Extensions to the Comprehensive Framework for Saturation Theorem Proving

Jasmin Blanchette Sophie Tourret

May 26, 2024

Abstract

This Isabelle/HOL formalization extends the `Saturation_Framework` entry of the *Archive of Formal Proofs* with the following contributions:

- an application of the framework to prove Bachmair and Ganzinger's resolution prover RP refutationally complete, which was formalized in a more ad hoc fashion by Schlichtkrull et al. in the *AFP* entry `Ordered_Resultion_Prover`;
- generalizations of various basic concepts formalized by Schlichtkrull et al., which were needed to verify RP and could be useful to formalize other calculi, such as superposition;
- alternative proofs of fairness (and hence saturation and ultimately refutational completeness) for the eager and lazy given clause procedures (GC and LGC) based on invariance.

Contents

1 Soundness

theory *Soundness*
imports *Saturation-Framework.Calculus*
begin

Although consistency-preservation usually suffices, soundness is a more precise concept and is sometimes useful.

locale *sound-inference-system* = *inference-system* + *consequence-relation* +
assumes
sound: ‹$\iota \in Inf \Longrightarrow set\ (prems\text{-}of\ \iota) \models \{concl\text{-}of\ \iota\}$›
begin

lemma *Inf-consist-preserving*:
assumes *n-cons*: $\neg\ N \models Bot$
shows $\neg\ N \cup concl\text{-}of\ `\ Inf\text{-}from\ N \models Bot$
proof −
have $N \models concl\text{-}of\ `\ Inf\text{-}from\ N$
using *sound* **unfolding** *Inf-from-def image-def Bex-def mem-Collect-eq*
by (*smt all-formulas-entailed entails-trans mem-Collect-eq subset-entailed*)
then show *?thesis*
using *n-cons entails-trans-strong* **by** *blast*
qed

end

The limit of a derivation based on a redundancy criterion is satisfiable if and only if the initial set is satisfiable. This material is partly based on Section 4.1 of Bachmair and Ganzinger's *Handbook* chapter, but adapted to the saturation framework of Waldmann et al.

context *calculus*
begin

The next three lemmas correspond to Lemma 4.2:

lemma *Red-F-Sup-subset-Red-F-Liminf*:
chain (▷) *Ns* $\Longrightarrow$ *Red-F* (*Sup-llist Ns*) $\subseteq$ *Red-F* (*Liminf-llist Ns*)
by (*metis Liminf-llist-subset-Sup-llist Red-in-Sup Un-absorb1 calculus.Red-F-of-Red-F-subset calculus-axioms double-diff sup-ge2*)

lemma *Red-I-Sup-subset-Red-I-Liminf*:
chain (▷) *Ns* $\Longrightarrow$ *Red-I* (*Sup-llist Ns*) $\subseteq$ *Red-I* (*Liminf-llist Ns*)
by (*metis Liminf-llist-subset-Sup-llist Red-I-of-Red-F-subset Red-in-Sup double-diff subset-refl*)

Proof idea due to Uwe Waldmann:

lemma *unsat-limit-iff*:
assumes
chain-red: *chain* (▷) *Ns* **and**

```
    chain-ent: chain (⊨) Ns
  shows Liminf-llist Ns ⊨ Bot ⟷ lhd Ns ⊨ Bot
proof
  assume Liminf-llist Ns ⊨ Bot
  moreover have Sup-llist Ns ⊨ Liminf-llist Ns
    by (simp add: Liminf-llist-subset-Sup-llist subset-entailed)
  moreover have lhd Ns ⊨ Sup-llist Ns
  proof −
    have lhd Ns ⊨ lnth Ns i if i < llength Ns for i
      using that
    proof (induct i)
      case 0
      then show ?case
        using chain-ent chain-not-lnull lhd-conv-lnth subset-entailed by fastforce
    next
      case (Suc i)
      then show ?case
        using Suc-ile-eq chain-ent chain-lnth-rel entails-trans less-le by blast
    qed
    thus ?thesis
      unfolding Sup-llist-def using entail-unions by fastforce
  qed
  ultimately show lhd Ns ⊨ Bot
    using entails-trans by blast
next
  assume lhd Ns ⊨ Bot
  then have Sup-llist Ns ⊨ Bot
    by (meson chain-ent chain-not-lnull entails-trans lhd-subset-Sup-llist subset-entailed)
  then have Sup-llist Ns − Red-F (Sup-llist Ns) ⊨ Bot
    using Red-F-Bot entail-set-all-formulas by blast
  then have Liminf-llist Ns − Red-F (Sup-llist Ns) ⊨ Bot
    by (smt Diff-idemp Diff-mono Diff-subset Liminf-llist-subset-Sup-llist
        Red-F-Sup-subset-Red-F-Liminf Red-F-of-subset Red-in-Sup antisym-conv chain-red double-diff
        entail-set-all-formulas order-refl order-trans subset-antisym)
  then show Liminf-llist Ns ⊨ Bot
    by (meson Diff-subset entails-trans subset-entailed)
qed
```

Some easy consequences:

```
lemma Red-F-limit-Sup: chain (▷) Ns ⟹ Red-F (Liminf-llist Ns) = Red-F (Sup-llist Ns)
  by (metis Liminf-llist-subset-Sup-llist Red-F-of-Red-F-subset Red-F-of-subset Red-in-Sup
      double-diff order-refl subset-antisym)

lemma Red-I-limit-Sup: chain (▷) Ns ⟹ Red-I (Liminf-llist Ns) = Red-I (Sup-llist Ns)
  by (metis Liminf-llist-subset-Sup-llist Red-I-of-Red-F-subset Red-I-of-subset Red-in-Sup
      double-diff order-refl subset-antisym)

end

end
```

2 Counterexample-Reducing Inference Systems and the Standard Redundancy Criterion

theory *Standard-Redundancy-Criterion*
imports
Saturation-Framework.Calculus
HOL−Library.Multiset-Order
begin

The standard redundancy criterion can be defined uniformly for all inference systems equipped with a compact consequence relation. The essence of the refutational completeness argument can be carried out abstractly for counterexample-reducing inference systems, which enjoy a "smallest counterexample" property. This material is partly based on Section 4.2 of Bachmair and Ganzinger's *Handbook* chapter, but adapted to the saturation framework of Waldmann et al.

2.1 Counterexample-Reducing Inference Systems

abbreviation *main-prem-of* :: *'f inference* ⇒ *'f* **where**
main-prem-of ι ≡ *last* (*prems-of* ι)

abbreviation *side-prems-of* :: *'f inference* ⇒ *'f list* **where**
side-prems-of ι ≡ *butlast* (*prems-of* ι)

lemma *set-prems-of*:
set (*prems-of* ι) = (*if prems-of* ι = [] *then* {} *else* {*main-prem-of* ι} ∪ *set* (*side-prems-of* ι))
by *clarsimp* (*metis Un-insert-right append-Nil2 append-butlast-last-id list.set*(*2*) *set-append*)

locale *counterex-reducing-inference-system* = *inference-system Inf* + *consequence-relation*
for *Inf* :: *'f inference set* +
fixes
I-of :: *'f set* ⇒ *'f set* **and**
less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ≺ *50*)
assumes
wfP-less: *wfP* (≺) **and**
Inf-counterex-reducing:
N ∩ *Bot* = {} ⟹ *D* ∈ *N* ⟹ ¬ *I-of N* ⊨ {*D*} ⟹
(⋀*C*. *C* ∈ *N* ⟹ ¬ *I-of N* ⊨ {*C*} ⟹ *D* ≺ *C* ∨ *D* = *C*) ⟹
∃ι ∈ *Inf*. *prems-of* ι ≠ [] ∧ *main-prem-of* ι = *D* ∧ *set* (*side-prems-of* ι) ⊆ *N* ∧
I-of N ⊨ *set* (*side-prems-of* ι) ∧ ¬ *I-of N* ⊨ {*concl-of* ι} ∧ *concl-of* ι ≺ *D*

begin

lemma *ex-min-counterex*:
fixes *N* :: *'f set*
assumes ¬ *I* ⊨ *N*
shows ∃ *C* ∈ *N*. ¬ *I* ⊨ {*C*} ∧ (∀ *D* ∈ *N*. *D* ≺ *C* ⟶ *I* ⊨ {*D*})
proof −
obtain *C* **where**
C ∈ *N* **and** ¬ *I* ⊨ {*C*}
using *assms all-formulas-entailed* **by** *blast*
then have *c-in*: *C* ∈ {*C* ∈ *N*. ¬ *I* ⊨ {*C*}}
by *blast*
show *?thesis*
using *wfP-eq-minimal*[*THEN iffD1*, *rule-format*, *OF wfP-less c-in*] **by** *blast*

qed

end

Theorem 4.4 (generalizes Theorems 3.9 and 3.16):

locale *counterex-reducing-inference-system-with-trivial-redundancy* =
  counterex-reducing-inference-system - - *Inf* + *calculus* - *Inf* - λ-. {} λ-. {}
  for *Inf* :: *'f inference set* +
  assumes *less-total*: $\bigwedge C\ D.\ C \neq D \Longrightarrow C \prec D \vee D \prec C$
begin

theorem *saturated-model*:
  assumes
    satur: *saturated N* **and**
    bot-ni-n: $N \cap Bot = \{\}$
  shows *I-of N* $\models N$
proof (*rule ccontr*)
  assume ¬ *I-of N* $\models N$
  then obtain *D* :: *'f* **where**
    d-in-n: $D \in N$ **and**
    d-cex: ¬ *I-of N* $\models \{D\}$ **and**
    d-min: $\bigwedge C.\ C \in N \Longrightarrow C \prec D \Longrightarrow$ *I-of N* $\models \{C\}$
    by (*meson ex-min-counterex*)
  then obtain ι :: *'f inference* **where**
    ι-inf: $\iota \in Inf$ **and**
    concl-cex: ¬ *I-of N* $\models \{$*concl-of* ι$\}$ **and**
    concl-lt-d: *concl-of* ι ≺ *D*
    using *Inf-counterex-reducing*[*OF bot-ni-n*] *less-total*
    by *force*
  have *concl-of* ι ∈ *N*
    using *ι-inf Red-I-of-Inf-to-N* **by** *blast*
  then show *False*
    using *concl-cex concl-lt-d d-min* **by** *blast*
qed

An abstract version of Corollary 3.10 does not hold without some conditions, according to Nitpick:

corollary *saturated-complete*:
  assumes
    satur: *saturated N* **and**
    unsat: *N* $\models$ *Bot*
  shows $N \cap Bot \neq \{\}$
  oops

end

2.2 Compactness

locale *concl-compact-consequence-relation* = *consequence-relation* +
  assumes
    entails-concl-compact: *finite EE* $\Longrightarrow$ *CC* $\models$ *EE* $\Longrightarrow \exists\, CC' \subseteq CC.$ *finite CC'* $\wedge$ *CC'* $\models$ *EE*
begin

lemma *entails-concl-compact-union*:
  assumes

fin-e: *finite EE* **and**
cd-ent: $CC \cup DD \models EE$
shows $\exists\, CC' \subseteq CC.$ *finite* $CC' \wedge CC' \cup DD \models EE$
proof −
obtain $CCDD'$ **where**
cd1-fin: *finite* $CCDD'$ **and**
cd1-sub: $CCDD' \subseteq CC \cup DD$ **and**
cd1-ent: $CCDD' \models EE$
using *entails-concl-compact*[*OF fin-e cd-ent*] **by** *blast*

define CC' **where**
$CC' = CCDD' - DD$
have $CC' \subseteq CC$
unfolding *CC'-def* **using** *cd1-sub* **by** *blast*
moreover have *finite* CC'
unfolding *CC'-def* **using** *cd1-fin* **by** *blast*
moreover have $CC' \cup DD \models EE$
unfolding *CC'-def* **using** *cd1-ent*
by (*metis Un-Diff-cancel2 Un-upper1 entails-trans subset-entailed*)
ultimately show *?thesis*
by *blast*
qed

end

2.3 The Finitary Standard Redundancy Criterion

locale *finitary-standard-formula-redundancy* =
consequence-relation Bot ($\models$)
for
Bot :: *'f set* **and**
entails :: *'f set* ⇒ *'f set* ⇒ *bool* (**infix** $\models$ *50*) +
fixes
less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** $\prec$ *50*)
assumes
transp-less: *transp* ($\prec$) **and**
wfP-less: *wfP* ($\prec$)
begin

definition *Red-F* :: *'f set* ⇒ *'f set* **where**
Red-F N = $\{C.\ \exists\, DD \subseteq N.$ *finite DD* $\wedge$ $DD \models \{C\} \wedge (\forall\, D \in DD.\ D \prec C)\}$

The following results correspond to Lemma 4.5. The lemma *wlog-non-Red-F* generalizes the core of the argument.

lemma *Red-F-of-subset*: $N \subseteq N' \Longrightarrow$ *Red-F N* $\subseteq$ *Red-F N'*
unfolding *Red-F-def* **by** *fast*

lemma *wlog-non-Red-F*:
assumes
dd0-fin: *finite DD0* **and**
dd0-sub: $DD0 \subseteq N$ **and**
dd0-ent: $DD0 \cup CC \models \{E\}$ **and**
dd0-lt: $\forall\, D' \in DD0.\ D' \prec D$
shows $\exists\, DD \subseteq N -$ *Red-F N*. *finite DD* $\wedge$ $DD \cup CC \models \{E\} \wedge (\forall\, D' \in DD.\ D' \prec D)$
proof −

have *mset-DD0-in*: *mset-set DD0* ∈
  {*DD*. *set-mset DD* ⊆ *N* ∧ *set-mset DD* ∪ *CC* ⊨ {*E*} ∧ (∀ *D'* ∈ *set-mset DD*. *D'* ≺ *D*)}
  using *assms finite-set-mset-mset-set* **by** *simp*
obtain *DD* :: *'f multiset* **where**
  dd-subs-n: *set-mset DD* ⊆ *N* **and**
  ddcc-ent-e: *set-mset DD* ∪ *CC* ⊨ {*E*} **and**
  dd-lt-d: ∀ *D'* ∈# *DD*. *D'* ≺ *D* **and**
  d-min: ∀ *y*. *multp* (≺) *y DD* ⟶
    y ∉ {*DD*. *set-mset DD* ⊆ *N* ∧ *set-mset DD* ∪ *CC* ⊨ {*E*} ∧ (∀ *D'*∈#*DD*. *D'* ≺ *D*)}
  using *wfP-eq-minimal*[*THEN iffD1*, *rule-format*, *OF wfP-less*[*THEN wfP-multp*] *mset-DD0-in*]
  by *blast*

have ∀ *Da* ∈# *DD*. *Da* ∉ *Red-F N*
proof *clarify*
  fix *Da* :: *'f*
  assume
    da-in-dd: *Da* ∈# *DD* **and**
    da-rf: *Da* ∈ *Red-F N*

  obtain *DDa0* :: *'f set* **where**
    dda0-subs-n: *DDa0* ⊆ *N* **and**
    dda0-fin: *finite DDa0* **and**
    dda0-ent-da: *DDa0* ⊨ {*Da*} **and**
    dda0-lt-da: ∀ *D* ∈ *DDa0*. *D* ≺ *Da*
    using *da-rf* **unfolding** *Red-F-def mem-Collect-eq*
    by *blast*

  define *DDa* :: *'f multiset* **where**
    DDa = *DD* − {#*Da*#} + *mset-set DDa0*

  have *set-mset DDa* ⊆ *N*
    unfolding *DDa-def* **using** *dd-subs-n dda0-subs-n finite-set-mset-mset-set*[*OF dda0-fin*]
    by (*smt* (*verit*, *best*) *contra-subsetD in-diffD subsetI union-iff*)
  moreover have *set-mset DDa* ∪ *CC* ⊨ {*E*}
  proof (*rule entails-trans-strong*[*of* - {*Da*}])
    show *set-mset DDa* ∪ *CC* ⊨ {*Da*}
      unfolding *DDa-def set-mset-union finite-set-mset-mset-set*[*OF dda0-fin*]
      by (*rule entails-trans*[*OF* - *dda0-ent-da*]) (*auto intro*: *subset-entailed*)
  next
    have *H*: *set-mset* (*DD* − {#*Da*#} + *mset-set DDa0*) ∪ *CC* ∪ {*Da*} =
      set-mset (*DD* + *mset-set DDa0*) ∪ *CC*
      by (*smt* (*verit*) *Un-insert-left Un-insert-right da-in-dd insert-DiffM*
          set-mset-add-mset-insert set-mset-union sup-bot.right-neutral)
    show *set-mset DDa* ∪ *CC* ∪ {*Da*} ⊨ {*E*}
      unfolding *DDa-def H*
      by (*rule entails-trans*[*OF* - *ddcc-ent-e*]) (*auto intro*: *subset-entailed*)
  qed
  moreover have ∀ *D'* ∈# *DDa*. *D'* ≺ *D*
    using *dd-lt-d dda0-lt-da da-in-dd* **unfolding** *DDa-def*
    using *transp-less*[*THEN transpD*]
    using *finite-set-mset-mset-set*[*OF dda0-fin*]
    by (*metis insert-DiffM2 union-iff*)
  moreover have *multp* (≺) *DDa DD*
    unfolding *DDa-def multp-eq-multp*$_{DM}$[*OF wfP-imp-asymp*[*OF wfP-less*] *transp-less*] *multp*$_{DM}$*-def*
    using *finite-set-mset-mset-set*[*OF dda0-fin*]

```
    by (metis da-in-dd dda0-lt-da mset-subset-eq-single multi-self-add-other-not-self
      union-single-eq-member)
   ultimately show False
    using d-min by (auto intro!: antisym)
  qed
  then show ?thesis
   using dd-subs-n ddcc-ent-e dd-lt-d by blast
qed

lemma Red-F-imp-ex-non-Red-F:
  assumes c-in: C ∈ Red-F N
  shows ∃ CC ⊆ N − Red-F N. finite CC ∧ CC ⊨ {C} ∧ (∀ C' ∈ CC. C' ≺ C)
proof −
  obtain DD :: 'f set where
   dd-fin: finite DD and
   dd-sub: DD ⊆ N and
   dd-ent: DD ⊨ {C} and
   dd-lt: ∀ D ∈ DD. D ≺ C
   using c-in[unfolded Red-F-def] by fast
  show ?thesis
   by (rule wlog-non-Red-F[of DD N {} C C, simplified, OF dd-fin dd-sub dd-ent dd-lt])
qed

lemma Red-F-subs-Red-F-diff-Red-F: Red-F N ⊆ Red-F (N − Red-F N)
proof
  fix C
  assume c-rf: C ∈ Red-F N
  then obtain CC :: 'f set where
   cc-subs: CC ⊆ N − Red-F N and
   cc-fin: finite CC and
   cc-ent-c: CC ⊨ {C} and
   cc-lt-c: ∀ C' ∈ CC. C' ≺ C
   using Red-F-imp-ex-non-Red-F[of C N] by blast
  have ∀ D ∈ CC. D ∉ Red-F N
   using cc-subs by fast
  then have cc-nr:
   ∀ C ∈ CC. ∀ DD ⊆ N. finite DD ∧ DD ⊨ {C} ⟶ (∃ D ∈ DD. ¬ D ≺ C)
   unfolding Red-F-def by simp
  have CC ⊆ N
   using cc-subs by auto
  then have CC ⊆ N − {C. ∃ DD ⊆ N. finite DD ∧ DD ⊨ {C} ∧ (∀ D ∈ DD. D ≺ C)}
   using cc-nr by blast
  then show C ∈ Red-F (N − Red-F N)
   using cc-fin cc-ent-c cc-lt-c unfolding Red-F-def by blast
qed

lemma Red-F-eq-Red-F-diff-Red-F: Red-F N = Red-F (N − Red-F N)
  by (simp add: Red-F-of-subset Red-F-subs-Red-F-diff-Red-F set-eq-subset)
```

The following results correspond to Lemma 4.6.

```
lemma Red-F-of-Red-F-subset: N' ⊆ Red-F N ⟹ Red-F N ⊆ Red-F (N − N')
  by (metis Diff-mono Red-F-eq-Red-F-diff-Red-F Red-F-of-subset order-refl)

lemma Red-F-model: M ⊨ N − Red-F N ⟹ M ⊨ N
  by (metis (no-types) DiffI Red-F-imp-ex-non-Red-F entail-set-all-formulas entails-trans
```

subset-entailed)

lemma *Red-F-Bot*: $B \in Bot \Longrightarrow N \models \{B\} \Longrightarrow N - Red\text{-}F\ N \models \{B\}$
using *Red-F-model entails-trans subset-entailed* **by** *blast*

end

locale *calculus-with-finitary-standard-redundancy* =
inference-system Inf + *finitary-standard-formula-redundancy Bot* ($\models$) ($\prec$)
for
Inf :: *'f inference set* **and**
Bot :: *'f set* **and**
entails :: *'f set* ⇒ *'f set* ⇒ *bool* (**infix** $\models$ *50*) **and**
less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** $\prec$ *50*) +
assumes
Inf-has-prem: $\iota \in Inf \Longrightarrow prems\text{-}of\ \iota \neq []$ **and**
Inf-reductive: $\iota \in Inf \Longrightarrow concl\text{-}of\ \iota \prec main\text{-}prem\text{-}of\ \iota$
begin

definition *redundant-infer* :: *'f set* ⇒ *'f inference* ⇒ *bool* **where**
redundant-infer N $\iota \longleftrightarrow$
($\exists\, DD \subseteq N.\ finite\ DD \wedge DD \cup set\ (side\text{-}prems\text{-}of\ \iota) \models \{concl\text{-}of\ \iota\} \wedge (\forall\, D \in DD.\ D \prec main\text{-}prem\text{-}of\ \iota)$)

definition *Red-I* :: *'f set* ⇒ *'f inference set* **where**
Red-I N = $\{\iota \in Inf.\ redundant\text{-}infer\ N\ \iota\}$

The following results correspond to Lemma 4.6. It also uses *wlog-non-Red-F*.

lemma *Red-I-of-subset*: $N \subseteq N' \Longrightarrow Red\text{-}I\ N \subseteq Red\text{-}I\ N'$
unfolding *Red-I-def redundant-infer-def* **by** *auto*

lemma *Red-I-subs-Red-I-diff-Red-F*: $Red\text{-}I\ N \subseteq Red\text{-}I\ (N - Red\text{-}F\ N)$
proof
fix ι
assume *ι-ri*: $\iota \in Red\text{-}I\ N$
define *CC* :: *'f set* **where**
CC = *set* (*side-prems-of* ι)
define *D* :: *'f* **where**
D = *main-prem-of* ι
define *E* :: *'f* **where**
E = *concl-of* ι
obtain *DD* :: *'f set* **where**
dd-fin: *finite DD* **and**
dd-sub: $DD \subseteq N$ **and**
dd-ent: $DD \cup CC \models \{E\}$ **and**
dd-lt-d: $\forall\, C \in DD.\ C \prec D$
using *ι-ri* **unfolding** *Red-I-def redundant-infer-def CC-def D-def E-def* **by** *blast*
obtain *DDa* :: *'f set* **where**
$DDa \subseteq N - Red\text{-}F\ N$ **and** *finite DDa* **and** $DDa \cup CC \models \{E\}$ **and** $\forall\, D' \in DDa.\ D' \prec D$
using *wlog-non-Red-F*[*OF dd-fin dd-sub dd-ent dd-lt-d*] **by** *blast*
then show $\iota \in Red\text{-}I\ (N - Red\text{-}F\ N)$
using *ι-ri* **unfolding** *Red-I-def redundant-infer-def CC-def D-def E-def* **by** *blast*
qed

lemma *Red-I-eq-Red-I-diff-Red-F*: $Red\text{-}I\ N = Red\text{-}I\ (N - Red\text{-}F\ N)$

by (*metis Diff-subset Red-I-of-subset Red-I-subs-Red-I-diff-Red-F subset-antisym*)

lemma *Red-I-to-Inf*: *Red-I N* ⊆ *Inf*
unfolding *Red-I-def* **by** *blast*

lemma *Red-I-of-Red-F-subset*: *N′* ⊆ *Red-F N* ⟹ *Red-I N* ⊆ *Red-I* (*N* − *N′*)
by (*metis Diff-mono Red-I-eq-Red-I-diff-Red-F Red-I-of-subset order-refl*)

lemma *Red-I-of-Inf-to-N*:
assumes
in-ι: *ι* ∈ *Inf* **and**
concl-in: *concl-of ι* ∈ *N*
shows *ι* ∈ *Red-I N*
proof −
have *redundant-infer N ι*
unfolding *redundant-infer-def*
by (*rule exI*[**where** *x* = {*concl-of ι*}])
(*simp add*: *Inf-reductive*[*OF in-ι*] *subset-entailed concl-in*)
then show *ι* ∈ *Red-I N*
by (*simp add*: *Red-I-def in-ι*)
qed

The following corresponds to Theorems 4.7 and 4.8:

sublocale *calculus Bot Inf* (⊨) *Red-I Red-F*
by (*unfold-locales*, *fact Red-I-to-Inf*, *fact Red-F-Bot*, *fact Red-F-of-subset*,
fact Red-I-of-subset, *fact Red-F-of-Red-F-subset*, *fact Red-I-of-Red-F-subset*,
fact Red-I-of-Inf-to-N)

end

2.4 The Standard Redundancy Criterion

locale *standard-formula-redundancy* =
concl-compact-consequence-relation Bot (⊨)
for
Bot :: *'f set* **and**
entails :: *'f set* ⇒ *'f set* ⇒ *bool* (**infix** ⊨ *50*) +
fixes
less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ≺ *50*)
assumes
transp-less: *transp* (≺) **and**
wfP-less: *wfP* (≺)
begin

definition *Red-F* :: *'f set* ⇒ *'f set* **where**
Red-F N = {*C*. ∃ *DD* ⊆ *N*. *DD* ⊨ {*C*} ∧ (∀ *D* ∈ *DD*. *D* ≺ *C*)}

Compactness of (⊨) implies that *Red-F* is equivalent to its finitary counterpart.

interpretation *fin-std-red-F*: *finitary-standard-formula-redundancy Bot* (⊨) (≺)
using *transp-less asymp-on-less wfP-less* **by** *unfold-locales*

lemma *Red-F-conv*: *Red-F* = *fin-std-red-F.Red-F*
proof (*intro ext*)
fix *N*
show *Red-F N* = *fin-std-red-F.Red-F N*

unfolding *Red-F-def fin-std-red-F.Red-F-def*
using *entails-concl-compact*
by (*smt* (*verit*, *best*) *Collect-cong finite.emptyI finite-insert subset-eq*)
qed

The results from *finitary-standard-formula-redundancy* can now be lifted.

The following results correspond to Lemma 4.5.

lemma *Red-F-of-subset*: $N \subseteq N' \Longrightarrow$ *Red-F* $N \subseteq$ *Red-F* N'
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-of-subset*)

lemma *Red-F-imp-ex-non-Red-F*: $C \in$ *Red-F* $N \Longrightarrow \exists\, CC \subseteq N -$ *Red-F* $N.$ $CC \models \{C\} \wedge (\forall\, C' \in CC.\ C' \prec C)$
unfolding *Red-F-conv*
using *fin-std-red-F.Red-F-imp-ex-non-Red-F* **by** *meson*

lemma *Red-F-subs-Red-F-diff-Red-F*: *Red-F* $N \subseteq$ *Red-F* $(N -$ *Red-F* $N)$
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-subs-Red-F-diff-Red-F*)

lemma *Red-F-eq-Red-F-diff-Red-F*: *Red-F* $N =$ *Red-F* $(N -$ *Red-F* $N)$
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-eq-Red-F-diff-Red-F*)

The following results correspond to Lemma 4.6.

lemma *Red-F-of-Red-F-subset*: $N' \subseteq$ *Red-F* $N \Longrightarrow$ *Red-F* $N \subseteq$ *Red-F* $(N - N')$
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-of-Red-F-subset*)

lemma *Red-F-model*: $M \models N -$ *Red-F* $N \Longrightarrow M \models N$
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-model*)

lemma *Red-F-Bot*: $B \in$ *Bot* $\Longrightarrow N \models \{B\} \Longrightarrow N -$ *Red-F* $N \models \{B\}$
unfolding *Red-F-conv*
by (*rule fin-std-red-F.Red-F-Bot*)

end

locale *calculus-with-standard-redundancy* =
inference-system Inf + *standard-formula-redundancy Bot* $(\models)$ $(\prec)$
for
Inf :: *'f inference set* **and**
Bot :: *'f set* **and**
entails :: *'f set* ⇒ *'f set* ⇒ *bool* (**infix** $\models$ *50*) **and**
less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** $\prec$ *50*) +
assumes
Inf-has-prem: $\iota \in$ *Inf* $\Longrightarrow$ *prems-of* $\iota \neq []$ **and**
Inf-reductive: $\iota \in$ *Inf* $\Longrightarrow$ *concl-of* $\iota \prec$ *main-prem-of* ι
begin

definition *redundant-infer* :: *'f set* ⇒ *'f inference* ⇒ *bool* **where**
redundant-infer $N\ \iota \longleftrightarrow$
$(\exists\, DD \subseteq N.$ $DD \cup$ *set* (*side-prems-of* ι) $\models \{$*concl-of* $\iota\} \wedge (\forall\, D \in DD.\ D \prec$ *main-prem-of* $\iota))$

definition *Red-I* :: *'f set* ⇒ *'f inference set* **where**
Red-I N = {*ι* ∈ *Inf*. *redundant-infer N ι*}

Compactness of (⊨) implies that *Red-I* is equivalent to its finitary counterpart.

interpretation *fin-std-red*: *calculus-with-finitary-standard-redundancy Inf Bot* (⊨)
using *transp-less asymp-on-less wfP-less Inf-has-prem Inf-reductive* **by** *unfold-locales*

lemma *redundant-infer-conv*: *redundant-infer* = *fin-std-red.redundant-infer*
proof (*intro ext*)
fix *N ι*
show *redundant-infer N ι* ⟷ *fin-std-red.redundant-infer N ι*
unfolding *redundant-infer-def fin-std-red.redundant-infer-def*
using *entails-concl-compact-union*
by (*smt* (*verit*, *ccfv-threshold*) *finite.emptyI finite-insert subset-eq*)
qed

lemma *Red-I-conv*: *Red-I* = *fin-std-red.Red-I*
unfolding *Red-I-def fin-std-red.Red-I-def*
unfolding *redundant-infer-conv*
by (*rule refl*)

The results from *calculus-with-finitary-standard-redundancy* can now be lifted.

The following results correspond to Lemma 4.6.

lemma *Red-I-of-subset*: *N* ⊆ *N′* ⟹ *Red-I N* ⊆ *Red-I N′*
unfolding *Red-I-conv*
by (*rule fin-std-red.Red-I-of-subset*)

lemma *Red-I-subs-Red-I-diff-Red-F*: *Red-I N* ⊆ *Red-I* (*N* − *Red-F N*)
unfolding *Red-F-conv Red-I-conv*
by (*rule fin-std-red.Red-I-subs-Red-I-diff-Red-F*)

lemma *Red-I-eq-Red-I-diff-Red-F*: *Red-I N* = *Red-I* (*N* − *Red-F N*)
unfolding *Red-F-conv Red-I-conv*
by (*rule fin-std-red.Red-I-eq-Red-I-diff-Red-F*)

lemma *Red-I-to-Inf*: *Red-I N* ⊆ *Inf*
unfolding *Red-I-conv*
by (*rule fin-std-red.Red-I-to-Inf*)

lemma *Red-I-of-Red-F-subset*: *N′* ⊆ *Red-F N* ⟹ *Red-I N* ⊆ *Red-I* (*N* − *N′*)
unfolding *Red-F-conv Red-I-conv*
by (*rule fin-std-red.Red-I-of-Red-F-subset*)

lemma *Red-I-of-Inf-to-N*:
ι ∈ *Inf* ⟹ *concl-of ι* ∈ *N* ⟹ *ι* ∈ *Red-I N*
unfolding *Red-I-conv*
by (*rule fin-std-red.Red-I-of-Inf-to-N*)

The following corresponds to Theorems 4.7 and 4.8:

sublocale *calculus Bot Inf* (⊨) *Red-I Red-F*
by (*unfold-locales*, *fact Red-I-to-Inf*, *fact Red-F-Bot*, *fact Red-F-of-subset*,
fact Red-I-of-subset, *fact Red-F-of-Red-F-subset*, *fact Red-I-of-Red-F-subset*,
fact Red-I-of-Inf-to-N)

end

2.5 Refutational Completeness

locale *calculus-with-standard-inference-redundancy* = *calculus Bot Inf* (⊨) *Red-I Red-F*
  for *Bot* :: *'f set* **and** *Inf* **and** *entails* (**infix** ⊨ *50*) **and** *Red-I* **and** *Red-F* +
  fixes
    less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ≺ *50*)
  assumes
    Inf-has-prem: ι ∈ *Inf* ⟹ *prems-of* ι ≠ [] **and**
    Red-I-imp-redundant-infer: ι ∈ *Red-I N* ⟹
      (∃ *DD*⊆*N*. *DD* ∪ *set* (*side-prems-of* ι) ⊨ {*concl-of* ι} ∧ (∀ *C*∈*DD*. *C* ≺ *main-prem-of* ι))

sublocale *calculus-with-finitary-standard-redundancy* ⊆
  calculus-with-standard-inference-redundancy Bot Inf (⊨) *Red-I Red-F*
  using *Inf-has-prem*
  by (*unfold-locales*) (*auto simp*: *Red-I-def redundant-infer-def*)

sublocale *calculus-with-standard-redundancy* ⊆
  calculus-with-standard-inference-redundancy Bot Inf (⊨) *Red-I Red-F*
  using *Inf-has-prem*
  by (*unfold-locales*) (*simp-all add*: *Red-I-def redundant-infer-def*)

locale *counterex-reducing-calculus-with-standard-inferance-redundancy* =
  calculus-with-standard-inference-redundancy Bot Inf (⊨) *Red-I Red-F* (≺) +
  counterex-reducing-inference-system Bot (⊨) *Inf I-of* (≺)
  for
    Bot :: *'f set* **and**
    Inf :: *'f inference set* **and**
    entails :: *'f set* ⇒ *'f set* ⇒ *bool* (**infix** ⊨ *50*) **and**
    Red-I :: *'f set* ⇒ *'f inference set* **and**
    Red-F :: *'f set* ⇒ *'f set* **and**
    I-of :: *'f set* ⇒ *'f set* **and**
    less :: *'f* ⇒ *'f* ⇒ *bool* (**infix** ≺ *50*) +
  assumes *less-total*: ⋀*C D*. *C* ≠ *D* ⟹ *C* ≺ *D* ∨ *D* ≺ *C*
begin

The following result loosely corresponds to Theorem 4.9.

lemma *saturated-model*:
  assumes
    satur: *saturated N* **and**
    bot-ni-n: *N* ∩ *Bot* = {}
  shows *I-of N* ⊨ *N*
proof (*rule ccontr*)
  assume ¬ *I-of N* ⊨ *N*
  then obtain *D* :: *'f* **where**
    d-in-n: *D* ∈ *N* **and**
    d-cex: ¬ *I-of N* ⊨ {*D*} **and**
    d-min: ⋀*C*. *C* ∈ *N* ⟹ *C* ≺ *D* ⟹ *I-of N* ⊨ {*C*}
    using *ex-min-counterex* **by** *blast*
  then obtain ι :: *'f inference* **where**
    ι-*in*: ι ∈ *Inf* **and**
    ι-*mprem*: *D* = *main-prem-of* ι **and**
    sprem-subs-n: *set* (*side-prems-of* ι) ⊆ *N* **and**
    sprem-true: *I-of N* ⊨ *set* (*side-prems-of* ι) **and**
    concl-cex: ¬ *I-of N* ⊨ {*concl-of* ι} **and**
    concl-lt-d: *concl-of* ι ≺ *D*
    using *Inf-counterex-reducing*[*OF bot-ni-n*] *less-total* **by** *metis*

have $\iota \in$ *Red-I N*
by (*rule subsetD*[*OF satur*[*unfolded saturated-def Inf-from-def*]],
simp add: ι*-in set-prems-of Inf-has-prem*)
(*use* ι*-mprem d-in-n sprem-subs-n* **in** *blast*)
then have $\iota \in$ *Red-I N*
using *Red-I-without-red-F* **by** *blast*
then obtain *DD* :: $'f$ *set* **where**
dd-subs-n: $DD \subseteq N$ **and**
dd-cc-ent-d: $DD \cup$ *set* (*side-prems-of* ι) $\models \{$*concl-of* $\iota\}$ **and**
dd-lt-d: $\forall C \in DD.\ C \prec D$
unfolding ι*-mprem* **using** *Red-I-imp-redundant-infer* **by** *meson*
from *dd-subs-n dd-lt-d* **have** *I-of N* $\models$ *DD*
using *d-min* **by** (*meson ex-min-counterex subset-iff*)
then have *I-of N* $\models \{$*concl-of* $\iota\}$
using *entails-trans dd-cc-ent-d entail-union sprem-true* **by** *blast*
then show *False*
using *concl-cex* **by** *auto*
qed

A more faithful abstract version of Theorem 4.9 does not hold without some conditions, according to Nitpick:

corollary *saturated-complete*:
assumes
satur: *saturated N* **and**
unsat: $N \models$ *Bot*
shows $N \cap$ *Bot* $\neq \{\}$
oops

end

end

3 Clausal Calculi

theory *Clausal-Calculus*
imports
Ordered-Resolution-Prover.Unordered-Ground-Resolution
Soundness
Standard-Redundancy-Criterion
begin

Various results about consequence relations, counterexample-reducing inference systems, and the standard redundancy criteria are specialized and customized for clauses as opposed to arbitrary formulas.

3.1 Setup

To avoid confusion, we use the symbol $\models$ (with or without subscripts) for the "models" and entailment relations on clauses and $\models$ for the abstract concept of consequence.

abbreviation *true-lit-thick* :: $'a$ *interp* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool* (**infix** $\models l$ *50*) **where**
$I \models l\ L \equiv I \models l\ L$

abbreviation *true-cls-thick* :: $'a$ *interp* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* (**infix** $\models$ *50*) **where**

$I \models C \equiv I \models C$

abbreviation *true-clss-thick* :: *'a interp* ⇒ *'a clause set* ⇒ *bool* (**infix** $\models s$ *50*) **where**
$I \models s\ \mathcal{C} \equiv I \models s\ \mathcal{C}$

abbreviation *true-cls-mset-thick* :: *'a interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** $\models m$ *50*) **where**
$I \models m\ \mathcal{C} \equiv I \models m\ \mathcal{C}$

no-notation *true-lit* (**infix** $\models l$ *50*)
no-notation *true-cls* (**infix** $\models$ *50*)
no-notation *true-clss* (**infix** $\models s$ *50*)
no-notation *true-cls-mset* (**infix** $\models m$ *50*)

3.2 Consequence Relation

abbreviation *entails-clss* :: *'a clause set* ⇒ *'a clause set* ⇒ *bool* (**infix** $\models e$ *50*) **where**
N1 $\models e$ *N2* ≡ ∀ *I*. *I* $\models s$ *N1* ⟶ *I* $\models s$ *N2*

lemma *entails-iff-unsatisfiable-single*:
CC $\models e$ {*E*} ⟷ ¬ *satisfiable* (*CC* ∪ {{#− *L*#} |*L*. *L* ∈# *E*}) (**is** - ⟷ - (- ∪ *?NegD*))
proof
assume *c-ent-e*: *CC* $\models e$ {*E*}
have ¬ *I* $\models s$ *CC* ∪ *?NegD* **for** *I*
using *c-ent-e*[*rule-format*, *of I*]
unfolding *true-clss-def true-cls-def true-lit-def if-distribR if-bool-eq-conj*
by (*fastforce simp*: *ball-Un is-pos-neg-not-is-pos*)
then show ¬ *satisfiable* (*CC* ∪ *?NegD*)
by *auto*
next
assume ¬ *satisfiable* (*CC* ∪ *?NegD*)
then have ¬ *I* $\models s$ *CC* ∪ *?NegD* **for** *I*
by *auto*
then show *CC* $\models e$ {*E*}
unfolding *true-clss-def true-cls-def true-lit-def if-distribR if-bool-eq-conj*
by (*fastforce simp*: *ball-Un is-pos-neg-not-is-pos*)
qed

lemma *entails-iff-unsatisfiable*:
CC $\models e$ *EE* ⟷ (∀ *E* ∈ *EE*. ¬ *satisfiable* (*CC* ∪ {{#− *L*#} |*L*. *L* ∈# *E*})) (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* ⟷ (∀ *E* ∈ *EE*. *CC* $\models e$ {*E*})
unfolding *true-clss-def* **by** *auto*
also have ... ⟷ *?rhs*
unfolding *entails-iff-unsatisfiable-single* **by** *auto*
finally show *?thesis*
.
qed

interpretation *consequence-relation* {{#}} ($\models e$)
proof
fix *N2 N1* :: *'a clause set*
assume ∀ *C* ∈ *N2*. *N1* $\models e$ {*C*}
then show *N1* $\models e$ *N2*
unfolding *true-clss-singleton* **by** (*simp add*: *true-clss-def*)
qed (*auto intro*: *true-clss-mono*)

interpretation *concl-compact-consequence-relation* $\{\{\#\}\}$ :: $('a$ :: *wellorder*$)$ *clause set* $(\models e)$
proof
fix *CC EE* :: $'a$ *clause set*
assume
fin-e: *finite EE* **and**
c-ent-e: $CC \models e\ EE$

have $\forall E \in EE.\ \neg$ *satisfiable* $(CC \cup \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\})$
using *c-ent-e*[*unfolded entails-iff-unsatisfiable*] **.**
then have $\forall E \in EE.\ \exists DD \subseteq CC \cup \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\}.$ *finite DD* $\wedge \neg$ *satisfiable DD*
by (*subst* (*asm*) *clausal-logic-compact*)
then obtain *DD-of* **where**
d-of: $\forall E \in EE.$ *DD-of* $E \subseteq CC \cup \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\} \wedge$ *finite* (*DD-of E*)
$\wedge \neg$ *satisfiable* (*DD-of E*)
by *moura*

define CC' **where**
$CC' = (\bigcup E \in EE.$ *DD-of* $E - \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\})$

have $CC' \subseteq CC$
unfolding CC'*-def* **using** *d-of* **by** *auto*
moreover have c'*-fin*: *finite* CC'
unfolding CC'*-def* **using** *d-of fin-e* **by** *blast*
moreover have $CC' \models e\ EE$
unfolding *entails-iff-unsatisfiable*
proof
fix *E*
assume *e-in*: $E \in EE$

have *DD-of* $E \subseteq CC' \cup \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\}$
using *e-in d-of* **unfolding** CC'*-def* **by** *auto*
moreover have $\neg$ *satisfiable* (*DD-of E*)
using *e-in d-of* **by** *auto*
ultimately show $\neg$ *satisfiable* $(CC' \cup \{\{\#-\ L\#\}\ |L.\ L \in\#\ E\})$
by (*rule unsatisfiable-mono*[*of DD-of E*])
qed
ultimately show $\exists CC' \subseteq CC.$ *finite* $CC' \wedge CC' \models e\ EE$
by *blast*
qed

3.3 Counterexample-Reducing Inference Systems

definition *clss-of-interp* :: $'a$ *set* $\Rightarrow$ $'a$ *literal multiset set* **where**
clss-of-interp I $= \{\{\#(\textit{if } A \in I \textit{ then Pos else Neg})\ A\#\}\ |A.\ True\}$

lemma *true-clss-of-interp-iff-equal*[*simp*]: $J \models s$ *clss-of-interp* $I \longleftrightarrow J = I$
unfolding *clss-of-interp-def true-clss-def true-cls-def true-lit-def* **by** *force*

lemma *entails-iff-models*[*simp*]: *clss-of-interp* $I \models e\ CC \longleftrightarrow I \models s\ CC$
by *simp*

locale *clausal-counterex-reducing-inference-system* = *inference-system Inf*
for *Inf* :: $('a$ :: *wellorder*$)$ *clause inference set* +
fixes *J-of* :: $'a$ *clause set* $\Rightarrow$ $'a$ *interp*
assumes *clausal-Inf-counterex-reducing*:
$\{\#\} \notin N \Longrightarrow D \in N \Longrightarrow \neg$ *J-of* $N \models D \Longrightarrow (\bigwedge C.\ C \in N \Longrightarrow \neg$ *J-of* $N \models C \Longrightarrow D \leq C) \Longrightarrow$

$\exists \iota \in \mathit{Inf}.\ \mathit{prems\text{-}of}\ \iota \neq [] \wedge \mathit{main\text{-}prem\text{-}of}\ \iota = D \wedge \mathit{set}\ (\mathit{side\text{-}prems\text{-}of}\ \iota) \subseteq N \wedge$
$\mathit{J\text{-}of}\ N \models s\ \mathit{set}\ (\mathit{side\text{-}prems\text{-}of}\ \iota) \wedge \neg\ \mathit{J\text{-}of}\ N \models \mathit{concl\text{-}of}\ \iota \wedge \mathit{concl\text{-}of}\ \iota < D$
begin

abbreviation *I-of* :: *'a clause set* ⇒ *'a clause set* **where**
I-of N ≡ *clss-of-interp* (*J-of N*)

lemma *Inf-counterex-reducing*:
assumes
bot-ni-n: $N \cap \{\{\#\}\} = \{\}$ **and**
d-in-n: $D \in N$ **and**
n-ent-d: ¬ *I-of N* ⊨*e* {*D*} **and**
d-min: $\bigwedge C.\ C \in N \Longrightarrow \neg\ \mathit{I\text{-}of}\ N \models e\ \{C\} \Longrightarrow D \leq C$
shows $\exists \iota \in \mathit{Inf}.\ \mathit{prems\text{-}of}\ \iota \neq [] \wedge \mathit{main\text{-}prem\text{-}of}\ \iota = D \wedge \mathit{set}\ (\mathit{side\text{-}prems\text{-}of}\ \iota) \subseteq N$
∧ *I-of N* ⊨*e set* (*side-prems-of* ι) ∧ ¬ *I-of N* ⊨*e* {*concl-of* ι} ∧ *concl-of* ι < *D*
using *bot-ni-n clausal-Inf-counterex-reducing d-in-n d-min n-ent-d* **by** *auto*

sublocale *counterex-reducing-inference-system* {{#}} (⊨*e*) *Inf I-of*
(<) :: *'a clause* ⇒ *'a clause* ⇒ *bool*
using *Inf-counterex-reducing*
by *unfold-locales* (*simp-all add*: *less-eq-multiset-def*)

end

3.4 Counterexample-Reducing Calculi Equipped with a Standard Redundancy Criterion

locale *clausal-counterex-reducing-calculus-with-standard-redundancy* =
calculus-with-standard-redundancy Inf {{#}} (⊨*e*) (<) :: *'a clause* ⇒ *'a clause* ⇒ *bool* +
clausal-counterex-reducing-inference-system Inf J-of
for
Inf :: (*'a* :: *wellorder*) *clause inference set* **and**
J-of :: *'a clause set* ⇒ *'a set*
begin

sublocale *counterex-reducing-calculus-with-standard-inferance-redundancy* {{#}} *Inf* (⊨*e*) *Red-I*
Red-F I-of (<) :: *'a clause* ⇒ *'a clause* ⇒ *bool*
proof *unfold-locales*
fix *C D* :: *'a clause*
show $C \neq D \Longrightarrow C < D \vee D < C$
by *fastforce*
qed

lemma *clausal-saturated-model*: *saturated N* ⟹ {#} ∉ *N* ⟹ *J-of N* ⊨*s N*
by (*simp add*: *saturated-model*[*simplified*])

corollary *clausal-saturated-complete*: *saturated N* ⟹ (∀ *I*. ¬ *I* ⊨*s N*) ⟹ {#} ∈ *N*
using *clausal-saturated-model* **by** *blast*

end

end

4 Application of the Saturation Framework to Bachmair and Ganzinger's RP

theory *FO-Ordered-Resolution-Prover-Revisited*
imports
Ordered-Resolution-Prover.FO-Ordered-Resolution-Prover
Saturation-Framework.Given-Clause-Architectures
Clausal-Calculus
Soundness
begin

The main results about Bachmair and Ganzinger's RP prover, as established in Section 4.3 of their *Handbook* chapter and formalized by Schlichtkrull et al., are re-proved here using the saturation framework of Waldmann et al.

4.1 Setup

no-notation *true-lit* (**infix** $\models l$ *50*)
no-notation *true-cls* (**infix** $\models$ *50*)
no-notation *true-clss* (**infix** $\models s$ *50*)
no-notation *true-cls-mset* (**infix** $\models m$ *50*)

hide-type (**open**) *Inference-System.inference*

hide-const (**open**) *Inference-System.Infer Inference-System.main-prem-of Inference-System.side-prems-of Inference-System.prems-of Inference-System.concl-of Inference-System.concls-of Inference-System.infer-from*

type-synonym *'a old-inference* = *'a Inference-System.inference*

abbreviation *old-Infer* ≡ *Inference-System.Infer*
abbreviation *old-side-prems-of* ≡ *Inference-System.side-prems-of*
abbreviation *old-main-prem-of* ≡ *Inference-System.main-prem-of*
abbreviation *old-concl-of* ≡ *Inference-System.concl-of*
abbreviation *old-prems-of* ≡ *Inference-System.prems-of*
abbreviation *old-concls-of* ≡ *Inference-System.concls-of*
abbreviation *old-infer-from* ≡ *Inference-System.infer-from*

lemmas *old-infer-from-def* = *Inference-System.infer-from-def*

4.2 Library

lemma *set-zip-replicate-right*[*simp*]:
set (*zip xs* (*replicate* (*length xs*) *y*)) = (λ*x*. (*x*, *y*)) ‘ *set xs*
by (*induct xs*) *auto*

4.3 Ground Layer

context *FO-resolution-prover*
begin

no-notation *RP* (**infix** ⇝ *50*)
notation *RP* (**infix** ⇝*RP* *50*)

interpretation *gr*: *ground-resolution-with-selection S-M S M*

using *selection-axioms* **by** *unfold-locales* (*fact S-M-selects-subseteq S-M-selects-neg-lits*)+

definition *G-Inf* :: *'a clause set* ⇒ *'a clause inference set* **where**
G-Inf M = {*Infer* (*CAs* @ [*DA*]) *E* |*CAs DA AAs As E*. *gr.ord-resolve M CAs DA AAs As E*}

lemma *G-Inf-have-prems*: ι ∈ *G-Inf M* ⟹ *prems-of* ι ≠ []
unfolding *G-Inf-def* **by** *auto*

lemma *G-Inf-reductive*: ι ∈ *G-Inf M* ⟹ *concl-of* ι < *main-prem-of* ι
unfolding *G-Inf-def* **by** (*auto dest*: *gr.ord-resolve-reductive*)

interpretation *G*: *sound-inference-system G-Inf M* {{#}} (⊫*e*)
proof
fix ι
assume *i-in*: ι ∈ *G-Inf M*
moreover
{
fix *I*
assume *I-ent-prems*: *I* ⊫*s set* (*prems-of* ι)
obtain *CAs AAs As* **where**
the-inf: *gr.ord-resolve M CAs* (*main-prem-of* ι) *AAs As* (*concl-of* ι) **and**
CAs: *CAs* = *side-prems-of* ι
using *i-in* **unfolding** *G-Inf-def* **by** *auto*
then have *I* ⊫ *concl-of* ι
using *gr.ord-resolve-sound*[*of M CAs main-prem-of* ι *AAs As concl-of* ι *I*]
by (*metis I-ent-prems G-Inf-have-prems i-in insert-is-Un set-mset-mset set-prems-of*
true-clss-insert true-clss-set-mset)
}
ultimately show *set* (*inference.prems-of* ι) ⊫*e* {*concl-of* ι}
by *simp*
qed

interpretation *G*: *clausal-counterex-reducing-inference-system G-Inf M gr.INTERP M*
proof
fix *N D*
assume
{#} ∉ *N* **and**
D ∈ *N* **and**
¬ *gr.INTERP M N* ⊨ *D* **and**
⋀*C*. *C* ∈ *N* ⟹ ¬ *gr.INTERP M N* ⊨ *C* ⟹ *D* ≤ *C*
then obtain *CAs AAs As E* **where**
cas-in: *set CAs* ⊆ *N* **and**
n-mod-cas: *gr.INTERP M N* ⊨*m mset CAs* **and**
ca-prod: ⋀*CA*. *CA* ∈ *set CAs* ⟹ *gr.production M N CA* ≠ {} **and**
e-res: *gr.ord-resolve M CAs D AAs As E* **and**
n-nmod-e: ¬ *gr.INTERP M N* ⊨ *E* **and**
e-lt-d: *E* < *D*
using *gr.ord-resolve-counterex-reducing* **by** *blast*
define ι **where**
ι = *Infer* (*CAs* @ [*D*]) *E*

have ι ∈ *G-Inf M*
unfolding ι*-def G-Inf-def* **using** *e-res* **by** *auto*
moreover have *prems-of* ι ≠ []
unfolding ι*-def* **by** *simp*

moreover have *main-prem-of* $\iota = D$
unfolding ι*-def* **by** *simp*
moreover have *set* (*side-prems-of* ι) $\subseteq N$
unfolding ι*-def* **using** *cas-in* **by** *simp*
moreover have *gr.INTERP M N* $\models s$ *set* (*side-prems-of* ι)
unfolding ι*-def* **using** *n-mod-cas ca-prod* **by** (*simp add*: *gr.productive-imp-INTERP true-clss-def*)
moreover have $\neg$ *gr.INTERP M N* $\models$ *concl-of* ι
unfolding ι*-def* **using** *n-nmod-e* **by** *simp*
moreover have *concl-of* $\iota < D$
unfolding ι*-def* **using** *e-lt-d* **by** *simp*
ultimately show $\exists \iota \in$ *G-Inf M*. *prems-of* $\iota \neq [] \wedge$ *main-prem-of* $\iota = D \wedge$ *set* (*side-prems-of* ι) $\subseteq N$ $\wedge$
gr.INTERP M N $\models s$ *set* (*side-prems-of* ι) $\wedge \neg$ *gr.INTERP M N* $\models$ *concl-of* $\iota \wedge$ *concl-of* $\iota < D$
by *blast*
qed

interpretation *G*: *clausal-counterex-reducing-calculus-with-standard-redundancy G-Inf M*
gr.INTERP M
using *G-Inf-have-prems G-Inf-reductive*
by (*unfold-locales*) *simp-all*

interpretation *G*: *statically-complete-calculus* {{#}} *G-Inf M* ($\models e$) *G.Red-I M G.Red-F*
by *unfold-locales* (*use G.clausal-saturated-complete* **in** *blast*)

4.4 First-Order Layer

abbreviation $\mathcal{G}$*-F* :: ‹*'a clause* ⇒ *'a clause set*› **where**
‹$\mathcal{G}$*-F* ≡ *grounding-of-cls*›

abbreviation $\mathcal{G}$*-Fset* :: ‹*'a clause set* ⇒ *'a clause set*› **where**
‹$\mathcal{G}$*-Fset* ≡ *grounding-of-clss*›

lemmas $\mathcal{G}$*-F-def* = *grounding-of-cls-def*
lemmas $\mathcal{G}$*-Fset-def* = *grounding-of-clss-def*

definition $\mathcal{G}$*-I* :: ‹*'a clause set* ⇒ *'a clause inference* ⇒ *'a clause inference set*› **where**
‹$\mathcal{G}$*-I M* ι = {*Infer* (*prems-of* ι ··*cl* ϱs) (*concl-of* ι · ϱ) |ϱ ϱs.
is-ground-subst-list $\varrho s \wedge$ *is-ground-subst* ϱ
$\wedge$ *Infer* (*prems-of* ι ··*cl* ϱs) (*concl-of* ι · ϱ) ∈ *G-Inf M*}›

abbreviation
$\mathcal{G}$*-I-opt* :: ‹*'a clause set* ⇒ *'a clause inference* ⇒ *'a clause inference set option*›
where
‹$\mathcal{G}$*-I-opt M* ι ≡ *Some* ($\mathcal{G}$*-I M* ι)›

definition *F-Inf* :: *'a clause inference set* **where**
F-Inf = {*Infer* (*CAs* @ [*DA*]) *E* | *CAs DA AAs As* σ *E*. *ord-resolve-rename S CAs DA AAs As* σ *E*}

lemma *F-Inf-have-prems*: $\iota \in$ *F-Inf* ⟹ *prems-of* $\iota \neq []$
unfolding *F-Inf-def* **by** *force*

interpretation *F*: *lifting-intersection F-Inf* {{#}} *UNIV G-Inf* λN. ($\models e$) *G.Red-I* λN. *G.Red-F* {{#}} λN. $\mathcal{G}$*-F* $\mathcal{G}$*-I-opt* λD *C C'*. *False*
proof (*unfold-locales*; (*intro ballI*)*?*)
show *UNIV* ≠ {}
by (*rule UNIV-not-empty*)

next
show *consequence-relation* {{#}} (⊨*e*)
by (*fact consequence-relation-axioms*)
next
show ⋀*M*. *tiebreaker-lifting* {{#}} *F-Inf* {{#}} (⊨*e*) (*G-Inf M*) (*G.Red-I M*) *G.Red-F* 𝒢*-F* (𝒢*-I-opt M*)
(λ*D C C'*. *False*)
proof
fix *M* ι
show *the* (𝒢*-I-opt M* ι) ⊆ *G.Red-I M* (𝒢*-F* (*concl-of* ι))
unfolding *option.sel*
proof
fix ι'
assume ι' ∈ 𝒢*-I M* ι
then obtain ϱ ϱ*s* **where**
ι': ι' = *Infer* (*prems-of* ι ··*cl* ϱ*s*) (*concl-of* ι · ϱ) **and**
ϱ*-gr*: *is-ground-subst* ϱ **and**
ϱ*-infer*: *Infer* (*prems-of* ι ··*cl* ϱ*s*) (*concl-of* ι · ϱ) ∈ *G-Inf M*
unfolding 𝒢*-I-def* **by** *blast*

show ι' ∈ *G.Red-I M* (𝒢*-F* (*concl-of* ι))
unfolding *G.Red-I-def G.redundant-infer-def mem-Collect-eq* **using** ι' ϱ*-gr* ϱ*-infer*
by (*metis inference.sel*(2) *G-Inf-reductive empty-iff ground-subst-ground-cls*
grounding-of-cls-ground insert-iff subst-cls-eq-grounding-of-cls-subset-eq
true-clss-union)
qed
qed (*auto simp*: 𝒢*-F-def ex-ground-subst*)
qed

notation *F.entails-*𝒢 (**infix** ⊨𝒢*e* 50)

lemma *F-entails-*𝒢*-iff*: *N1* ⊨𝒢*e N2* ⟷ ⋃ (𝒢*-F* ‘ *N1*) ⊨*e* ⋃ (𝒢*-F* ‘ *N2*)
unfolding *F.entails-*𝒢*-def* **by** *simp*

lemma *true-Union-grounding-of-cls-iff*:
I ⊨*s* (⋃ *C* ∈ *N*. {*C* · σ |σ. *is-ground-subst* σ}) ⟷ (∀σ. *is-ground-subst* σ ⟶ *I* ⊨*s N* ·*cs* σ)
unfolding *true-clss-def subst-clss-def* **by** *blast*

interpretation *F*: *sound-inference-system F-Inf* {{#}} (⊨𝒢*e*)
proof
fix ι
assume *i-in*: ι ∈ *F-Inf*
moreover
{
fix *I* η
assume
I-entails-prems: ∀σ. *is-ground-subst* σ ⟶ *I* ⊨*s set* (*prems-of* ι) ·*cs* σ **and**
η*-gr*: *is-ground-subst* η
obtain *CAs AAs As* σ **where**
the-inf: *ord-resolve-rename S CAs* (*main-prem-of* ι) *AAs As* σ (*concl-of* ι) **and**
CAs: *CAs* = *side-prems-of* ι
using *i-in* **unfolding** *F-Inf-def* **by** *auto*
have *prems*: *mset* (*prems-of* ι) = *mset* (*side-prems-of* ι) + {#*main-prem-of* ι#}
by (*metis* (*no-types*) *F-Inf-have-prems*[*OF i-in*] *add.right-neutral append-Cons append-Nil2*
append-butlast-last-id mset.simps(2) *mset-rev mset-single-iff-right rev-append*

```
      rev-is-Nil-conv union-mset-add-mset-right)
    have I ⊨ concl-of ι · η
      using ord-resolve-rename-sound[OF the-inf, of I η, OF - η-gr]
      unfolding CAs prems[symmetric] using I-entails-prems
      by (metis set-mset-mset set-mset-subst-cls-mset-subst-clss true-clss-set-mset)
  }
  ultimately show set (inference.prems-of ι) ⊨𝒢e {concl-of ι}
    unfolding F.entails-𝒢-def 𝒢-F-def true-Union-grounding-of-cls-iff by auto
qed

lemma G-Inf-overapprox-F-Inf: ι₀ ∈ G.Inf-from M (⋃ (𝒢-F ‘ M)) ⟹ ∃ι ∈ F.Inf-from M. ι₀ ∈ 𝒢-I M ι
proof −
  assume ι₀-in: ι₀ ∈ G.Inf-from M (⋃ (𝒢-F ‘ M))
  have prems-ι₀-in: set (prems-of ι₀) ⊆ ⋃ (𝒢-F ‘ M)
    using ι₀-in unfolding G.Inf-from-def by simp
  note ι₀-G-Inf = G.Inf-if-Inf-from[OF ι₀-in]
  then obtain CAs DA AAs As E where
    gr-res: ‹gr.ord-resolve M CAs DA AAs As E› and
    ι₀-is: ‹ι₀ = Infer (CAs @ [DA]) E›
    unfolding G-Inf-def by auto

  have CAs-in: ‹set CAs ⊆ set (prems-of ι₀)›
    by (simp add: ι₀-is subsetI)
  then have ground-CAs: ‹is-ground-cls-list CAs›
    using prems-ι₀-in union-grounding-of-cls-ground is-ground-cls-list-def is-ground-clss-def by auto
  have DA-in: ‹DA ∈ set (prems-of ι₀)›
    using ι₀-is by simp
  then have ground-DA: ‹is-ground-cls DA›
    using prems-ι₀-in union-grounding-of-cls-ground is-ground-clss-def by auto
  obtain σ where
    grounded-res: ‹ord-resolve (S-M S M) CAs DA AAs As σ E›
    using ground-ord-resolve-imp-ord-resolve[OF ground-DA ground-CAs
        gr.ground-resolution-with-selection-axioms gr-res] by auto
  have prems-ground: ‹{DA} ∪ set CAs ⊆ 𝒢-Fset M›
    using prems-ι₀-in CAs-in DA-in unfolding 𝒢-Fset-def by fast

  obtain ηs η η2 CAs0 DA0 AAs0 As0 E0 τ where
    ground-n: is-ground-subst η and
    ground-ns: is-ground-subst-list ηs and
    ground-n2: is-ground-subst η2 and
    ngr-res: ord-resolve-rename S CAs0 DA0 AAs0 As0 τ E0 and
    CAs0-is: CAs0 ··cl ηs = CAs and
    DA0-is: DA0 · η = DA and
    E0-is: E0 · η2 = E and
    prems-in: {DA0} ∪ set CAs0 ⊆ M and
    len-CAs0: length CAs0 = length CAs and
    len-ns: length ηs = length CAs
    using ord-resolve-rename-lifting[OF - grounded-res selection-axioms prems-ground] sel-stable
    by smt

  have length CAs0 = length ηs
    using len-CAs0 len-ns by simp
  then have ι₀-is′: ι₀ = Infer ((CAs0 @ [DA0]) ··cl (ηs @ [η])) (E0 · η2)
    unfolding ι₀-is by (auto simp: CAs0-is[symmetric] DA0-is[symmetric] E0-is[symmetric])
```

define ι :: *'a clause inference* **where**
ι = *Infer* (*CAs0* @ [*DA0*]) *E0*

have *i-F-Inf*: ‹$\iota \in$ *F-Inf*›
unfolding ι*-def F-Inf-def* **using** *ngr-res* **by** *auto*
have $\exists \varrho\ \varrho s.\ \iota_0$ = *Infer* ((*CAs0* @ [*DA0*]) ·*cl* ϱs) (*E0* · ϱ) $\wedge$ *is-ground-subst-list* ϱs
$\wedge$ *is-ground-subst* $\varrho \wedge$ *Infer* ((*CAs0* @ [*DA0*]) ·*cl* ϱs) (*E0* · ϱ) $\in$ *G-Inf M*
by (*rule exI*[*of* - $\eta 2$], *rule exI*[*of* - ηs @ [η]], *use ground-ns* **in**
‹*auto intro*: *ground-n ground-n2* ι_0*-G-Inf*[*unfolded* ι_0*-is'*]
simp: ι_0*-is' is-ground-subst-list-def*›)
then have ‹$\iota_0 \in \mathcal{G}$*-I M* ι›
unfolding $\mathcal{G}$*-I-def* ι*-def CAs0-is*[*symmetric*] *DA0-is*[*symmetric*] *E0-is*[*symmetric*] **by** *simp*
moreover have ‹$\iota \in$ *F.Inf-from M*›
using *prems-in i-F-Inf* **unfolding** *F.Inf-from-def* ι*-def* **by** *simp*
ultimately show *?thesis*
by *blast*
qed

interpretation *F*: *statically-complete-calculus* {{#}} *F-Inf* ($\models\mathcal{G}$ *e*) *F.Red-I-*$\mathcal{G}$ *F.Red-F-*$\mathcal{G}$*-empty*
proof (*rule F.stat-ref-comp-to-non-ground-fam-inter*; *clarsimp*; (*intro exI*)*?*)
show $\bigwedge$*M. statically-complete-calculus* {{#}} (*G-Inf M*) ($\models$*e*) (*G.Red-I M*) *G.Red-F*
by (*fact G.statically-complete-calculus-axioms*)
next
fix *N*
assume *F.saturated N*
show *F.ground.Inf-from-q N* ($\bigcup$ ($\mathcal{G}$*-F* ‘ *N*)) $\subseteq$ {ι. $\exists \iota' \in$ *F.Inf-from N*. $\iota \in \mathcal{G}$*-I N* ι'}
$\cup$ *G.Red-I N* ($\bigcup$ ($\mathcal{G}$*-F* ‘ *N*))
using *G-Inf-overapprox-F-Inf* **unfolding** *F.ground.Inf-from-q-def* $\mathcal{G}$*-I-def* **by** *fastforce*
qed

4.5 Labeled First-Order or Given Clause Layer

datatype *label* = *New* | *Processed* | *Old*

abbreviation *F-Equiv* :: *'a clause* ⇒ *'a clause* ⇒ *bool* (**infix** $\doteq$ *50*) **where**
$C \doteq D \equiv$ *generalizes C D* $\wedge$ *generalizes D C*

abbreviation *F-Prec* :: *'a clause* ⇒ *'a clause* ⇒ *bool* (**infix** $\prec\cdot$ *50*) **where**
$C \prec\cdot D \equiv$ *strictly-generalizes C D*

fun *L-Prec* :: *label* ⇒ *label* ⇒ *bool* (**infix** $\sqsubset l$ *50*) **where**
Old $\sqsubset l\ l \longleftrightarrow l \neq$ *Old*
| *Processed* $\sqsubset l\ l \longleftrightarrow l =$ *New*
| *New* $\sqsubset l\ l \longleftrightarrow$ *False*

lemma *irrefl-L-Prec*: $\neg\ l \sqsubset l\ l$
by (*cases l*) *auto*

lemma *trans-L-Prec*: *l1* $\sqsubset l$ *l2* $\Longrightarrow$ *l2* $\sqsubset l$ *l3* $\Longrightarrow$ *l1* $\sqsubset l$ *l3*
by (*cases l1*; *cases l2*; *cases l3*) *auto*

lemma *wf-L-Prec*: *wfP* ($\sqsubset l$)
by (*metis L-Prec.elims*(*2*) *L-Prec.simps*(*3*) *not-accp-down wfP-accp-iff*)

interpretation *FL*: *given-clause* {{#}} *F-Inf* {{#}} *UNIV* λ*N*. ($\models$*e*) *G-Inf G.Red-I*

```
  λN. G.Red-F λN. 𝒢-F 𝒢-I-opt (≐) (≺·) (⊏l) Old
proof (unfold-locales; (intro ballI)?)
  show equivp (≐)
    unfolding equivp-def by (meson generalizes-refl generalizes-trans)
next
  show po-on (≺·) UNIV
    unfolding po-on-def irreflp-on-def transp-on-def
    using strictly-generalizes-irrefl strictly-generalizes-trans by auto
next
  show wfp-on (≺·) UNIV
    unfolding wfp-on-UNIV by (metis wf-strictly-generalizes)
next
  show po-on (⊏l) UNIV
    unfolding po-on-def irreflp-on-def transp-on-def using irrefl-L-Prec trans-L-Prec by blast
next
  show wfp-on (⊏l) UNIV
    unfolding wfp-on-UNIV by (rule wf-L-Prec)
next
  fix C1 D1 C2 D2
  assume
    C1 ≐ D1
    C2 ≐ D2
    C1 ≺· C2
  then show D1 ≺· D2
    by (smt antisym size-mset-mono size-subst strictly-generalizes-def generalizes-def
        generalizes-trans)
next
  fix N C1 C2
  assume C1 ≐ C2
  then show 𝒢-F C1 ⊆ 𝒢-F C2
    unfolding generalizes-def 𝒢-F-def by clarsimp (metis is-ground-comp-subst subst-cls-comp-subst)
next
  fix N C2 C1
  assume C2 ≺· C1
  then show 𝒢-F C1 ⊆ 𝒢-F C2
    unfolding strictly-generalizes-def generalizes-def 𝒢-F-def
    by clarsimp (metis is-ground-comp-subst subst-cls-comp-subst)
next
  show ∃l. L-Prec Old l
    using L-Prec.simps(1) by blast
qed (auto simp: F-Inf-have-prems)

notation FL.Prec-FL (infix ⊏ 50)
notation FL.entails-𝒢-L (infix ⊨𝒢Le 50)
notation FL.derive (infix ▷L 50)
notation FL.step (infix ⇝GC 50)

lemma FL-Red-F-eq:
  FL.Red-F N =
   {C. ∀D ∈ 𝒢-F (fst C). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ N)) ∨ (∃E ∈ N. E ⊏ C ∧ D ∈ 𝒢-F (fst E))}
  unfolding FL.Red-F-def FL.Red-F-𝒢-q-def by auto

lemma mem-FL-Red-F-because-G-Red-F:
  (∀D ∈ 𝒢-F (fst Cl). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ N))) ⟹ Cl ∈ FL.Red-F N
  unfolding FL-Red-F-eq by auto
```

lemma *mem-FL-Red-F-because-Prec-FL*:
(∀ *D* ∈ *G-F* (*fst Cl*). ∃ *El* ∈ *N*. *El* ⊏ *Cl* ∧ *D* ∈ *G-F* (*fst El*)) ⟹ *Cl* ∈ *FL.Red-F N*
unfolding *FL-Red-F-eq* **by** *auto*

4.6 Resolution Prover Layer

interpretation *sq*: *selection S-Q Sts*
unfolding *S-Q-def* **using** *S-M-selects-subseteq S-M-selects-neg-lits selection-axioms*
by *unfold-locales auto*

interpretation *gd*: *ground-resolution-with-selection S-Q Sts*
by *unfold-locales*

interpretation *src*: *standard-redundancy-criterion-counterex-reducing gd.ord-Γ Sts*
ground-resolution-with-selection.INTERP (*S-Q Sts*)
by *unfold-locales*

definition *lclss-of-state* :: *'a state* ⇒ (*'a clause* × *label*) *set* **where**
lclss-of-state St =
(λ*C*. (*C*, *New*)) ‘ *N-of-state St* ∪ (λ*C*. (*C*, *Processed*)) ‘ *P-of-state St*
∪ (λ*C*. (*C*, *Old*)) ‘ *Q-of-state St*

lemma *image-hd-lclss-of-state*[*simp*]: *fst* ‘ *lclss-of-state St* = *clss-of-state St*
unfolding *lclss-of-state-def* **by** (*auto simp*: *image-Un image-comp*)

lemma *insert-lclss-of-state*[*simp*]:
insert (*C*, *New*) (*lclss-of-state* (*N*, *P*, *Q*)) = *lclss-of-state* (*N* ∪ {*C*}, *P*, *Q*)
insert (*C*, *Processed*) (*lclss-of-state* (*N*, *P*, *Q*)) = *lclss-of-state* (*N*, *P* ∪ {*C*}, *Q*)
insert (*C*, *Old*) (*lclss-of-state* (*N*, *P*, *Q*)) = *lclss-of-state* (*N*, *P*, *Q* ∪ {*C*})
unfolding *lclss-of-state-def image-def* **by** *auto*

lemma *union-lclss-of-state*[*simp*]:
lclss-of-state (*N1*, *P1*, *Q1*) ∪ *lclss-of-state* (*N2*, *P2*, *Q2*) =
lclss-of-state (*N1* ∪ *N2*, *P1* ∪ *P2*, *Q1* ∪ *Q2*)
unfolding *lclss-of-state-def* **by** *auto*

lemma *mem-lclss-of-state*[*simp*]:
(*C*, *New*) ∈ *lclss-of-state* (*N*, *P*, *Q*) ⟷ *C* ∈ *N*
(*C*, *Processed*) ∈ *lclss-of-state* (*N*, *P*, *Q*) ⟷ *C* ∈ *P*
(*C*, *Old*) ∈ *lclss-of-state* (*N*, *P*, *Q*) ⟷ *C* ∈ *Q*
unfolding *lclss-of-state-def image-def* **by** *auto*

lemma *lclss-Liminf-commute*:
Liminf-llist (*lmap lclss-of-state Sts*) = *lclss-of-state* (*Liminf-state Sts*)
proof −
have ‹*Liminf-llist* (*lmap lclss-of-state Sts*) =
(λ*C*. (*C*, *New*)) ‘ *Liminf-llist* (*lmap N-of-state Sts*) ∪
(λ*C*. (*C*, *Processed*)) ‘ *Liminf-llist* (*lmap P-of-state Sts*) ∪
(λ*C*. (*C*, *Old*)) ‘ *Liminf-llist* (*lmap Q-of-state Sts*)›
unfolding *lclss-of-state-def* **using** *Liminf-llist-lmap-union Liminf-llist-lmap-image*
by (*smt Pair-inject Un-iff disjoint-iff-not-equal imageE inj-onI label.simps*(*1*,*3*,*5*)
llist.map-cong)
then show *?thesis*
unfolding *lclss-of-state-def Liminf-state-def* **by** *auto*
qed

lemma *GC-tautology-step*:
assumes *tauto*: *Neg A* ∈# *C Pos A* ∈# *C*
shows *lclss-of-state* (*N* ∪ {*C*}, *P*, *Q*) ⇝*GC lclss-of-state* (*N*, *P*, *Q*)
proof −
have *cϑ-red*: *Cϑ* ∈ *G.Red-F* (⋃ *D* ∈ *N′*. *G-F* (*fst D*)) **if** *in-g*: *Cϑ* ∈ *G-F C*
for *N′* :: (*′a clause* × *label*) *set* **and** *Cϑ*
proof −
obtain *ϑ* **where**
Cϑ = *C* · *ϑ*
using *in-g* **unfolding** *G-F-def* **by** *blast*
then have *Neg* (*A* ·*a ϑ*) ∈# *Cϑ* **and** *Pos* (*A* ·*a ϑ*) ∈# *Cϑ*
using *tauto Neg-Melem-subst-atm-subst-cls Pos-Melem-subst-atm-subst-cls* **by** *auto*
then have {} ⊨*e* {*Cϑ*}
unfolding *true-clss-def true-cls-def true-lit-def if-distrib-fun*
by (*metis literal.disc literal.sel singletonD*)
then show *?thesis*
unfolding *G.Red-F-def* **by** *auto*
qed

show *?thesis*
proof (*rule FL.step.process*[*of - lclss-of-state* (*N*, *P*, *Q*) {(*C*, *New*)} - {}])
show ‹{(*C*, *New*)} ⊆ *FL.Red-F-G* (*lclss-of-state* (*N*, *P*, *Q*) ∪ {})›
using *mem-FL-Red-F-because-G-Red-F cϑ-red*[*of - lclss-of-state* (*N*, *P*, *Q*)]
unfolding *lclss-of-state-def* **by** *auto*
qed (*auto simp*: *lclss-of-state-def FL.active-subset-def*)
qed

lemma *GC-subsumption-step*:
assumes
d-in: *Dl* ∈ *N* **and**
d-sub-c: *strictly-subsumes* (*fst Dl*) (*fst Cl*) ∨ *subsumes* (*fst Dl*) (*fst Cl*) ∧ *snd Dl* ⊏*l snd Cl*
shows *N* ∪ {*Cl*} ⇝*GC N*
proof −
have *d-sub′-c*: *Cl* ∈ *FL.Red-F* {*Dl*} ∨ *Dl* ⊏ *Cl*
proof (*cases size* (*fst Dl*) = *size* (*fst Cl*))
case *True*
assume *sizes-eq*: ‹*size* (*fst Dl*) = *size* (*fst Cl*)›
have ‹*size* (*fst Dl*) = *size* (*fst Cl*) ⟹
strictly-subsumes (*fst Dl*) (*fst Cl*) ∨ *subsumes* (*fst Dl*) (*fst Cl*) ∧ *snd Dl* ⊏*l snd Cl* ⟹
Dl ⊏ *Cl*›
unfolding *FL.Prec-FL-def*
unfolding *generalizes-def strictly-generalizes-def strictly-subsumes-def subsumes-def*
by (*metis size-subst subset-mset.order-refl subseteq-mset-size-eql*)
then have *Dl* ⊏ *Cl*
using *sizes-eq d-sub-c* **by** *auto*
then show *?thesis*
by (*rule disjI2*)
next
case *False*
then have *d-ssub-c*: *strictly-subsumes* (*fst Dl*) (*fst Cl*)
using *d-sub-c* **unfolding** *strictly-subsumes-def subsumes-def*
by (*metis size-subst strict-subset-subst-strictly-subsumes strictly-subsumes-antisym*
subset-mset.antisym-conv2)
have *Cl* ∈ *FL.Red-F* {*Dl*}

```
    proof (rule mem-FL-Red-F-because-G-Red-F)
      show ‹∀ D ∈ 𝒢-F (fst Cl). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ {Dl}))›
        using d-ssub-c unfolding G.Red-F-def strictly-subsumes-def subsumes-def 𝒢-F-def
      proof clarsimp
        fix σ σ'
        assume
          fst-not-in: ‹∀ σ. ¬ fst Cl · σ ⊆# fst Dl› and
          fst-in: ‹fst Dl · σ ⊆# fst Cl› and
          gr-sig: ‹is-ground-subst σ'›
        have ‹{fst Dl · σ · σ'} ⊆ {fst Dl · σ |σ. is-ground-subst σ}›
          using gr-sig
          by (metis (mono-tags, lifting) is-ground-comp-subst mem-Collect-eq singletonD subsetI
              subst-cls-comp-subst)
        moreover have ‹∀ I. I ⊫s {fst Dl · σ · σ'} ⟶ I ⊫ fst Cl · σ'›
          using fst-in
          by (meson subst-cls-mono-mset true-clss-insert true-clss-subclause)
        moreover have ‹∀ D ∈ {fst Dl · σ · σ'}. D < fst Cl · σ'›
          using fst-not-in fst-in gr-sig
        proof clarify
          show ‹∀ σ. ¬ fst Cl · σ ⊆# fst Dl ⟹ fst Dl · σ ⊆# fst Cl ⟹ is-ground-subst σ' ⟹
            fst Dl · σ · σ' < fst Cl · σ'›
            by (metis False size-subst subset-imp-less-mset subset-mset.le-less subst-subset-mono)
        qed
        ultimately show ‹∃ DD ⊆ {fst Dl · σ |σ. is-ground-subst σ}.
          (∀ I. I ⊫s DD ⟶ I ⊫ fst Cl · σ') ∧ (∀ D ∈ DD. D < fst Cl · σ')›
          by blast
      qed
    qed
    then show ?thesis
      by (rule disjI1)
  qed
  show ?thesis
  proof (rule FL.step.process[of - N {Cl} - {}], simp+)
    show ‹Cl ∈ FL.Red-F-𝒢 N›
      using d-sub'-c unfolding FL-Red-F-eq
    proof −
      have ‹⋀D. D ∈ 𝒢-F (fst Cl) ⟹ ∀ E ∈ N. E ⊏ Cl ⟶ D ∉ 𝒢-F (fst E) ⟹
        ∀ D ∈ 𝒢-F (fst Cl). D ∈ G.Red-F (𝒢-F (fst Dl)) ∨ Dl ⊏ Cl ∧ D ∈ 𝒢-F (fst Dl) ⟹
        D ∈ G.Red-F (⋃ a ∈ N. 𝒢-F (fst a))›
        by (metis (no-types, lifting) G.Red-F-of-subset SUP-upper d-in subset-iff)
      moreover have ‹⋀D. D ∈ 𝒢-F (fst Cl) ⟹ ∀ E ∈ N. E ⊏ Cl ⟶ D ∉ 𝒢-F (fst E) ⟹ Dl ⊏ Cl
⟹
        D ∈ G.Red-F (⋃ a ∈ N. 𝒢-F (fst a))›
        by (smt FL.Prec-FL-def FL.equiv-F-grounding FL.prec-F-grounding UNIV-witness d-in in-mono)
      ultimately show ‹Cl ∈ {C. ∀ D ∈ 𝒢-F (fst C). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ {Dl})) ∨
        (∃ E ∈ {Dl}. E ⊏ C ∧ D ∈ 𝒢-F (fst E))} ∨ Dl ⊏ Cl ⟹
        Cl ∈ {C. ∀ D ∈ 𝒢-F (fst C). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ N)) ∨
        (∃ E ∈ N. E ⊏ C ∧ D ∈ 𝒢-F (fst E))}›
        by auto
    qed
  qed (simp add: FL.active-subset-def)
qed

lemma GC-reduction-step:
  assumes
```

```
    young: snd Dl ≠ Old and
    d-sub-c: fst Dl ⊂# fst Cl
  shows N ∪ {Cl} ⇝GC N ∪ {Dl}
proof (rule FL.step.process[of - N {Cl} - {Dl}])
  have Cl ∈ FL.Red-F {Dl}
  proof (rule mem-FL-Red-F-because-G-Red-F)
    show ‹∀ D ∈ 𝒢-F (fst Cl). D ∈ G.Red-F (⋃ (𝒢-F ‘ fst ‘ {Dl}))›
      using d-sub-c unfolding G.Red-F-def strictly-subsumes-def subsumes-def 𝒢-F-def
    proof clarsimp
      fix σ
      assume ‹is-ground-subst σ›
      then have ‹{fst Dl · σ} ⊆ {fst Dl · σ |σ. is-ground-subst σ}›
        by blast
      moreover have ‹fst Dl · σ < fst Cl · σ›
        using subst-subset-mono[OF d-sub-c, of σ] by (simp add: subset-imp-less-mset)
      moreover have ‹∀ I. I ⊨ fst Dl · σ ⟶ I ⊨ fst Cl · σ›
        using subst-subset-mono[OF d-sub-c] true-clss-subclause by fast
      ultimately show ‹∃ DD ⊆ {fst Dl · σ |σ. is-ground-subst σ}. (∀ I. I ⊨s DD ⟶ I ⊨ fst Cl · σ)
        ∧ (∀ D ∈ DD. D < fst Cl · σ)›
        by blast
    qed
  qed
  then show {Cl} ⊆ FL.Red-F (N ∪ {Dl})
    using FL.Red-F-of-subset by blast
qed (auto simp: FL.active-subset-def young)

lemma GC-processing-step: N ∪ {(C, New)} ⇝GC N ∪ {(C, Processed)}
proof (rule FL.step.process[of - N {(C, New)} - {(C, Processed)}])
  have (C, New) ∈ FL.Red-F {(C, Processed)}
    by (rule mem-FL-Red-F-because-Prec-FL) (simp add: FL.Prec-FL-def generalizes-refl)
  then show {(C, New)} ⊆ FL.Red-F (N ∪ {(C, Processed)})
    using FL.Red-F-of-subset by blast
qed (auto simp: FL.active-subset-def)

lemma old-inferences-between-eq-new-inferences-between:
  old-concl-of ‘ inference-system.inferences-between (ord-FO-Γ S) N C =
   concl-of ‘ F.Inf-between N {C} (is ?rp = ?f)
proof (intro set-eqI iffI)
  fix E
  assume e-in: E ∈ old-concl-of ‘ inference-system.inferences-between (ord-FO-Γ S) N C

  obtain CAs DA AAs As σ where
    e-res: ord-resolve-rename S CAs DA AAs As σ E and
    cd-sub: set CAs ∪ {DA} ⊆ N ∪ {C} and
    c-in: C ∈ set CAs ∪ {DA}
    using e-in unfolding inference-system.inferences-between-def infer-from-def ord-FO-Γ-def by auto

  show E ∈ concl-of ‘ F.Inf-between N {C}
    unfolding F.Inf-between-alt F.Inf-from-def
  proof −
    have ‹Infer (CAs @ [DA]) E ∈ F-Inf ∧ set (prems-of (Infer (CAs @ [DA]) E)) ⊆ insert C N ∧
      C ∈ set (prems-of (Infer (CAs @ [DA]) E)) ∧ E = concl-of (Infer (CAs @ [DA]) E)›
      using e-res cd-sub c-in F-Inf-def by auto
    then show ‹E ∈ concl-of ‘ {ι ∈ F-Inf. ι ∈ {ι ∈ F-Inf. set (prems-of ι) ⊆ N ∪ {C}} ∧
      set (prems-of ι) ∩ {C} ≠ {}}›
```

by (*smt Un-insert-right boolean-algebra-cancel.sup0 disjoint-insert mem-Collect-eq image-def*)
qed
next
fix *E*
assume *e-in*: $E \in$ *concl-of* ‘ *F.Inf-between N* $\{C\}$

obtain *CAs DA AAs As* σ **where**
e-res: *ord-resolve-rename S CAs DA AAs As* σ *E* **and**
cd-sub: *set CAs* $\cup$ $\{DA\} \subseteq N \cup \{C\}$ **and**
c-in: $C \in$ *set CAs* $\cup$ $\{DA\}$
using *e-in* **unfolding** *F.Inf-between-alt F.Inf-from-def F-Inf-def inference-system.Inf-between-alt inference-system.Inf-from-def*
by (*auto simp*: *image-def Bex-def*)

show $E \in$ *old-concl-of* ‘ *inference-system.inferences-between* (*ord-FO*-Γ *S*) *N C*
unfolding *inference-system.inferences-between-def infer-from-def ord-FO*-Γ*-def*
using *e-res cd-sub c-in*
by (*clarsimp simp*: *image-def Bex-def*, *rule-tac x* = *old-Infer* (*mset CAs*) *DA E* **in** *exI*, *auto*)
qed

lemma *GC-inference-step*:
assumes
young: $l \neq$ *Old* **and**
no-active: *FL.active-subset M* = $\{\}$ **and**
m-sup: *fst* ‘ *M* $\supseteq$ *old-concl-of* ‘ *inference-system.inferences-between* (*ord-FO*-Γ *S*) (*fst* ‘ *FL.active-subset N*) *C*
shows $N \cup \{(C, l)\} \rightsquigarrow_{GC} N \cup \{(C, Old)\} \cup M$
proof (*rule FL.step.infer*[*of - N C l - M*])
have *m-sup'*: *fst* ‘ *M* $\supseteq$ *concl-of* ‘ *F.Inf-between* (*fst* ‘ *FL.active-subset N*) $\{C\}$
using *m-sup* **unfolding** *old-inferences-between-eq-new-inferences-between* **.**

show *F.Inf-between* (*fst* ‘ *FL.active-subset N*) $\{C\} \subseteq$ *F.Red-I* (*fst* ‘ $(N \cup \{(C, Old)\} \cup M)$)
proof
fix ι
assume ι*-in-if2*: $\iota \in$ *F.Inf-between* (*fst* ‘ *FL.active-subset N*) $\{C\}$
note ι*-in* = *F.Inf-if-Inf-between*[*OF* ι*-in-if2*]
have *concl-of* $\iota \in$ *fst* ‘ *M*
using *m-sup'* ι*-in-if2 m-sup'* **by** (*auto simp*: *image-def Collect-mono-iff F.Inf-between-alt*)
then have *concl-of* $\iota \in$ *fst* ‘ $(N \cup \{(C, Old)\} \cup M)$
by *auto*
then show $\iota \in$ *F.Red-I-*$\mathcal{G}$ (*fst* ‘ $(N \cup \{(C, Old)\} \cup M)$)
by (*rule F.Red-I-of-Inf-to-N*[*OF* ι*-in*])
qed
qed (*use young no-active* **in** *auto*)

lemma *RP-step-imp-GC-step*: *St* $\rightsquigarrow_{RP}$ *St'* $\Longrightarrow$ *lclss-of-state St* $\rightsquigarrow_{GC}$ *lclss-of-state St'*
proof (*induction rule*: *RP.induct*)
case (*tautology-deletion A C N P Q*)
then show *?case*
by (*rule GC-tautology-step*)
next
case (*forward-subsumption D P Q C N*)
note *d-in* = *this*(*1*) **and** *d-sub-c* = *this*(*2*)
show *?case*
proof (*cases D* $\in$ *P*)

```
    case True
    then show ?thesis
      using GC-subsumption-step[of (D, Processed) lclss-of-state (N, P, Q) (C, New)] d-sub-c
      by auto
  next
    case False
    then have D ∈ Q
      using d-in by simp
    then show ?thesis
      using GC-subsumption-step[of (D, Old) lclss-of-state (N, P, Q) (C, New)] d-sub-c by auto
  qed
next
  case (backward-subsumption-P D N C P Q)
  note d-in = this(1) and d-ssub-c = this(2)
  then show ?case
    using GC-subsumption-step[of (D, New) lclss-of-state (N, P, Q) (C, Processed)] d-ssub-c
    by auto
next
  case (backward-subsumption-Q D N C P Q)
  note d-in = this(1) and d-ssub-c = this(2)
  then show ?case
    using GC-subsumption-step[of (D, New) lclss-of-state (N, P, Q) (C, Old)] d-ssub-c by auto
next
  case (forward-reduction D L' P Q L σ C N)
  show ?case
    using GC-reduction-step[of (C, New) (C + {#L#}, New) lclss-of-state (N, P, Q)] by auto
next
  case (backward-reduction-P D L' N L σ C P Q)
  show ?case
    using GC-reduction-step[of (C, Processed) (C + {#L#}, Processed) lclss-of-state (N, P, Q)]
    by auto
next
  case (backward-reduction-Q D L' N L σ C P Q)
  show ?case
    using GC-reduction-step[of (C, Processed) (C + {#L#}, Old) lclss-of-state (N, P, Q)]
    by auto
next
  case (clause-processing N C P Q)
  show ?case
    using GC-processing-step[of lclss-of-state (N, P, Q) C] by auto
next
  case (inference-computation N Q C P)
  note n = this(1)
  show ?case
  proof −
    have ‹FL.active-subset (lclss-of-state (N, {}, {})) = {}›
      unfolding n by (auto simp: FL.active-subset-def)
    moreover have ‹old-concls-of (inference-system.inferences-between (ord-FO-Γ S)
      (fst ‘ FL.active-subset (lclss-of-state ({}, P, Q))) C) ⊆ N›
      unfolding n inference-system.inferences-between-def image-def mem-Collect-eq
        lclss-of-state-def infer-from-def
      by (auto simp: FL.active-subset-def)
    ultimately have ‹lclss-of-state ({}, insert C P, Q) ⇝GC lclss-of-state (N, P, insert C Q)›
      using GC-inference-step[of Processed lclss-of-state (N, {}, {})
        lclss-of-state ({}, P, Q) C, simplified] by blast
```

```
    then show ?case
      by (auto simp: FL.active-subset-def)
  qed
qed

lemma RP-derivation-imp-GC-derivation: chain (⇝RP) Sts ⟹ chain (⇝GC) (lmap lclss-of-state Sts)
  using chain-lmap RP-step-imp-GC-step by blast

lemma RP-step-imp-derive-step: St ⇝RP St' ⟹ lclss-of-state St ▷L lclss-of-state St'
  by (rule FL.one-step-equiv) (rule RP-step-imp-GC-step)

lemma RP-derivation-imp-derive-derivation:
  chain (⇝RP) Sts ⟹ chain (▷L) (lmap lclss-of-state Sts)
  using chain-lmap RP-step-imp-derive-step by blast

theorem RP-sound-new-statement:
  assumes
    deriv: chain (⇝RP) Sts and
    bot-in: {#} ∈ clss-of-state (Liminf-state Sts)
  shows clss-of-state (lhd Sts) ⊨𝒢e {{#}}
proof −
  have clss-of-state (Liminf-state Sts) ⊨𝒢e {{#}}
    using F.subset-entailed bot-in by auto
  then have fst ‘ Liminf-llist (lmap lclss-of-state Sts) ⊨𝒢e {{#}}
    by (metis image-hd-lclss-of-state lclss-Liminf-commute)
  then have Liminf-llist (lmap lclss-of-state Sts) ⊨𝒢Le FL.Bot-FL
    using FL.labeled-entailment-lifting by simp
  then have lhd (lmap lclss-of-state Sts) ⊨𝒢Le FL.Bot-FL
  proof −
    assume ‹FL.entails-𝒢 (Liminf-llist (lmap lclss-of-state Sts)) ({{#}} × UNIV)›
    moreover have ‹chain (▷L) (lmap lclss-of-state Sts)›
      using deriv RP-derivation-imp-derive-derivation by simp
    moreover have ‹chain FL.entails-𝒢 (lmap lclss-of-state Sts)›
      by (smt F-entails-𝒢-iff FL.labeled-entailment-lifting RP-model chain-lmap deriv 𝒢-Fset-def
        image-hd-lclss-of-state)
    ultimately show ‹FL.entails-𝒢 (lhd (lmap lclss-of-state Sts)) ({{#}} × UNIV)›
      using FL.unsat-limit-iff by blast
  qed
  then have lclss-of-state (lhd Sts) ⊨𝒢Le FL.Bot-FL
    using chain-not-lnull deriv by fastforce
  then show ?thesis
    unfolding FL.entails-𝒢-L-def F.entails-𝒢-def lclss-of-state-def by auto
qed

theorem RP-saturated-if-fair-new-statement:
  assumes
    deriv: chain (⇝RP) Sts and
    init: FL.active-subset (lclss-of-state (lhd Sts)) = {} and
    final: FL.passive-subset (Liminf-llist (lmap lclss-of-state Sts)) = {}
  shows FL.saturated (Liminf-llist (lmap lclss-of-state Sts))
proof −
  note nnil = chain-not-lnull[OF deriv]
  have gc-deriv: chain (⇝GC) (lmap lclss-of-state Sts)
    by (rule RP-derivation-imp-GC-derivation[OF deriv])
  show ?thesis
```

using *nnil init final*
FL.fair-implies-Liminf-saturated[*OF FL.gc-to-red*[*OF gc-deriv*] *FL.gc-fair*[*OF gc-deriv*]] **by** *simp*
qed

corollary *RP-complete-if-fair-new-statement*:
assumes
deriv: *chain* (⇝*RP*) *Sts* **and**
init: *FL.active-subset* (*lclss-of-state* (*lhd Sts*)) = {} **and**
final: *FL.passive-subset* (*Liminf-llist* (*lmap lclss-of-state Sts*)) = {} **and**
unsat: *grounding-of-state* (*lhd Sts*) ⊨*e* {{#}}
shows {#} ∈ *Q-of-state* (*Liminf-state Sts*)
proof −
note *nnil* = *chain-not-lnull*[*OF deriv*]
note *len* = *chain-length-pos*[*OF deriv*]
have *gc-deriv*: *chain* (⇝*GC*) (*lmap lclss-of-state Sts*)
by (*rule RP-derivation-imp-GC-derivation*[*OF deriv*])

have *hd-lcls*: *fst* ‘ *lhd* (*lmap lclss-of-state Sts*) = *lhd* (*lmap clss-of-state Sts*)
using *len zero-enat-def* **by** *auto*
have *hd-unsat*: *fst* ‘ *lhd* (*lmap lclss-of-state Sts*) ⊨*G*e {{#}}
unfolding *hd-lcls F-entails-G-iff* **unfolding** *true-clss-def* **using** *unsat* **unfolding** *G-Fset-def*
by (*metis* (*no-types*, *lifting*) *chain-length-pos gc-deriv gr.ex-min-counterex i0-less*
llength-eq-0 llength-lmap llength-lmap llist.map-sel(*1*) *true-cls-empty true-clss-singleton*)
have ∃ *BL* ∈ {{#}} × *UNIV*. *BL* ∈ *Liminf-llist* (*lmap lclss-of-state Sts*)
by (*rule FL.gc-complete-Liminf*[*OF gc-deriv*,*of* {#}])
(*use final hd-unsat* **in** ‹*auto simp*: *init nnil*›)
then show *?thesis*
unfolding *Liminf-state-def lclss-Liminf-commute*
using *final*[*unfolded FL.passive-subset-def*] *Liminf-state-def lclss-Liminf-commute* **by** *fastforce*
qed

4.7 Alternative Derivation of Previous RP Results

lemma *old-fair-imp-new-fair*:
assumes
nnul: ¬ *lnull Sts* **and**
fair: *fair-state-seq Sts* **and**
empty-Q0: *Q-of-state* (*lhd Sts*) = {}
shows
FL.active-subset (*lclss-of-state* (*lhd Sts*)) = {} **and**
FL.passive-subset (*Liminf-llist* (*lmap lclss-of-state Sts*)) = {}
proof −
show *FL.active-subset* (*lclss-of-state* (*lhd Sts*)) = {}
using *nnul empty-Q0* **unfolding** *FL.active-subset-def* **by** (*cases Sts*) *auto*
next
show *FL.passive-subset* (*Liminf-llist* (*lmap lclss-of-state Sts*)) = {}
using *fair*
unfolding *fair-state-seq-def FL.passive-subset-def lclss-Liminf-commute lclss-of-state-def*
by *auto*
qed

lemma *old-redundant-infer-iff*:
src.redundant-infer N γ ⟷
(∃ *DD*. *DD* ⊆ *N* ∧ *DD* ∪ *set-mset* (*old-side-prems-of* γ) ⊨*e* {*old-concl-of* γ}
∧ (∀ *D* ∈ *DD*. *D* < *old-main-prem-of* γ))
(**is** *?lhs* ⟷ *?rhs*)

proof
  assume *?rhs*
  then obtain *DD0* **where**
    DD0 $\subseteq$ *N* **and**
    DD0 $\cup$ *set-mset* (*old-side-prems-of* γ) $\models e$ {*old-concl-of* γ} **and**
    $\forall$ *D* $\in$ *DD0*. *D* $<$ *old-main-prem-of* γ
    by *blast*
  then obtain *DD* **where**
    fin-dd: *finite DD* **and**
    dd-in: *DD* $\subseteq$ *N* **and**
    dd-un: *DD* $\cup$ *set-mset* (*old-side-prems-of* γ) $\models e$ {*old-concl-of* γ} **and**
    all-dd: $\forall$ *D* $\in$ *DD*. *D* $<$ *old-main-prem-of* γ
    using *entails-concl-compact-union*[*of* {*old-concl-of* γ} *DD0 set-mset* (*old-side-prems-of* γ)]
    by *fast*
  show *?lhs*
    unfolding *src.redundant-infer-def* **using** *fin-dd dd-in dd-un all-dd*
    by *simp* (*metis finite-set-mset-mset-set true-clss-set-mset*)
qed (*auto simp*: *src.redundant-infer-def*)

definition *old-infer-of* :: *'a clause inference* $\Rightarrow$ *'a old-inference* **where**
  old-infer-of ι = *old-Infer* (*mset* (*side-prems-of* ι)) (*main-prem-of* ι) (*concl-of* ι)

lemma *new-redundant-infer-imp-old-redundant-infer*:
  G.redundant-infer N $\iota \Longrightarrow$ *src.redundant-infer N* (*old-infer-of* ι)
  unfolding *old-redundant-infer-iff G.redundant-infer-def old-infer-of-def* **by** *simp*

lemma *saturated-imp-saturated-RP*:
  assumes
    satur: *FL.saturated* (*Liminf-llist* (*lmap lclss-of-state Sts*)) **and**
    no-passive: *FL.passive-subset* (*Liminf-llist* (*lmap lclss-of-state Sts*)) = {}
  shows *src.saturated-upto Sts* (*grounding-of-state* (*Liminf-state Sts*))
proof −
  define *Q* **where**
    Q = *Liminf-llist* (*lmap Q-of-state Sts*)
  define *Ql* **where**
    Ql = (λ*C*. (*C*, *Old*)) ‘ *Q*
  define *G* **where**
    G = $\bigcup$ (*G*-*F* ‘ *Q*)

  have *n-empty*: *N-of-state* (*Liminf-state Sts*) = {} **and**
    p-empty: *P-of-state* (*Liminf-state Sts*) = {}
    using *no-passive*[*unfolded FL.passive-subset-def*] *Liminf-state-def lclss-Liminf-commute*
    by *fastforce*+
  then have *limuls-eq*: *Liminf-llist* (*lmap lclss-of-state Sts*) = *Ql*
    unfolding *Ql-def Q-def* **using** *Liminf-state-def lclss-Liminf-commute lclss-of-state-def* **by** *auto*
  have *clst-eq*: *clss-of-state* (*Liminf-state Sts*) = *Q*
    unfolding *n-empty p-empty Q-def* **by** (*auto simp*: *Liminf-state-def*)
  have *gflimuls-eq*: ($\bigcup$ *Cl* $\in$ *Ql*. *G*-*F* (*fst Cl*)) = *G*
    unfolding *Ql-def G-def* **by** *auto*

  have *gd.inferences-from Sts G* $\subseteq$ *src.Ri Sts G*
  proof
    fix γ
    assume γ-*inf*: γ $\in$ *gd.inferences-from Sts G*

obtain ι **where**
  ι*-inff*: $\iota \in$ *G.Inf-from Q G* **and**
  γ: $\gamma =$ *old-infer-of* ι
  using γ*-inf*
  unfolding *gd.inferences-from-def old-infer-from-def G.Inf-from-def old-infer-of-def*
proof (*atomize-elim*, *clarify*)
  assume
    g-is: ‹$\gamma \in$ *gd.ord-*Γ *Sts*› **and**
    prems-in: ‹*set-mset* (*old-side-prems-of* γ + {#*old-main-prem-of* γ#}) $\subseteq$ *G*›
  obtain *CAs DA AAs As E* **where** *main-in*: ‹*DA* $\in$ *G*› **and** *side-in*: ‹*set CAs* $\subseteq$ *G*› **and**
    g-is2: ‹$\gamma =$ *old-Infer* (*mset CAs*) *DA E*› **and**
    ord-res: ‹*gd.ord-resolve Sts CAs DA AAs As E*›
    using *g-is prems-in* **unfolding** *gd.ord-*Γ*-def* **by** *auto*
  define ι-γ **where** ι-$\gamma =$ *Infer* (*CAs* @ [*DA*]) *E*
  then have ‹ι-$\gamma \in$ *G-Inf Q*› **using** *Q-of-state.simps g-is g-is2 ord-res Liminf-state-def S-Q-def*
    unfolding *gd.ord-*Γ*-def G-Inf-def Q-def* **by** *auto*
  moreover have ‹*set* (*prems-of* ι-γ) $\subseteq$ *G*›
    using *g-is2 prems-in* **unfolding** ι-γ*-def* **by** *simp*
  moreover have ‹$\gamma =$ *old-Infer* (*mset* (*side-prems-of* ι-γ)) (*main-prem-of* ι-γ) (*concl-of* ι-γ)›
    using *g-is2* **unfolding** ι-γ*-def* **by** *simp*
  ultimately show
    ‹$\exists \iota.\ \iota \in \{\iota \in$ *G-Inf Q*. *set* (*prems-of* ι) $\subseteq$ *G*$\} \wedge \gamma =$ *old-Infer* (*mset* (*side-prems-of* ι))
     (*main-prem-of* ι) (*concl-of* ι)›
    by *blast*
qed
obtain ι' **where**
  ι'*-inff*: $\iota' \in$ *F.Inf-from Q* **and**
  ι*-in-*ι': $\iota \in \mathcal{G}$*-I Q* ι'
  using *G-Inf-overapprox-F-Inf* ι*-inff* **unfolding** *G-def* **by** *blast*

note ι'*-inf* = *F.Inf-if-Inf-from*[*OF* ι'*-inff*]

let *?olds* = *replicate* (*length* (*prems-of* ι')) *Old*

obtain ι'' **and** *l0* **where**
  ι'': $\iota'' =$ *Infer* (*zip* (*prems-of* ι') *?olds*) (*concl-of* ι', *l0*) **and**
  ι''*-inf*: $\iota'' \in$ *FL.Inf-FL*
  using *FL.Inf-F-to-Inf-FL*[*OF* ι'*-inf*, *of ?olds*, *simplified*] **by** *simp*

have *set* (*prems-of* ι'') $\subseteq$ *Ql*
  using ι'*-inff*[*unfolded F.Inf-from-def*, *simplified*] **unfolding** ι'' *Ql-def* **by** *auto*
then have $\iota'' \in$ *FL.Inf-from Ql*
  unfolding *FL.Inf-from-def* **using** ι''*-inf* **by** *simp*
moreover have $\iota' =$ *FL.to-F* ι''
  unfolding ι'' **unfolding** *FL.to-F-def* **by** *simp*
ultimately have $\iota \in$ *G.Red-I Q G*
  using ι*-in-*ι'
    FL.sat-inf-imp-ground-red-fam-inter[*OF satur*, *unfolded limuls-eq gflimuls-eq*, *simplified*]
  by *blast*
then have *G.redundant-infer G* ι
  unfolding *G.Red-I-def* **by** *auto*
then have γ*-red*: *src.redundant-infer G* γ
  unfolding γ **by** (*rule new-redundant-infer-imp-old-redundant-infer*)
moreover have $\gamma \in$ *gd.ord-*Γ *Sts*
  using γ*-inf gd.inferences-from-def* **by** *blast*

ultimately show $\gamma \in$ *src.Ri Sts G*
unfolding *src.Ri-def* **by** *auto*
qed
then show *?thesis*
unfolding *G-def clst-eq src.saturated-upto-def*
by *clarsimp* (*smt Diff-subset gd.inferences-from-mono subset-eq* $\mathcal{G}$*-Fset-def*)
qed

theorem *RP-sound-old-statement*:
assumes
deriv: *chain* ($\rightsquigarrow$*RP*) *Sts* **and**
bot-in: $\{\#\} \in$ *clss-of-state* (*Liminf-state Sts*)
shows $\neg$ *satisfiable* (*grounding-of-state* (*lhd Sts*))
using *RP-sound-new-statement*[*OF deriv bot-in*] **unfolding** *F-entails-*$\mathcal{G}$*-iff* $\mathcal{G}$*-Fset-def* **by** *simp*

The theorem below is stated differently than the original theorem in RP: The grounding of the limit might be a strict subset of the limit of the groundings. Because saturation is neither monotone nor antimonotone, the two results are incomparable. See also *grounding-of-state-Liminf-state-subseteq*.

theorem *RP-saturated-if-fair-old-statement-altered*:
assumes
deriv: *chain* ($\rightsquigarrow$*RP*) *Sts* **and**
fair: *fair-state-seq Sts* **and**
empty-Q0: *Q-of-state* (*lhd Sts*) = {}
shows *src.saturated-upto Sts* (*grounding-of-state* (*Liminf-state Sts*))
proof −
note *fair'* = *old-fair-imp-new-fair*[*OF chain-not-lnull*[*OF deriv*] *fair empty-Q0*]
show *?thesis*
by (*rule saturated-imp-saturated-RP*[*OF - fair'*(*2*)], *rule RP-saturated-if-fair-new-statement*)
(*rule deriv fair'*)+
qed

corollary *RP-complete-if-fair-old-statement*:
assumes
deriv: *chain* ($\rightsquigarrow$*RP*) *Sts* **and**
fair: *fair-state-seq Sts* **and**
empty-Q0: *Q-of-state* (*lhd Sts*) = {} **and**
unsat: $\neg$ *satisfiable* (*grounding-of-state* (*lhd Sts*))
shows $\{\#\} \in$ *Q-of-state* (*Liminf-state Sts*)
proof (*rule RP-complete-if-fair-new-statement*)
show ‹$\mathcal{G}$*-Fset* (*N-of-state* (*lhd Sts*) ∪ *P-of-state* (*lhd Sts*) ∪ *Q-of-state* (*lhd Sts*)) $\models e$ {{#}}›
using *unsat* **unfolding** *F-entails-*$\mathcal{G}$*-iff* **by** *auto*
qed (*rule deriv old-fair-imp-new-fair*[*OF chain-not-lnull*[*OF deriv*] *fair empty-Q0*])+

end

end

5 New Fairness Proofs for the Given Clause Prover Architectures

theory *Given-Clause-Architectures-Revisited*
imports *Saturation-Framework.Given-Clause-Architectures*
begin

The given clause and lazy given clause procedures satisfy key invariants. This provides an

alternative way to prove fairness and hence saturation of the limit.

5.1 Given Clause Procedure

context *given-clause*
begin

definition *gc-invar* :: ('*f* × '*l*) *set llist* ⇒ *enat* ⇒ *bool* **where**
gc-invar Ns i ⟷
Inf-from (*active-subset* (*Liminf-upto-llist Ns i*)) ⊆ *Sup-upto-llist* (*lmap Red-I-G Ns*) *i*

lemma *gc-invar-infinity*:
assumes
nnil: ¬ *lnull Ns* **and**
invar: ∀ *i*. *enat i* < *llength Ns* ⟶ *gc-invar Ns* (*enat i*)
shows *gc-invar Ns* ∞
unfolding *gc-invar-def*
proof (*intro subsetI*, *unfold Liminf-upto-llist-infinity Sup-upto-llist-infinity*)
fix ι
assume ι*-inff*: ι ∈ *Inf-from* (*active-subset* (*Liminf-llist Ns*))

define *As* **where**
As = *lmap active-subset Ns*

have *act-ns*: *active-subset* (*Liminf-llist Ns*) = *Liminf-llist As*
unfolding *As-def active-subset-def Liminf-set-filter-commute*[*symmetric*] **..**

note ι*-inf* = *Inf-if-Inf-from*[*OF* ι*-inff*]
note ι*-inff'* = ι*-inff*[*unfolded act-ns*]

have ¬ *lnull As*
unfolding *As-def* **using** *nnil* **by** *auto*
moreover have *set* (*prems-of* ι) ⊆ *Liminf-llist As*
using ι*-inff'*[*unfolded Inf-from-def*] **by** *simp*
ultimately obtain *i* **where**
i-lt-as: *enat i* < *llength As* **and**
prems-sub-ge-i: *set* (*prems-of* ι) ⊆ (⋂ *j* ∈ {*j*. *i* ≤ *j* ∧ *enat j* < *llength As*}. *lnth As j*)
using *finite-subset-Liminf-llist-imp-exists-index* **by** *blast*

note *i-lt-ns* = *i-lt-as*[*unfolded As-def*, *simplified*]

have *set* (*prems-of* ι) ⊆ *lnth As i*
using *prems-sub-ge-i i-lt-as* **by** *auto*
then have ι ∈ *Inf-from* (*active-subset* (*lnth Ns i*))
using *i-lt-as* ι*-inf* **unfolding** *Inf-from-def As-def* **by** *auto*
then have ι ∈ *Sup-upto-llist* (*lmap Red-I-G Ns*) (*enat i*)
using *nnil i-lt-ns invar*[*unfolded gc-invar-def*] **by** *auto*
then show ι ∈ *Sup-llist* (*lmap Red-I-G Ns*)
using *Sup-upto-llist-subset-Sup-llist* **by** *fastforce*
qed

lemma *gc-invar-gc-init*:
assumes
¬ *lnull Ns* **and**
active-subset (*lhd Ns*) = {}

shows *gc-invar Ns 0*
using *assms labeled-inf-have-prems* **unfolding** *gc-invar-def Inf-from-def* **by** *auto*

lemma *gc-invar-gc-step*:
assumes
Si-lt: *enat* (*Suc i*) < *llength Ns* **and**
invar: *gc-invar Ns i* **and**
step: *lnth Ns i* ⇝*GC lnth Ns* (*Suc i*)
shows *gc-invar Ns* (*Suc i*)
using *step Si-lt invar*
proof *cases*
have *i-lt*: *enat i* < *llength Ns*
using *Si-lt Suc-ile-eq order.strict-implies-order* **by** *blast*
have *lim-i*: *Liminf-upto-llist Ns* (*enat i*) = *lnth Ns i*
using *i-lt* **by** *auto*
have *lim-Si*: *Liminf-upto-llist Ns* (*enat* (*Suc i*)) = *lnth Ns* (*Suc i*)
using *Si-lt* **by** *auto*

{
case (*process N M M′*)
note *ni* = *this*(*1*) **and** *nSi* = *this*(*2*) **and** *m′-pas* = *this*(*4*)

have *Inf-from* (*active-subset* (*N* ∪ *M′*)) ⊆ *Inf-from* (*active-subset* (*N* ∪ *M*))
using *m′-pas* **by** (*simp add*: *Inf-from-mono*)
also have ... ⊆ *Sup-upto-llist* (*lmap Red-I-*G *Ns*) (*enat i*)
using *invar* **unfolding** *gc-invar-def lim-i ni* **by** *auto*
also have ... ⊆ *Sup-upto-llist* (*lmap Red-I-*G *Ns*) (*enat* (*Suc i*))
by *simp*
finally show *?thesis*
unfolding *gc-invar-def lim-Si nSi* **.**
next
case (*infer N C L M*)
note *ni* = *this*(*1*) **and** *nSi* = *this*(*2*) **and** *l-pas* = *this*(*3*) **and** *m-pas* = *this*(*4*) **and**
inff-red = *this*(*5*)

{
fix ι
assume *ι-inff*: ι ∈ *Inf-from* (*active-subset* (*N* ∪ {(*C*, *active*)} ∪ *M*))

have *ι-inf*: ι ∈ *Inf-FL*
using *ι-inff* **unfolding** *Inf-from-def* **by** *auto*
then have *Fι-inf*: *to-F* ι ∈ *Inf-F*
using *in-Inf-FL-imp-to-F-in-Inf-F* **by** *blast*

have ι ∈ *Inf-from* (*active-subset N* ∪ {(*C*, *active*)})
using *ι-inff m-pas* **by** *simp*
then have *Fι-inff*:
to-F ι ∈ *no-labels.Inf-from* (*fst* ‘ (*active-subset N* ∪ {(*C*, *active*)}))
using *Fι-inf* **unfolding** *to-F-def Inf-from-def no-labels.Inf-from-def* **by** *auto*

have ι ∈ *Sup-upto-llist* (*lmap Red-I-*G *Ns*) (*enat* (*Suc i*))
proof (*cases to-F* ι ∈ *no-labels.Inf-between* (*fst* ‘ *active-subset N*) {*C*})
case *True*
then have *to-F* ι ∈ *no-labels.Red-I-*G (*fst* ‘ (*N* ∪ {(*C*, *active*)} ∪ *M*))
using *inff-red* **by** *auto*

```
      then have ι ∈ Red-I-G (N ∪ {(C, active)} ∪ M)
        by (subst labeled-red-inf-eq-red-inf[OF ι-inf])
      then show ?thesis
        using Si-lt using nSi by auto
    next
      case False
      then have to-F ι ∈ no-labels.Inf-from (fst ‘ active-subset N)
        using Fι-inff unfolding no-labels.Inf-from-def no-labels.Inf-between-def by auto
      then have ι ∈ Inf-from (active-subset N)
        using ι-inf l-pas unfolding to-F-def Inf-from-def no-labels.Inf-from-def
        by clarsimp (smt ι-inff[unfolded Inf-from-def] active-subset-def imageE image-subset-iff
            in-mono mem-Collect-eq prod.collapse)
      then show ?thesis
        using invar l-pas unfolding gc-invar-def lim-i ni by auto
    qed
  }
  then show ?thesis
    unfolding gc-invar-def lim-Si nSi by blast
 }
qed

lemma gc-invar-gc:
 assumes
  gc: chain (⇝GC) Ns and
  init: active-subset (lhd Ns) = {} and
  i-lt: i < llength Ns
 shows gc-invar Ns i
 using i-lt
proof (induct i)
 case (enat i)
 then show ?case
 proof (induct i)
  case 0
  then show ?case
   using gc-invar-gc-init[OF chain-not-lnull[OF gc] init] by (simp add: enat-0)
 next
  case (Suc i)
  note ih = this(1) and Si-lt = this(2)
  have i-lt: enat i < llength Ns
   using Si-lt Suc-ile-eq less-le by blast
  show ?case
   by (rule gc-invar-gc-step[OF Si-lt ih[OF i-lt] chain-lnth-rel[OF gc Si-lt]])
 qed
qed simp

lemma gc-fair-new-proof:
 assumes
  gc: chain (⇝GC) Ns and
  init: active-subset (lhd Ns) = {} and
  lim: passive-subset (Liminf-llist Ns) = {}
 shows fair Ns
 unfolding fair-def
proof −
 have Inf-from (Liminf-llist Ns) ⊆ Inf-from (active-subset (Liminf-llist Ns)) (is ?lhs ⊆ -)
  using lim unfolding active-subset-def passive-subset-def
```

by (*metis* (*no-types*, *lifting*) *Inf-from-mono empty-Collect-eq mem-Collect-eq subsetI*)
also have ... ⊆ *Sup-llist* (*lmap Red-I-G Ns*) (**is** - ⊆ *?rhs*)
using *gc-invar-infinity*[*OF chain-not-lnull*[*OF gc*]] *gc-invar-gc*[*OF gc init*]
unfolding *gc-invar-def* **by** *fastforce*
finally show *?lhs* ⊆ *?rhs*
.
qed

end

5.2 Lazy Given Clause

context *lazy-given-clause*
begin

definition *from-F* :: *'f inference* ⇒ (*'f* × *'l*) *inference set* **where**
from-F ι = {ι′ ∈ *Inf-FL*. *to-F* ι′ = ι}

definition *lgc-invar* :: (*'f inference set* × (*'f* × *'l*) *set*) *llist* ⇒ *enat* ⇒ *bool* **where**
lgc-invar TNs i ⟷
Inf-from (*active-subset* (*Liminf-upto-llist* (*lmap snd TNs*) *i*))
⊆ ⋃ (*from-F* ‘ *Liminf-upto-llist* (*lmap fst TNs*) *i*) ∪ *Sup-upto-llist* (*lmap* (*Red-I-G* ∘ *snd*) *TNs*) *i*

lemma *lgc-invar-infinity*:
assumes
nnil: ¬ *lnull TNs* **and**
invar: ∀ *i*. *enat i* < *llength TNs* ⟶ *lgc-invar TNs* (*enat i*)
shows *lgc-invar TNs* ∞
unfolding *lgc-invar-def*
proof (*intro subsetI*, *unfold Liminf-upto-llist-infinity Sup-upto-llist-infinity*)
fix ι
assume ι-*inff*: ι ∈ *Inf-from* (*active-subset* (*Liminf-llist* (*lmap snd TNs*)))

define *As* **where**
As = *lmap* (*active-subset* ∘ *snd*) *TNs*

have *act-ns*: *active-subset* (*Liminf-llist* (*lmap snd TNs*)) = *Liminf-llist As*
unfolding *As-def active-subset-def Liminf-set-filter-commute*[*symmetric*] *llist.map-comp* **..**

note ι-*inf* = *Inf-if-Inf-from*[*OF* ι-*inff*]
note ι-*inff′* = ι-*inff*[*unfolded act-ns*]

show ι ∈ ⋃ (*from-F* ‘ *Liminf-llist* (*lmap fst TNs*)) ∪ *Sup-llist* (*lmap* (*Red-I-G* ∘ *snd*) *TNs*)
proof −
{
assume ι-*ni-lim*: ι ∉ ⋃ (*from-F* ‘ *Liminf-llist* (*lmap fst TNs*))

have ¬ *lnull As*
unfolding *As-def* **using** *nnil* **by** *auto*
moreover have *set* (*prems-of* ι) ⊆ *Liminf-llist As*
using ι-*inff′*[*unfolded Inf-from-def*] **by** *simp*
ultimately obtain *i* **where**
i-lt-as: *enat i* < *llength As* **and**
prems-sub-ge-i: *set* (*prems-of* ι) ⊆ (⋂ *j* ∈ {*j*. *i* ≤ *j* ∧ *enat j* < *llength As*}. *lnth As j*)
using *finite-subset-Liminf-llist-imp-exists-index* **by** *blast*

```
    have ts-nnil: ¬ lnull (lmap fst TNs)
      using As-def nnil by simp

    have Fι-ni-lim: to-F ι ∉ Liminf-llist (lmap fst TNs)
      using ι-inf ι-ni-lim unfolding from-F-def by auto
    obtain i' where
      i-le-i': i ≤ i' and
      i'-lt-as: enat i' < llength As and
      Fι-ni-i': to-F ι ∉ lnth (lmap fst TNs) i'
      using i-lt-as not-Liminf-llist-imp-exists-index[OF ts-nnil Fι-ni-lim, of i] unfolding As-def
      by auto

    have ι-ni-i': ι ∉ ⋃ (from-F ‘ fst (lnth TNs i'))
      using Fι-ni-i' i'-lt-as[unfolded As-def] unfolding from-F-def by auto

    have set (prems-of ι) ⊆ (⋂j ∈ {j. i' ≤ j ∧ enat j < llength As}. lnth As j)
      using prems-sub-ge-i i-le-i' by auto
    then have set (prems-of ι) ⊆ lnth As i'
      using i'-lt-as by auto
    then have ι ∈ Inf-from (active-subset (snd (lnth TNs i')))
      using i'-lt-as ι-inf unfolding Inf-from-def As-def by auto
    then have ι-in-i': ι ∈ Sup-upto-llist (lmap (Red-I-𝒢 ∘ snd) TNs) (enat i')
      using ι-ni-i' i'-lt-as[unfolded As-def] invar[unfolded lgc-invar-def] by auto
    then have ι ∈ Sup-llist (lmap (Red-I-𝒢 ∘ snd) TNs)
      using Sup-upto-llist-subset-Sup-llist by fastforce
  }
  then show ?thesis
    by blast
 qed
qed

lemma lgc-invar-lgc-init:
 assumes
  nnil: ¬ lnull TNs and
  n-init: active-subset (snd (lhd TNs)) = {} and
  t-init: ∀ι ∈ Inf-F. prems-of ι = [] ⟶ ι ∈ fst (lhd TNs)
 shows lgc-invar TNs 0
 unfolding lgc-invar-def
proof −
 have Inf-from (active-subset (Liminf-upto-llist (lmap snd TNs) 0)) =
  Inf-from {} (is ?lhs = -)
  using nnil n-init by auto
 also have ... ⊆ ⋃ (from-F ‘ fst (lhd TNs))
  using t-init Inf-FL-to-Inf-F unfolding Inf-from-def from-F-def to-F-def by force
 also have ... ⊆ ⋃ (from-F ‘ fst (lhd TNs)) ∪ Red-I-𝒢 (snd (lhd TNs))
  by fast
 also have ... = ⋃ (from-F ‘ Liminf-upto-llist (lmap fst TNs) 0)
  ∪ Sup-upto-llist (lmap (Red-I-𝒢 ∘ snd) TNs) 0 (is - = ?rhs)
  using nnil by auto
 finally show ?lhs ⊆ ?rhs
  .
qed

lemma lgc-invar-lgc-step:
 assumes
```

```
    Si-lt: enat (Suc i) < llength TNs and
    invar: lgc-invar TNs i and
    step: lnth TNs i ⇝LGC lnth TNs (Suc i)
  shows lgc-invar TNs (Suc i)
  using step Si-lt invar
proof cases
  let ?Sup-Red-i = Sup-upto-llist (lmap (Red-I-G ∘ snd) TNs) (enat i)
  let ?Sup-Red-Si = Sup-upto-llist (lmap (Red-I-G ∘ snd) TNs) (enat (Suc i))

  have i-lt: enat i < llength TNs
    using Si-lt Suc-ile-eq order.strict-implies-order by blast

  have lim-i:
    Liminf-upto-llist (lmap fst TNs) (enat i) = lnth (lmap fst TNs) i
    Liminf-upto-llist (lmap snd TNs) (enat i) = lnth (lmap snd TNs) i
    using i-lt by auto
  have lim-Si:
    Liminf-upto-llist (lmap fst TNs) (enat (Suc i)) = lnth (lmap fst TNs) (Suc i)
    Liminf-upto-llist (lmap snd TNs) (enat (Suc i)) = lnth (lmap snd TNs) (Suc i)
    using Si-lt by auto

  {
    case (process N1 N M N2 M' T)
    note tni = this(1) and tnSi = this(2) and n1 = this(3) and n2 = this(4) and m-red = this(5)
and
      m'-pas = this(6)

    have ni: lnth (lmap snd TNs) i = N ∪ M
      by (simp add: i-lt n1 tni)
    have nSi: lnth (lmap snd TNs) (Suc i) = N ∪ M'
      by (simp add: Si-lt n2 tnSi)
    have ti: lnth (lmap fst TNs) i = T
      by (simp add: i-lt tni)
    have tSi: lnth (lmap fst TNs) (Suc i) = T
      by (simp add: Si-lt tnSi)

    have Inf-from (active-subset (N ∪ M')) ⊆ Inf-from (active-subset (N ∪ M))
      using m'-pas by (simp add: Inf-from-mono)
    also have ... ⊆ ⋃ (from-F ‘ T) ∪ ?Sup-Red-i
      using invar unfolding lgc-invar-def lim-i ni ti .
    also have ... ⊆ ⋃ (from-F ‘ T) ∪ ?Sup-Red-Si
      using Sup-upto-llist-mono by auto
    finally show ?thesis
      unfolding lgc-invar-def lim-Si nSi tSi .
  next
    case (schedule-infer T2 T1 T' N1 N C L N2)
    note tni = this(1) and tnSi = this(2) and t2 = this(3) and n1 = this(4) and n2 = this(5) and
      l-pas = this(6) and t' = this(7)

    have ni: lnth (lmap snd TNs) i = N ∪ {(C, L)}
      by (simp add: i-lt n1 tni)
    have nSi: lnth (lmap snd TNs) (Suc i) = N ∪ {(C, active)}
      by (simp add: Si-lt n2 tnSi)
    have ti: lnth (lmap fst TNs) i = T1
      by (simp add: i-lt tni)
```

```
    have tSi: lnth (lmap fst TNs) (Suc i) = T1 ∪ T'
      by (simp add: Si-lt t2 tnSi)

    {
      fix ι
      assume ι-inff: ι ∈ Inf-from (active-subset (N ∪ {(C, active)}))

      have ι-inf: ι ∈ Inf-FL
        using ι-inff unfolding Inf-from-def by auto
      then have Fι-inf: to-F ι ∈ Inf-F
        using in-Inf-FL-imp-to-F-in-Inf-F by blast

      have ι ∈ ⋃ (from-F ‘ (T1 ∪ T')) ∪ ?Sup-Red-Si
      proof (cases to-F ι ∈ no-labels.Inf-between (fst ‘ active-subset N) {C})
        case True
        then have ι ∈ ⋃ (from-F ‘ (T1 ∪ T'))
          unfolding t' from-F-def using ι-inf by auto
        then show ?thesis
          by blast
      next
        case False
        moreover have to-F ι ∈ no-labels.Inf-from (fst ‘ (active-subset N ∪ {(C, active)}))
          using ι-inff Fι-inf unfolding to-F-def Inf-from-def no-labels.Inf-from-def by auto
        ultimately have to-F ι ∈ no-labels.Inf-from (fst ‘ active-subset N)
          unfolding no-labels.Inf-from-def no-labels.Inf-between-def by auto
        then have ι ∈ Inf-from (active-subset N)
          using ι-inf unfolding to-F-def no-labels.Inf-from-def
          by clarsimp (smt Inf-from-def Un-insert-right ι-inff active-subset-def
            boolean-algebra-cancel.sup0 imageE image-subset-iff insert-iff mem-Collect-eq
            prod.collapse snd-conv subset-iff)
        then have ι ∈ ⋃ (from-F ‘ (T1 ∪ T')) ∪ ?Sup-Red-i
          using invar[unfolded lgc-invar-def] l-pas unfolding lim-i ni ti by auto
        then show ?thesis
          using Sup-upto-llist-mono by force
      qed
    }
    then show ?thesis
      unfolding lgc-invar-def lim-i lim-Si nSi tSi by fast
  next
    case (compute-infer T1 T2 ι' N2 N1 M)
    note tni = this(1) and tnSi = this(2) and t1 = this(3) and n2 = this(4) and m-pas = this(5)
and
      ι'-red = this(6)

    have ni: lnth (lmap snd TNs) i = N1
      by (simp add: i-lt tni)
    have nSi: lnth (lmap snd TNs) (Suc i) = N1 ∪ M
      by (simp add: Si-lt n2 tnSi)
    have ti: lnth (lmap fst TNs) i = T2 ∪ {ι'}
      by (simp add: i-lt t1 tni)
    have tSi: lnth (lmap fst TNs) (Suc i) = T2
      by (simp add: Si-lt tnSi)

    {
      fix ι
```

```
    assume ι-inff: ι ∈ Inf-from (active-subset (N1 ∪ M))

    have ι-bef: ι ∈ ⋃ (from-F ‘ (T2 ∪ {ι'})) ∪ ?Sup-Red-i
      using ι-inff invar[unfolded lgc-invar-def lim-i ti ni] m-pas by auto
    have ι ∈ ⋃ (from-F ‘ T2) ∪ ?Sup-Red-Si
    proof (cases ι ∈ from-F ι')
      case ι-in-ι': True
      then have ι ∈ Red-I-G (N1 ∪ M)
        using ι'-red from-F-def labeled-red-inf-eq-red-inf by auto
      then have ι ∈ ?Sup-Red-Si
        using Si-lt by (simp add: n2 tnSi)
      then show ?thesis
        by auto
    next
      case False
      then show ?thesis
        using ι-bef by auto
    qed
  }
  then show ?thesis
    unfolding lgc-invar-def lim-Si tSi nSi by blast
next
  case (delete-orphan-infers T1 T2 T' N)
  note tni = this(1) and tnSi = this(2) and t1 = this(3) and t'-orph = this(4)

  have ni: lnth (lmap snd TNs) i = N
    by (simp add: i-lt tni)
  have nSi: lnth (lmap snd TNs) (Suc i) = N
    by (simp add: Si-lt tnSi)
  have ti: lnth (lmap fst TNs) i = T2 ∪ T'
    by (simp add: i-lt t1 tni)
  have tSi: lnth (lmap fst TNs) (Suc i) = T2
    by (simp add: Si-lt tnSi)

  {
    fix ι
    assume ι-inff: ι ∈ Inf-from (active-subset N)

    have to-F ι ∉ T'
      using t'-orph ι-inff in-Inf-from-imp-to-F-in-Inf-from by blast
    hence ι ∉ ⋃ (from-F ‘ T')
      unfolding from-F-def by auto
    then have ι ∈ ⋃ (from-F ‘ T2) ∪ ?Sup-Red-Si
      using ι-inff invar unfolding lgc-invar-def lim-i ni ti by auto
  }
  then show ?thesis
    unfolding lgc-invar-def lim-Si tSi nSi by blast
}
qed

lemma lgc-invar-lgc:
  assumes
    lgc: chain (⇝LGC) TNs and
    n-init: active-subset (snd (lhd TNs)) = {} and
    t-init: ∀ ι ∈ Inf-F. prems-of ι = [] ⟶ ι ∈ fst (lhd TNs) and
```

```
    i-lt: i < llength TNs
  shows lgc-invar TNs i
  using i-lt
proof (induct i)
  case (enat i)
  then show ?case
  proof (induct i)
    case 0
    then show ?case
      using lgc-invar-lgc-init[OF chain-not-lnull[OF lgc] n-init t-init] by (simp add: enat-0)
  next
    case (Suc i)
    note ih = this(1) and Si-lt = this(2)
    have i-lt: enat i < llength TNs
      using Si-lt Suc-ile-eq less-le by blast
    show ?case
      by (rule lgc-invar-lgc-step[OF Si-lt ih[OF i-lt] chain-lnth-rel[OF lgc Si-lt]])
  qed
qed simp

lemma lgc-fair-new-proof:
  assumes
    lgc: chain (⇝LGC) TNs and
    n-init: active-subset (snd (lhd TNs)) = {} and
    n-lim: passive-subset (Liminf-llist (lmap snd TNs)) = {} and
    t-init: ∀ ι ∈ Inf-F. prems-of ι = [] ⟶ ι ∈ fst (lhd TNs) and
    t-lim: Liminf-llist (lmap fst TNs) = {}
  shows fair (lmap snd TNs)
  unfolding fair-def llist.map-comp
proof −
  have Inf-from (Liminf-llist (lmap snd TNs))
    ⊆ Inf-from (active-subset (Liminf-llist (lmap snd TNs))) (is ?lhs ⊆ -)
    by (rule Inf-from-mono) (use n-lim passive-subset-def active-subset-def in blast)
  also have ... ⊆ ⋃ (from-F ‘ Liminf-upto-llist (lmap fst TNs) ∞)
    ∪ Sup-llist (lmap (Red-I-G ∘ snd) TNs)
    using lgc-invar-infinity[OF chain-not-lnull[OF lgc]] lgc-invar-lgc[OF lgc n-init t-init]
    unfolding lgc-invar-def by fastforce
  also have ... ⊆ Sup-llist (lmap (Red-I-G ∘ snd) TNs) (is - ⊆ ?rhs)
    using t-lim by auto
  finally show ?lhs ⊆ ?rhs
    .
qed

end

end
```