

    **have** *len-add*: *length* (*snd* (*add-llist es′ Q*)) ≥ *1*
    **proof** (*cases Q*)
      **case** *q*: (*Pair num-nils ps*)
      **show** *?thesis*
      **proof** (*cases es′*)
        **case** *es′*: *LNil*
        **show** *?thesis*
          **using** *len-q* **unfolding** *q es′* **by** *simp*
      **next**
        **case** *es′*: (*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** ¬ (∃ *ess*. *LCons e es* ∈ *set ess*
    ∧ *remove-lqueue-step-w-details* (*lnth QDs* (*i′* + *l*)) *ess* (*lnth QDs* (*i′* + *Suc l*)))

    **using** *not-rem-step add-Suc-right i′-ge trans-le-add1* **by** *presburger*
  **ultimately have** *ees-ni*: *LCons e es* ∉ *set ess*
    **by** *blast*

  **obtain** *ps′* :: (*′e* × *′e llist*) *list* **where**
    *snd-q*: *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* = (*num-nils′*, (*e*, *es*) # *ps′*)
    **by** (*metis prod.collapse snd-q*)

  **have** *take-1*: *take 1* (*snd* (*fold remove-llist ess Q*)) = *take 1* (*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* ∉ *set ess′*
      **using** *ees-ni* **by** *simp*
    **note** *ih* = *ih*[*OF ees-ni′*]

    **have** *es′-ne*: *es′* ≠ *LCons e es*
      **using** *ees-ni* **by** *auto*

    **show** *?case*
    **proof** (*cases es′*)
      **case** *LNil*
      **then show** *?thesis*
        **using** *ih* **by** *auto*
    **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-elemI 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** ∃ *e es. 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-elem lqueue-step-pick-elemI*(*2*) *rest*)
    **thus** *?thesis*
      **using** *pick-lqueue-stepI* **by** *fast*

```
        qed
        moreover have ¬ 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) ∈ set (snd (fst (lnth QDs j)))
    using in-set-takeD by force

  obtain ps' :: ('e × 'e llist) list where
    fst-at-j: 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: fst (pick-elem (fst (lnth QDs j))) = e
    using fst-at-j by (metis fst-conv pick-elem.simps(2) surjective-pairing)
  have snd-pick: llists (snd (pick-elem (fst (lnth QDs j)))) =
    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: 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 lnth QDs (Suc j) = (snd (pick-elem Q), D ∪ {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-elem Q) = e
      using at-j fst-pick by force
  next
    show llists (snd (pick-elem 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**

# 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* $= 't \times 'f$ *inference set* $\times\ 'p \times 'f$ *option* $\times\ 'f$ *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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-I-q Equiv-F Prec-F* +
  *todo*: *fair-prover-lazy-list-queue t-empty t-add-llist t-remove-llist t-pick-elem 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* :: $'q \Rightarrow 'g$ *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* **and**
    *Equiv-F* :: $'f \Rightarrow 'f \Rightarrow$ *bool* (**infix** ‹$\doteq$› *50*) **and**
    *Prec-F* :: $'f \Rightarrow 'f \Rightarrow$ *bool* (**infix** ‹$\prec\cdot$› *50*) **and**
    *t-empty* :: *'t* **and**
    *t-add-llist* :: *'f inference llist* $\Rightarrow 't \Rightarrow 't$ **and**
    *t-remove-llist* :: *'f inference llist* $\Rightarrow 't \Rightarrow 't$ **and**
    *t-pick-elem* :: $'t \Rightarrow 'f$ *inference* $\times\ 't$ **and**
    *t-llists* :: $'t \Rightarrow 'f$ *inference llist multiset* **and**
    *p-empty* :: *'p* **and**
    *p-select* :: $'p \Rightarrow 'f$ **and**
    *p-add* :: $'f \Rightarrow 'p \Rightarrow 'p$ **and**
    *p-remove* :: $'f \Rightarrow 'p \Rightarrow 'p$ **and**
    *p-felems* :: $'p \Rightarrow 'f$ *fset* +
  **fixes**
    *Prec-S* :: $'f \Rightarrow 'f \Rightarrow$ *bool* (**infix** ‹$\prec S$› *50*)
  **assumes**
    *wfp-Prec-S*: *wfp* $(\prec S)$ **and**
    *transp-Prec-S*: *transp* $(\prec S)$ **and**
    *countable-Inf-between*: *finite* $A \Longrightarrow$ *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** *wfp-imp-irreflp wfp-Prec-S* **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-zl-fstate* :: (′*t*, ′*p*, ′*f*) *ZLf-state llist* ⇒ ′*f set* × ′*f set* × ′*f set*
**where**
*Liminf-zl-fstate Sts* =
(*Liminf-llist* (*lmap* (*passive.elems* ∘ *passive-of*) *Sts*),
*Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*),
*Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*))

**lemma** *Liminf-zl-fstate-commute*:
*Liminf-llist* (*lmap* (*snd* ∘ *zl-fstate*) *Sts*) = *labeled-formulas-of* (*Liminf-zl-fstate Sts*)
**proof** −
**have** *Liminf-llist* (*lmap* (*snd* ∘ *zl-fstate*) *Sts*) =
(*λC*. (*C*, *Passive*)) ‘ *Liminf-llist* (*lmap* (*passive.elems* ∘ *passive-of*) *Sts*) ∪
(*λC*. (*C*, *YY*)) ‘ *Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*) ∪
(*λC*. (*C*, *Active*)) ‘ *Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*)
**unfolding** *zl-fstate-alt-def zl-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-zl-fstate-def* **by** *fastforce*
**qed**

**fun** *formulas-union* :: *'f set* × *'f set* × *'f set* ⇒ *'f set* **where**
*formulas-union* (*P*, *Y*, *A*) = *P* ∪ *Y* ∪ *A*

**inductive**
*fair-ZL* :: (*'t*, *'p*, *'f*) *ZLf-state* ⇒ (*'t*, *'p*, *'f*) *ZLf-state* ⇒ *bool* (**infix** ‹⇝*ZLf*› *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*, *D*, *P*, *None*, *A*) ⇝*ZLf* (*T'*, *D* ∪ {*ι0*}, *p-add C P*, *None*, *A*)
| *choose-p*: *P* ≠ *p-empty* ⟹
(*T*, *D*, *P*, *None*, *A*) ⇝*ZLf* (*T*, *D*, *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*, *D*, *P*, *Some C*, *A*) ⇝*ZLf* (*T*, *D*, *P*, *None*, *A*)
| *simplify-fwd*: *C'* ≺*S C* ⟹ *C* ∈ *no-labels.Red-F* (*fset A* ∪ {*C'*}) ⟹
(*T*, *D*, *P*, *Some C*, *A*) ⇝*ZLf* (*T*, *D*, *P*, *Some C'*, *A*)
| *delete-bwd*: *C'* |∉| *A* ⟹ *C'* ∈ *no-labels.Red-F* {*C*} ∨ *C'* ·≻ *C* ⟹
(*T*, *D*, *P*, *Some C*, *A* |∪| {|*C'*|}) ⇝*ZLf* (*T*, *D*, *P*, *Some C*, *A*)
| *simplify-bwd*: *C'* |∉| *A* ⟹ *C''* ≺*S C'* ⟹ *C'* ∈ *no-labels.Red-F* {*C*, *C''*} ⟹
(*T*, *D*, *P*, *Some C*, *A* |∪| {|*C'*|}) ⇝*ZLf* (*T*, *D*, *p-add C'' P*, *Some C*, *A*)
| *schedule-infer*: *flat-inferences-of* (*mset ιss*) = *no-labels.Inf-between* (*fset A*) {*C*} ⟹
(*T*, *D*, *P*, *Some C*, *A*) ⇝*ZLf*
(*fold t-add-llist ιss T*, *D* − *flat-inferences-of* (*mset ιss*), *P*, *None*, *A* |∪| {|*C*|})
| *delete-orphan-infers*: *ιs* ∈# *t-llists T* ⟹ *lset ιs* ∩ *no-labels.Inf-from* (*fset A*) = {} ⟹
(*T*, *D*, *P*, *Y*, *A*) ⇝*ZLf* (*t-remove-llist ιs T*, *D* ∪ *lset ιs*, *P*, *Y*, *A*)

**inductive** *compute-infer-step* :: (*'t*, *'p*, *'f*) *ZLf-state* ⇒ (*'t*, *'p*, *'f*) *ZLf-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*, *D*, *P*, *None*, *A*) (*T'*, *D* ∪ {*ι0*}, *p-add C P*, *None*, *A*)

The step below is slightly more general than the corresponding step in (⇝*ZLf*), 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* ⇒ (*'t*, *'p*, *'f*) *ZLf-state* ⇒ *bool* **where**
*P* ≠ *p-empty* ⟹
*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* ⇒ *bool* **where**
*flat-inferences-of* (*mset ιss*) = *no-labels.Inf-from* {} ⟹
*is-initial-ZLf-state* (*fold t-add-llist ιss t-empty*, {}, *p-empty*, *None*, {||})

**inductive** *ZLf-invariant* :: (*'t*, *'p*, *'f*) *ZLf-state* ⇒ *bool* **where**
*flat-inferences-of* (*t-llists T*) ⊆ *Inf-F* ⟹ *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** *ιss*

**assume**
*st*: *St* = (*fold t-add-llist ιss t-empty*, {}, *p-empty*, *None*, {||}) **and**
*ιss*: *flat-inferences-of* (*mset ιss*) = *no-labels.Inf-from* {}

**have** *flat-inferences-of* (*t-llists* (*fold t-add-llist ιss t-empty*)) ⊆ *Inf-F*
**using** *ιss 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* ⇝*ZLf St'*
**shows** *ZLf-invariant St'*
**using** *step inv*
**proof** *cases*
**case** (*compute-infer T ι0 T' A C D P*)
**note** *defs* = *this*(*1*,*2*) **and** *has-el* = *this*(*3*) **and** *pick* = *this*(*4*)

**have** *t'*: *T'* = *snd* (*t-pick-elem T*)
**using** *pick* **by** *simp*

**obtain** *ιs'* **where**
*ι0ιs'-in*: *LCons ι0 ιs'* ∈# *t-llists T* **and**
*lists-t'*: *t-llists T'* = *t-llists T* − {#*LCons ι0 ιs'*#} + {#*ιs'*#}
**using** *todo.llists-pick-elem*[*OF has-el*, *folded t'*] *pick* **by** *auto*

**let** *?II* = {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T*}
**let** *?I* = ⋃ *?II*

**have** ⋃ {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T* − {#*LCons ι0 ιs'*#} + {#*ιs'*#}} =
(⋃ {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T* − {#*LCons ι0 ιs'*#}}) ∪ *lset ιs'*
**by** *auto*
**also have** ... ⊆ (⋃ {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T* − {#*LCons ι0 ιs'*#}}) ∪ {*ι0*} ∪ *lset ιs'*
**unfolding** *lists-t'*
**by** *auto*
**also have** ... ⊆ *?I* ∪ {*ι0*} ∪ *lset ιs'*
**proof** −
**have** ⋃ {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T* − {#*LCons ι0 ιs'*#}} ⊆ ⋃ {*lset ιs* |*ιs*. *ιs* ∈# *t-llists T*}
**using** *Union-Setcompr-member-mset-mono*[*of t-llists T* − {#*LCons ι0 ιs'*#} *t-llists T lset*]
**by** *auto*
**thus** *?thesis*
**by** *blast*
**qed**
**also have** ... ⊆ *?I*
**proof** −
**have** *ι0* ∈ *?I*
**using** *todo.llists-pick-elem*[*OF has-el*, *folded t'*] *pick* **by** *auto*
**moreover have** *lset ιs'* ⊆ *?I*
**using** *todo.llists-pick-elem*[*OF has-el*, *folded t'*] *pick ι0ιs'-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 ιss A C T D P)
  note defs = this(1,2) and ιss-inf-betw = this(3)
  have ⋃ {lset ι |ι. ι ∈ set ιss} ⊆ Inf-F
    using ιss-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 ιs T A D P Y)
  note defs = this(1,2)
  have ⋃ {lset ι |ι. ι ∈# t-llists T − {#ιs#}} ⊆ ⋃ {lset ι |ι. ι ∈# t-llists T}
    using Union-Setcompr-member-mset-mono[of t-llists T − {#ιs#} 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 (↝ZLf) Sts and
    fair-hd: ZLf-invariant (lhd Sts) and
    i-lt: enat i < 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 = 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: ZLf-invariant (lnth Sts i)
    by (rule ih)
  have step: lnth Sts i ↝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 (↝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* ⇒ *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* ⟷ (∀ *St'*. ¬ *St* ⇝*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** ∀ *St'*. ¬ *St* ⇝*ZLf St'*
**unfolding** *st*
**proof** (*intro allI notI*)
**fix** *St'*
**assume** (*t-empty*, *D*, *p-empty*, *None*, *A*) ⇝*ZLf St'*
**thus** *False*
**by** *cases auto*
**qed**
**next**
**assume** *no-step*: ∀ *St'*. ¬ *St* ⇝*ZLf St'*
**show** *is-final-ZLf-state St*
**proof** (*rule ccontr*)
**assume** *not-fin*: ¬ *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* ≠ *t-empty* ∨ *P* ≠ *p-empty* ∨ *Y* ≠ *None*
**using** *not-fin* **unfolding** *st is-final-ZLf-state.simps* **by** *auto*
**moreover {**
**assume**
*t*: *T* ≠ *t-empty* **and**
*y*: *Y* = *None*

**have** ∃ *St'*. *St* ⇝*ZLf St'*
**proof** (*cases todo.has-elem T*)
**case** *has-el*: *True*

**obtain** *ι0* :: *'f inference* **and** *T'* :: *'t* **where**
*pick*: *t-pick-elem T* = (*ι0*, *T'*)
**by** *fastforce*

**obtain** $\iota s'$ **where**
$\iota 0 \iota s'$*-in*: *LCons* $\iota 0$ $\iota s'$ $\in\#$ *t-llists* $T$ **and**
*lists-t*$'$: *t-llists* $T'$ = *t-llists* $T$ − $\{\#LCons\ \iota 0\ \iota s'\#\}$ + $\{\#\iota s'\#\}$
**using** *todo.llists-pick-elem*[*OF has-el*] *pick* **by** *auto*

**have** $\iota 0 \in \bigcup$ $\{$*lset* $\iota$ $|\iota.\ \iota \in\#$ *t-llists* $T\}$
**using** $\iota 0 \iota s'$*-in* **by** *auto*
**hence** $\iota 0 \in$ *Inf-F*
**using** *inv t* **unfolding** *st ZLf-invariant.simps* **by** *auto*
**hence** $\iota 0$*-red*: $\iota 0 \in$ *no-labels.Red-I-*$\mathcal{G}$ (*fset A* $\cup$ $\{$*concl-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-in*: *LNil* $\in\#$ *t-llists* $T$
**by** (*metis has-no-el multiset-nonemptyE t todo.llists-not-empty*)
**have** *nil-inter*: *lset LNil* $\cap$ *no-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-empty* **and**
*y*: $Y$ = *None*

**have** $\exists\, St'.\ St \leadsto ZLf\ St'$
**using** *fair-ZL.choose-p*[*OF p*] **unfolding** *st p y* **by** *fast*
**}**
**moreover**
**{**
**assume** $Y \neq$ *None*
**then obtain** $C$ :: $'f$ **where**
*y*: $Y$ = *Some C*
**by** *blast*

**obtain** $\iota s$ :: $'f$ *inference llist* **where**
$\iota ss$: *flat-inferences-of* (*mset* $[\iota s]$) = *no-labels.Inf-between* (*fset A*) $\{C\}$
**using** *countable-imp-lset*[*OF countable-Inf-between*[*OF finite-fset*]] **by** *force*

**have** $\exists\, St'.\ St \leadsto ZLf\ St'$
**using** *fair-ZL.schedule-infer*[*OF* $\iota ss$] **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*: $(T, D, P, Y, A) \leadsto ZLf\ (T', D', P', Y', A')$
**shows** *zl-fstate* $(T, D, P, Y, A) \leadsto ZL$ *zl-fstate* $(T', D', P', Y', A')$
**using** *zlf*
**proof** *cases*
**case** (*compute-infer* $\iota 0$ *C*)
**note** *defs* = *this*(*1*−*5*) **and** *has-el* = *this*(*6*) **and** *pick* = *this*(*7*) **and** *ι-red* = *this*(*8*)

**obtain** $\iota s'$ **where**
*ι0ιs′-in*: *LCons ι0 ιs′* ∈# *t-llists T* **and**
*lists-t′*: *t-llists T′* = *t-llists T* − {#*LCons ι0 ιs′*#} + {#*ιs′*#}
**using** *todo.llists-pick-elem*[*OF has-el*] *pick* **by** *auto*

**show** *?thesis*
**unfolding** *defs zl-fstate-alt-def prod.sel option.set lists-t′*
**using** *ZL.compute-infer*[*OF ι-red*, *of t-llists T* − {#*LCons ι0 ιs′*#} *ιs′ D passive.elems P*]
*ι0ιs′-in*
**by** *auto*
**next**
**case** *choose-p*
**note** *defs* = *this*(*1*−*6*) **and** *p-nemp* = *this*(*7*)

**have** *elems-rem-sel-uni-sel*:
*passive.elems* (*p-remove* (*p-select P*) *P*) ∪ {*p-select P*} = *passive.elems P*
**using** *p-nemp* **by** *force*

**show** *?thesis*
**unfolding** *defs zl-fstate-alt-def prod.sel option.set*
**using** *ZL.choose-p*[*of t-llists T D passive.elems* (*p-remove* (*p-select P*) *P*) *p-select P*
*fset A*]
**by** (*metis elems-rem-sel-uni-sel*)
**next**
**case** (*delete-fwd C*)
**note** *defs* = *this*(*1*−*6*) **and** *c-red* = *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 ιss C*)
**note** *defs* = *this*(*1*−*6*) **and** *ιss* = *this*(*7*)
**show** *?thesis*
**unfolding** *defs zl-fstate-alt-def prod.sel option.set*

**using** *ZL.schedule-infer*[*OF* *ιss*, *of* *t-llists* *T* *D* *passive.elems* *P*]
**by** (*simp* *add*: *Un-commute*)
**next**
**case** (*delete-orphan-infers* *ιs*)
**note** *defs* = *this*(*1*−*5*) **and** *ιs-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* − {#*ιs*#} *D* *passive.elems* *P* *set-option* *Y*]
*ιs-in*
**by** *simp*
**qed**

**lemma** *fair-ZL-step-imp-GC-step*:
(*T*, *D*, *P*, *Y*, *A*) ⇝*ZLf* (*T*′, *D*′, *P*′, *Y*′, *A*′) ⟹
*zl-fstate* (*T*, *D*, *P*, *Y*, *A*) ⇝*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* ⇒ ′*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* ⇒ ′*f* *multiset* ⇒ *bool* (**infix** ‹≺≺*S*› *50*) **where**
(≺≺*S*) ≡ *multp* (≺*S*)

**lemma** *wfP-Precprec-S*: *wfP* (≺≺*S*)
**using** *wfp-multp* *wfp-Prec-S* **by** *blast*

**definition** *Less-state* :: (′*t*, ′*p*, ′*f*) *ZLf-state* ⇒ (′*t*, ′*p*, ′*f*) *ZLf-state* ⇒ *bool* (**infix** ‹⊏› *50*)
**where**
*St*′ ⊏ *St* ⟷
*mset-of-zl-fstate* *St*′ ≺≺*S* *mset-of-zl-fstate* *St*
∨ (*mset-of-zl-fstate* *St*′ = *mset-of-zl-fstate* *St*
∧ (*mset-set* (*passive.elems* (*passive-of* *St*′)) ≺≺*S* *mset-set* (*passive.elems* (*passive-of* *St*))
∨ (*passive.elems* (*passive-of* *St*′) = *passive.elems* (*passive-of* *St*)
∧ (*mset-set* (*set-option* (*yy-of* *St*′)) ≺≺*S* *mset-set* (*set-option* (*yy-of* *St*))
∨ (*mset-set* (*set-option* (*yy-of* *St*′)) = *mset-set* (*set-option* (*yy-of* *St*))
∧ *size* (*t-llists* (*todo-of* *St*′)) < *size* (*t-llists* (*todo-of* *St*)))))))

**lemma** *wfP-Less-state*: *wfP* (⊏)
**proof** −
**let** *?msetset* = {(*M*′, *M*). *M*′ ≺≺*S* *M*}
**let** *?natset* = {(*n*′, *n* :: *nat*). *n*′ < *n*}
**let** *?quad-of* = λ*St*. (*mset-of-zl-fstate* *St*, *mset-set* (*passive.elems* (*passive-of* *St*)),
*mset-set* (*set-option* (*yy-of* *St*)), *size* (*t-llists* (*todo-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* <∗*lex*∗> *?msetset* <∗*lex*∗> *?msetset* <∗*lex*∗> *?natset*)
**by** (*rule* *wf-lex-prod*[*OF* *wf-msetset* *wf-lex-prod*[*OF* *wf-msetset*
*wf-lex-prod*[*OF* *wf-msetset* *wf-natset*]]])

```
  have Less-state-alt-def: ⋀St' St. St' ⊏ St ⟷
    (?quad-of St', ?quad-of St) ∈ ?msetset <*lex*> ?msetset <*lex*> ?msetset <*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 ⇝ZLf St' and
    non-ci: ¬ compute-infer-step St St'
  shows St' ⊏ 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-mset (p-select P) (mset-set (passive.elems P − {p-select P})) =
    mset-set (passive.elems P)
    by (metis finite-fset local.choose-p(3) mset-set.remove passive.select-in-felems)
  have pas: mset-set (passive.elems P − {p-select P}) ≺≺S mset-set (passive.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.elems P) + mset-set (fset A) + {#C#}
  let ?new-aft = mset-set (passive.elems 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.elems 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.elems P) + mset-set (insert C′ (fset A))) =
      add-mset C (mset-set (passive.elems 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.elems P)) + mset-set (fset A)) =
      add-mset C (mset-set (passive.elems 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 ιss 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-multp)
next
  case (delete-orphan-infers ιs T A D P Y)
  note defs = this(1,2) and ιs = this(3)
  have size (t-llists T − {#ιs#}) < size (t-llists T)
    using ιs 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 ↝ZLf St′ and
    yy: yy-of St ≠ None
```

**shows** $St' \sqsubset St$
**proof** −
  **have** ¬ *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*: *llength Sts* = ∞ **and**
    *full*: *full-chain* (⇝*ZLf*) *Sts* **and**
    *inv*: *ZLf-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*) ≠ {}

  **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*

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

  **have** *yy-j*: *yy-of* (*lnth Sts j*) ≠ *None* **if** *j-ge*: *j* ≥ *i* **for** *j*
    **using** *c-in' len j-ge* **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 llength-eq-infty-conv-lfinite*
    **by** *blast*

  **have** *lnth Sts* (*Suc j*) ⊏ *lnth Sts j* **if** *j-ge*: *j* ≥ *i* **for** *j*
    **using** *yy-nonempty-ZLf-step-imp-Less-state* **by** (*meson step j-ge yy-j*)
  **hence** $(\sqsubset)^{-1-1}$ (*lnth Sts j*) (*lnth Sts* (*Suc j*))
    **if** *j-ge*: *j* ≥ *i* **for** *j*
    **using** *j-ge* **by** *blast*
  **hence** *inf-down-chain*: *chain* $(\sqsubset)^{-1-1}$ (*ldropn i Sts*)
    **by** (*simp add*: *chain-ldropnI si-lt*)

  **have** *inf-i*: ¬ *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* (⊏)] *wfP-Less-state*
    **by** *metis*
**qed**

**lemma** *ZLf-step-imp-passive-queue-step*:
  **assumes** *St* ⇝*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* ($\leadsto$*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.elems* $\circ$ *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*: $\forall j \geq i.$ ¬ *compute-infer-step* (*lnth Sts j*) (*lnth Sts* (*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*: ¬ *compute-infer-step* (*lnth Sts j*) (*lnth Sts* (*Suc j*)) **if** *j-ge*: $j \geq i$ **for** *j*
**using** *i j-ge* **by** *auto*

**have** *step*: *lnth Sts j* $\leadsto$*ZLf lnth Sts* (*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** $(\sqsubset)^{-1-1}$ (*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* (⊏)$^{-1-1}$ (*ldropn i Sts*)
**using** *chain-ldropn-lmapI*[*OF - si-lt*, *of - id*, *simplified llist.map-id*] **by** *simp*
**have** *inf-i*: ¬ *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* (⊏)] *wfP-Less-state*
**by** *blast*
**qed**

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

**have** *Liminf-llist* (*lmap passive.elems* (*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* ⇝*ZLf St′*
**shows** *todo.lqueue-step* (*todo-of St*, *done-of St*) (*todo-of St′*, *done-of St′*)
**using** *assms*
**proof** *cases*
**case** (*compute-infer T ι0 T′ A C D P*)
**note** *defs* = *this*(*1*,*2*) **and** *has-el* = *this*(*3*) **and** *pick* = *this*(*4*)
**have** *t′*: *T′* = *snd* (*t-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 ιss 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 ιss*] **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
      llength-eq-infty-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 ∃ j ≥ i. todo.pick-lqueue-step (lnth TDs j) (lnth TDs (Suc j))
        using assms
      proof cases
        case (1 T ι0 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 ι0′ :: ′f inference and ιs :: ′f inference llist where
          cons-in0: LCons ι0′ ιs ∈# t-llists T and
          fst0: fst (t-pick-elem T) = ι0′ and
          snd0: t-llists (snd (t-pick-elem T)) = t-llists T − {#LCons ι0′ ιs#} + {#ιs#}
          using todo.llists-pick-elem[OF has-el] by blast

        have ι0′: ι0′ = ι0
          using pick fst0 by auto

        have
          cons-in: LCons ι0 ιs ∈# t-llists T and
          fst: fst (t-pick-elem T) = ι0 and
```

*snd*: *t-llists* (*snd* (*t-pick-elem T*)) = *t-llists T* − {#*LCons ι0 ιs*#} + {#*ιs*#}
**unfolding** *ι0*′[*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 ι0 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 - ιs*], *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*) ⊏ *lnth Sts j* **if** *j-ge*: *j* ≥ *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* ≥ *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*: ¬ *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* (⊏)] *wfP-Less-state*
**by** *blast*
**qed**

**have** *ι* ∈ *lnth flat-Ts i*
**using** *ι-in-infs* **unfolding** *Infs-def flat-Ts-def* **by** (*simp add*: *lt-sts*)
**then obtain** *ιs* :: ′*f inference llist* **where**
*ιs-in*: *ιs* ∈# *t-llists* (*fst* (*lnth TDs i*)) **and**
*ι-in-ιs*: *ι* ∈ *lset ιs*
**using** *lnth-lmap*[*OF lt-sts*] **unfolding** *flat-Ts-def TDs-def*
**by** (*smt* (*verit*, *ccfv-SIG*) *Union-iff flat-inferences-of.simps fst-conv mem-Collect-eq*)

**obtain** *k* :: *nat* **where**
*k-lt*: *enat k* < *llength ιs* **and**
*at-k*: *lnth ιs k* = *ι*
**using** *ι-in-ιs* **by** (*meson in-lset-conv-lnth*)

**obtain** *j* :: *nat* **where**
*j-ge*: *j* ≥ *i* **and**
*rem-or-pick-step*: (∃ *k*′ ≤ *k*. ∃ *ιss*.
*ldrop* (*enat k*′) *ιs* ∈ *set ιss* ∧ *todo.remove-lqueue-step-w-details* (*lnth TDs j*) *ιss*
(*lnth TDs* (*Suc j*)))
∨ *todo.pick-lqueue-step-w-details* (*lnth TDs j*) (*lnth ιs k*) (*ldrop* (*enat* (*Suc k*)) *ιs*)
(*lnth TDs* (*Suc j*))

**using** *todo.fair-strong*[*OF chain-ts inf-oft ιs-in k-lt*] **by** *blast*

**have** $\exists j.\ j \geq i \wedge j < llength\ Sts \wedge \iota \notin lnth\ Infs\ j$
**proof** (*rule exI*[*of - Suc j*], *intro conjI*)
{
**assume** $\exists k' \leq k.\ \exists \iota ss.\ ldrop\ (enat\ k')\ \iota s \in set\ \iota ss$
$\wedge$ *todo.remove-lqueue-step-w-details* (*lnth TDs j*) *ιss* (*lnth TDs* (*Suc j*))
**then obtain** $k'$ :: *nat* **and** *ιss* :: *'f inference llist list* **where**
*k'-le*: $k' \leq k$ **and**
*in-ιss*: *ldrop* (*enat k'*) *ιs* ∈ *set ιss* **and**
*rem-step*: *todo.remove-lqueue-step-w-details* (*lnth TDs j*) *ιss* (*lnth TDs* (*Suc j*))
**by** *blast*

**have** *ι* ∉ *lnth Infs* (*Suc j*)
**using** *rem-step*
**proof** *cases*
**case** (*remove-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* ∪ ⋃ {*lset ιs* |*ιs*. *ιs* ∈ *set ιss*}
**unfolding** *at-sj* **using** *TDs-def at-sj len* **by** *auto*

**have** *ι* ∈ *lset* (*ldrop* (*enat k'*) *ιs*)
**proof** −
**have** *nth-drop*: *lnth* (*ldrop* (*enat k'*) *ιs*) (*k* − *k'*) = *ι*
**by** (*simp add*: *at-k k'-le k-lt*)
**thus** *?thesis*
**using** *at-k k'-le k-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** *ι* ∈ ⋃ {*lset ιs* |*ιs*. *ιs* ∈ *set ιss*}
**using** *in-ιss* **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 ιs k*) (*ldrop* (*enat* (*Suc k*)) *ιs*)
(*lnth TDs* (*Suc j*))
**hence** *ι* ∉ *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* ∪ {*ι*}
**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** *ι* ∉ *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* (⇝*ZLf*) *Sts* **and**
*init*: *is-initial-ZLf-state* (*lhd Sts*) **and**
*fair*: *infinitely-often compute-infer-step Sts* ⟶ *infinitely-often choose-p-step Sts*
**shows**
*fair-ZL-Liminf-saturated*: *saturated* (*labeled-formulas-of* (*Liminf-zl-fstate Sts*)) **and**
*fair-ZL-complete-Liminf*: $B \in$ *Bot-F* $\Longrightarrow$ *passive.elems* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists B' \in$ *Bot-F*. $B' \in$ *formulas-union* (*Liminf-zl-fstate Sts*) **and**
*fair-ZL-complete*: $B \in$ *Bot-F* $\Longrightarrow$ *passive.elems* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists i$. *enat i* $<$ *llength Sts* $\wedge$ ($\exists B' \in$ *Bot-F*. $B' \in$ *all-formulas-of* (*lnth Sts i*))
**proof** −
**have** *chain*: *chain* (⇝*ZLf*) *Sts*
**by** (*rule full-chain-imp-chain*[*OF full*])
**have** *zl-chain*: *chain* (⇝*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*: ¬ *lnull Sts*
**using** *chain chain-not-lnull* **by** *blast*
**hence** *lhd-lmap*: ⋀*f*. *lhd* (*lmap f Sts*) = *f* (*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* ∘ *zl-fstate*) *Sts*)) = {} ∧
*Liminf-llist* (*lmap* (*fst* ∘ *zl-fstate*) *Sts*) = {} (**is** *?pas-fml* ∧ *?t-inf*)
**proof** (*cases lfinite Sts*)
**case** *fin*: *True*

**have** *lim-fst*: *Liminf-llist* (*lmap* (*fst* ∘ *zl-fstate*) *Sts*) = *fst* (*zl-fstate* (*llast Sts*)) **and**
*lim-snd*: *Liminf-llist* (*lmap* (*snd* ∘ *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** ∀ *St'*. ¬ *llast Sts* ⇝*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** *ιss* :: *′f inference llist list* **where**
*hd*: *lhd Sts* = (*fold t-add-llist ιss t-empty*, {}, *p-empty*, *None*, {||}) **and**
*infs*: *flat-inferences-of* (*mset ιss*) = {*ι* ∈ *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 ιss t-empty*, {}, *p-empty*, *None*, {||})
**using** *hd* **by** (*simp add*: *lhd-lmap*)

**have** *no-prems-init*: ∀ *ι* ∈ *Inf-F*. *prems-of ι* = [] ⟶ *ι* ∈ *fst* (*lhd* (*lmap zl-fstate 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* ∈ *Bot-F* **and**
*unsat*: *passive.elems* (*passive-of* (*lhd Sts*)) ⊨∩𝒢 {*B*}

**have** *unsat′*: *fst* ‘ *snd* (*lhd* (*lmap zl-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* (*snd* ∘ *zl-fstate*) *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** $\exists B' \in$ *Bot-F*. $B' \in$ *formulas-union* (*Liminf-zl-fstate Sts*)
**unfolding** *Liminf-zl-fstate-def Liminf-zl-fstate-commute* **by** *auto*
**thus** $\exists i$. *enat* $i <$ *llength Sts* $\wedge$ ($\exists B' \in$ *Bot-F*. $B' \in$ *all-formulas-of* (*lnth 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* × *'p* × *'f option* × *'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* $\mathcal{G}$*-F-q*
$\mathcal{G}$*-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**
$\mathcal{G}$*-F-q* :: *'q* ⇒ *'f* ⇒ *'g set* **and**
$\mathcal{G}$*-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* ⇒ ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ *bool*
**where**
P ≠ *p-empty* ⟹
*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* ⇒ 't **where**
*todo-of St* ≡ *fst St*
**abbreviation** *passive-of* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ 'p **where**
*passive-of St* ≡ *fst* (*snd St*)
**abbreviation** *yy-of* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ 'f *option* **where**
*yy-of St* ≡ *fst* (*snd* (*snd St*))
**abbreviation** *active-of* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ 'f *fset* **where**
*active-of St* ≡ *snd* (*snd* (*snd St*))

**abbreviation** *all-formulas-of* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ 'f *set* **where**
*all-formulas-of St* ≡ *passive.elems* (*passive-of St*) ∪ *set-option* (*yy-of St*) ∪ *fset* (*active-of St*)

**definition**
*Liminf-zl-fstate* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state llist* ⇒ 'f *set* × 'f *set* × 'f *set*
**where**
*Liminf-zl-fstate Sts* =
(*Liminf-llist* (*lmap* (*passive.elems* ∘ *passive-of*) *Sts*),
*Liminf-llist* (*lmap* (*set-option* ∘ *yy-of*) *Sts*),
*Liminf-llist* (*lmap* (*fset* ∘ *active-of*) *Sts*))

## 14.3 Initial States and Invariants

**inductive** *is-initial-ZLf-wo-ghosts-state* :: ('t, 'p, 'f) *ZLf-wo-ghosts-state* ⇒ *bool* **where**
*flat-inferences-of* (*mset ιss*) = *no-labels.Inf-from* {} ⟹
*is-initial-ZLf-wo-ghosts-state* (*fold t-add-llist ιss 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** ∃ *Sts0*. *lhd Sts0* = *St0* ∧ *lmap erase* (*ltl Sts0*) = *Sts* ∧ *chain* (⇝*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** ∃ *St0 Sts*. (*lnull Sts* ∨ *chain* (⇝) *Sts* ∧ *erase St0* ⇝ *lhd Sts*) ∧ *Sts'* = *theSts0 St0 Sts*
**hence** *chain* (⇝*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* (⇝*0*) (*theSts0 St0 Sts*)
**using** *assms* **by** *auto*
**qed**

**lemma** *simul-chain*:

```
  assumes
    chain: chain (↝) Sts and
    hd: lhd Sts = erase St0
  shows ∃ Sts0. lhd Sts0 = St0 ∧ lmap erase Sts0 = Sts ∧ chain (↝0) Sts0
proof −
  {
    assume nnul: ¬ lnull (ltl Sts)
    have chain (↝) (ltl Sts) ∧ erase St0 ↝ lhd (ltl Sts)
      (is ?thesis1 ∧ ?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) ∨ (chain (↝) (ltl Sts) ∧ erase St0 ↝ 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 (↝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* ⇝*ZLf St'*
  **shows** *wo-ghosts-of St* ⇝*ZLfw wo-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-ghosts-of St0* ⇝*ZLfw St'*
  **shows** ∃ *St0'*. *wo-ghosts-of St0'* = *St'* ∧ *St0* ⇝*ZLf St0'*
  **using** *assms*
**proof** *cases*
  **case** (*compute-infer T ι0 T' A C 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* ∪ {*ι0*}, *p-add C P*, *None*, *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* ⇝*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* ⇝*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* ⇝*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 ιss 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-add-llist ιss T*, *D* − *flat-inferences-of* (*mset ιss*), *P*, *None*, *A* |∪| {|*C*|})

**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.schedule-infer*[*OF rest*])

**show** *?thesis*
  **by** (*rule exI*[*of - St0′*]) (*use wo-st0′ step0* **in** *blast*)
**next**
**case** (*delete-orphan-infers ιs T A P Y*)
**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-remove-llist ιs T, D ∪ lset ιs, P, Y, A)*

**have** *wo-st0'*: *wo-ghosts-of St0' = St'*
**unfolding** *St0'-def st'* **by** *simp*

**have** *st0*: *St0 = (T, D, P, Y, 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.delete-orphan-infers*[*OF rest*])

**show** *?thesis*
**by** (*rule exI*[*of - St0'*]) (*use wo-st0' step0* **in** *blast*)
**qed**

**interpretation** *bisim*: *bisim wo-ghosts-of* (*⇝ZLfw*) (*⇝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* (*⇝ZLfw*) *Sts*
**shows** ∃*Sts0*. *lmap wo-ghosts-of Sts0 = Sts* ∧ *chain* (*⇝ZLf*) *Sts0* ∧ *done-of* (*lhd Sts0*) = {}
**proof** −
**define** *St0* :: (*'t*, *'p*, *'f*) *ZLf-state* **where**
*St0 = (todo-of (lhd Sts), {}, passive-of (lhd Sts), yy-of (lhd Sts), active-of (lhd Sts))*

**have** *hd*: *lhd Sts = wo-ghosts-of St0*
**unfolding** *St0-def* **by** (*cases lhd Sts*) *auto*

**obtain** *Sts0* **where**
*wog0*: *lmap wo-ghosts-of Sts0 = Sts* **and**
*chain0*: *chain* (*⇝ZLf*) *Sts0* **and**
*hd0*: *lhd Sts0 = St0*
**using** *bisim.simul-chain*[*OF chain hd*] **by** *blast*

**have** *don0*: *done-of* (*lhd Sts0*) = {}
**unfolding** *hd0 St0-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* (*⇝ZLfw*) *Sts*
**shows** ∃*Sts0*. *Sts = lmap wo-ghosts-of Sts0* ∧ *full-chain* (*⇝ZLf*) *Sts0* ∧ *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* (*⇝ZLfw*) *Sts* **and**
*init*: *is-initial-ZLf-wo-ghosts-state* (*lhd Sts*) **and**
*fair*: *infinitely-often compute-infer-step Sts* ⟶ *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*: $B \in$ *Bot-F* $\Longrightarrow$
*passive.elems* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists B' \in$ *Bot-F*. $B' \in$ *formulas-union* (*Liminf-zl-fstate Sts*) **and**
*fair-ZL-wo-ghosts-complete*: $B \in$ *Bot-F* $\Longrightarrow$ *passive.elems* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$ $\Longrightarrow$
$\exists\, i$. *enat* $i <$ *llength Sts* $\wedge$ ($\exists B \in$ *Bot-F*. $B \in$ *all-formulas-of* (*lnth Sts i*))
**proof** −
**obtain** *Sts0* :: (*'t*, *'p*, *'f*) *ZLf-state llist* **where**
*full0*: *full-chain* ($\leadsto$*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*: $B \in$ *Bot-F* **and**
*unsat*: *passive.elems* (*passive-of* (*lhd Sts*)) $\models\cap\mathcal{G}$ $\{B\}$

**have** *unsat0*: *passive.elems* (*w-ghosts.passive-of* (*lhd Sts0*)) $\models\cap\mathcal{G}$ $\{B\}$
**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** $\exists B' \in$ *Bot-F*. $B' \in$ *formulas-union* (*w-ghosts.Liminf-zl-fstate Sts0*)
      **by** (*rule fair-ZL-complete-Liminf*[*OF full0 init0 fair0 bot unsat0*])
    **thus** $\exists B' \in$ *Bot-F*. $B' \in$ *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*. *enat i* < *llength Sts* $\wedge$ ($\exists B \in$ *Bot-F*. $B \in$ *all-formulas-of* (*lnth 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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-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* ⇒ *'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**
    $\mathcal{G}$*-F-q* :: *'q* ⇒ *'f* ⇒ *'g set* **and**
    $\mathcal{G}$*-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*) +
  **fixes**
    *Prec-S* :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ‹≺S› *50*)
  **assumes**
    *wfp-Prec-S*: *wfp* (≺S) **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* $\mathcal{G}$*-F-q* $\mathcal{G}$*-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** *wfp* (≺S)

**by** (*rule wfp-Prec-S*)
**next**
**show** *transp* $(\prec S)$
**by** (*rule transp-Prec-S*)
**next**
**show** $\bigwedge A\ C.$ *finite* $A \Longrightarrow$ *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**