The Linear-Time–Branching-Time Spectrum of Modal Expressiveness

Benjamin Bisping

November 8, 2024

Abstract

...

Contents

1 Labeled Transition Systems **1**

1.1 Labeled Transition Systems 2

1.2 Paths and Traces 2

1.3 Transition Systems with Internal Behavior 3

2 Strong Equivalences **5**

2.1 Trivial notions of equality 5

2.1.1 Identity 5

2.1.2 Universal equality 5

2.2 Trace Equality 6

2.3 Isomorphism 6

2.4 Simulation preorder and equivalence 7

3 Hennessy–Milner Logic **10**

4 Reachability Games **11**

5 The Bisimulation Game **13**

6 Priced Spectrum **21**

6.1 Enabledness 21

6.2 Traces 22

6.3 Simulation 22

[1]....

1 Labeled Transition Systems

```
theory Labeled_Transition_Systems
  imports
    Main
begin
```

1.1 Labeled Transition Systems

A locale for Labeled Transition Systems, parameterized over action type 'a and state type 's.

```
locale lts =
  fixes step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool›
    (‹_ ⟼_ _› [70, 70, 70] 80)
begin
```

Example definitions for derivatives and deadlock.

```
abbreviation derivatives :: ‹'s ⇒ 'a ⇒ 's set›
  where ‹derivatives p α ≡ {p'. p ⟼α p'}›

abbreviation deadlocked :: ‹'s ⇒ bool›
  where ‹deadlocked s ≡ ∀α. derivatives s α = {}›

definition image_finite :: ‹bool›
  where ‹image_finite ≡ (∀ p α. finite (derivatives p α))›
```

1.2 Paths and Traces

Step sequences as inductive definition

Teaching Hint: Inductive definitions!

```
inductive step_sequence :: ‹'s ⇒ 'a list ⇒ 's ⇒ bool›
    ("_ ⟼$ _ _" [70, 70, 70] 80)
  where
  srefl: ‹p ⟼$ [] p› |
  sstep: ‹p ⟼$ (a#tr) p''› if ‹∃p'. p ⟼ a p' ∧ p' ⟼$ tr p''›
```

Traces as enabled step sequences

```
abbreviation traces :: ‹'s ⇒ 'a list set›
  where ‹traces p ≡ {tr. ∃p'. p ⟼$ tr p'}›

lemma empty_trace_trivial:
  fixes p
  shows ‹[] ∈ traces p›
  using step_sequence.srefl by blast

inductive path :: ‹'s list ⇒ bool›
  where
  init: ‹path [p]› |
  step: ‹path (p # (p' # pp))› if ‹∃α. p ⟼ α p' ∧ path (p' # pp)›

lemma no_empty_paths:
  assumes ‹path []›
  shows ‹False›
  using assms path.cases by blast

lemma path_implies_trace:
  assumes ‹path pp›
  shows ‹∃tr. (hd pp) ⟼$ tr (last pp)›
  using assms
proof induct
  case (init p)
  then show ?case using step_sequence.srefl by force
```

```
next
  case (step p p' pp)
  then show ?case by (metis last_ConsR list.distinct(1) list.sel(1) step_sequence.simps)

qed

lemma trace_implies_path:
  assumes ‹p ⟼$ tr p'›
  shows ‹∃pp. path pp ∧ hd pp = p ∧ last pp = p'›
  using assms
proof induct
  case (srefl p)
  then show ?case using path.init by fastforce
next
  case (sstep p a tr p'')
  then show ?case by (metis last.simps list.collapse list.sel(1) no_empty_paths
path.step)
qed

end — of locale lts
```

1.3 Transition Systems with Internal Behavior

```
locale lts_tau =
  lts step
  for
    step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ⟼_ _› [70, 70, 70] 80) +
  fixes
    τ :: 'a
begin
```

Define silent-reachability ↠ and prove its transitivity

```
inductive silent_reachable :: ‹'s ⇒ 's ⇒ bool› (infix ‹↠› 80) where
  refl: ‹p ↠ p› |
  step: ‹p ↠ p''› if ‹p ⟼ τ p'› and ‹p' ↠ p''›

thm silent_reachable.induct

lemma silent_reachable_compose:
  fixes
    p p' p''
  assumes
    ‹p ↠ p'›
    ‹p' ↠ p''›
  shows
    ‹p ↠ p''›
  using assms
proof induct
  fix p
  assume ‹p ↠ p''›
  thus ‹p ↠ p''› .
next
  fix p p' p'''
  assume ‹p ⟼τ p'› ‹p''' ↠ p'' ⟹ p' ↠ p''› ‹p''' ↠ p''›
  from this(2,3) have ‹p' ↠ p''› .
  with silent_reachable.step ‹p ⟼τ p'› show ‹p ↠ p''› .
```

```
qed

lemma silent_reachable_preorder:
   ‹reflp (↠)›
   ‹transp (↠)›
  using silent_reachable.refl silent_reachable_compose
  unfolding reflp_def transp_def by blast+
```

A weak step ↠⟼ is ⟼ wrapped in ↠ (or just ↠ for τ)

```
definition weak_step (‹_ ↠⟼ _ _› [70,70,70] 80) where
  ‹p ↠⟼ α p''' ≡
    if α = τ then p ↠ p''' else
    (∃p' p''. p ↠ p' ∧ p' ⟼ α p'' ∧ p'' ↠ p''')›
```

weak step sequence ↠⟼$ and traces analogous to strong steps.

```
inductive weak_step_sequence :: ‹'s ⇒ 'a list ⇒ 's ⇒ bool›
    ("_ ↠⟼$ _ _" [70, 70, 70] 80)
  where
  internal: ‹p ↠⟼$ [] p'› if ‹p ↠ p'› |
  step: ‹p ↠⟼$ (α#tr) p''› if ‹∃p'. p ↠⟼ α p' ∧ p' ↠⟼$ tr p''›

abbreviation weak_traces :: ‹'s ⇒ 'a list set›
  where ‹weak_traces p ≡ {tr. ∃p'. p ↠⟼$ tr p'}›

lemma empty_weak_trace_trivial:
  fixes p
  shows ‹[] ∈ weak_traces p›
  using weak_step_sequence.internal silent_reachable.refl by blast

lemma weak_seq_tau_transparent:
  assumes ‹p ↠⟼$ tr p'›
  shows ‹p ↠⟼$ (filter (λα. α ≠ τ) tr) p'›
  using assms
proof (induct tr arbitrary: p p')
  case Nil
  then show ?case by simp
next
  case (Cons α tr p p'')
  from this(2) obtain p' where step1: ‹p ↠⟼ α p'› ‹p' ↠⟼$ tr p''›
    using weak_step_sequence.cases by force
  then have step2: ‹p' ↠⟼$ (filter (λα. α ≠ τ) tr) p''› using Cons(1) by blast
  show ?case
  proof (cases ‹α ≠ τ›)
    case True
    then show ?thesis using step1(1) step2
      using weak_step_sequence.step by fastforce
  next
    case False
    hence ‹α = τ› by blast
    hence ‹p ↠ p'› using step1(1) unfolding weak_step_def by auto
    hence ‹p ↠⟼$ tr p''› using step1(2) weak_step_def silent_reachable_compose
      by (smt (verit, best)
        weak_step_sequence.internal weak_step_sequence.cases weak_step_sequence.step)
    hence ‹p ↠⟼$ (filter (λα. α ≠ τ) tr) p''› using Cons.hyps by blast
    then show ?thesis using ‹α = τ› by auto
  qed
```

```
qed

end

end
```

2 Strong Equivalences

```
theory Strong_Equivalences
  imports Labeled_Transition_Systems
begin

context lts begin
```

2.1 Trivial notions of equality

2.1.1 Identity

```
definition identical :: ‹'s ⇒ 's ⇒ bool›
  where ‹identical p q ≡ p = q›
```

It's reflexive

```
lemma identical_reflexive:
  shows ‹identical p p›
  unfolding identical_def using refl .

lemma non_identity:
  assumes ‹p ≠ q›
  shows ‹¬ identical p q›
  using assms unfolding identical_def .
```

It's an equivalence.

```
lemma identity_equivalence:
  shows ‹equivp identical›
proof (rule equivpI)
  show ‹reflp identical› unfolding reflp_def using allI identical_reflexive .
next
  show ‹symp identical› unfolding symp_def
  proof (rule+)
    fix x y
    assume ‹identical x y›
    then have ‹x = y› unfolding identical_def .
    with sym have ‹y = x› .
    then show ‹identical y x› unfolding identical_def .
  qed
next
  show ‹transp identical›
    unfolding transp_def identical_def by blast
qed
```

2.1.2 Universal equality

```
definition universal_equal :: ‹'s ⇒ 's ⇒ bool›
  where ‹universal_equal p q ≡ True›

lemma universal_equal_equivalence:
  shows ‹equivp universal_equal›
  unfolding equivp_def universal_equal_def by simp
```

2.2 Trace Equality

Trace preorder as inclusion of trace sets

```
definition trace_preordered :: ‹'s ⇒ 's ⇒ bool› (infix ‹≲T› 80) where
    ‹p ≲T q ≡ traces p ⊆ traces q›
```

Trace equivalence as mutual preorder

```
abbreviation trace_equivalent (infix ‹≃T› 80) where
  ‹p ≃T q ≡ p ≲T q ∧ q ≲T p›
```

Trace preorder is transitive

```
lemma trace_preorder_transitive:
  shows ‹transp (≲T)›
  unfolding trace_preordered_def
by (standard, blast)

lemma trace_equivalence_equiv:
  shows ‹equivp trace_equivalent›
proof (rule equivpI)
  show ‹transp trace_equivalent›
    using trace_preorder_transitive
    unfolding transp_def
    by blast
next
  show ‹reflp trace_equivalent›
    unfolding reflp_def trace_preordered_def by blast
next
  show ‹symp trace_equivalent›
    unfolding symp_def trace_preordered_def by blast
qed
```

2.3 Isomorphism

```
definition isomorphism :: ‹('s ⇒ 's) ⇒ bool›
  where ‹isomorphism f ≡ bij f ∧ (∀p p' a. p ⟼ a p' ⟷ (f p) ⟼ a (f p'))›

definition is_isomorphic_to (infix ‹≃ISO› 80)
  where ‹p ≃ISO q ≡ ∃f. f p = q ∧ isomorphism f›
```

Isomorphism yields an equivalence

```
lemma iso_equivalence_equiv:
  shows ‹equivp is_isomorphic_to›
proof (rule equivpI)
  show ‹reflp is_isomorphic_to›
    unfolding reflp_def is_isomorphic_to_def isomorphism_def by (metis bij_id id_apply)
next
  show ‹symp is_isomorphic_to›
    unfolding symp_def is_isomorphic_to_def isomorphism_def by (metis (no_types,
opaque_lifting) bij_iff)
next
  show ‹transp is_isomorphic_to›
    unfolding transp_def is_isomorphic_to_def
  proof safe
    fix p f1 f2
    assume ‹isomorphism f1› ‹isomorphism f2›
    then have ‹isomorphism (f2 ∘ f1)› unfolding isomorphism_def using bij_comp
by auto
    then show ‹∃fb. fb p = f2 (f1 p) ∧ isomorphism fb› unfolding comp_def by auto
  qed
qed
```

Isomorphism equivalence is closed under steps (i.e. isomorphism equivalence is a simulation, but we have not yet defined this notion.)

```
lemma iso_sim:
  assumes
    ‹is_isomorphic_to p q›
    ‹p ⟼ a p'›
  shows ‹∃q'. q ⟼ a q' ∧ is_isomorphic_to p' q'›
using assms unfolding is_isomorphic_to_def isomorphism_def by blast
```

Isomorphic states have the same traces.

Teaching hint: Inductive proofs

```
lemma iso_implies_trace_preord:
  assumes ‹is_isomorphic_to p q›
  shows ‹trace_preordered p q›
  unfolding trace_preordered_def
proof safe
  fix tr p'
  assume ‹p ⟼$tr p'›
  thus ‹∃p'. q ⟼$tr p'› using assms
  proof (induct tr arbitrary: p q p')
    case Nil
    then show ?case using empty_trace_trivial by blast
  next
    case (Cons a tr p q p')
    from ‹p ⟼$(a # tr) p'›
      obtain p'' where ‹p ⟼a p''› ‹p'' ⟼$ tr p'›
      by (cases, auto)
    then show ?case
      using iso_sim[OF ‹is_isomorphic_to p q› ‹p ⟼a p''›]
        Cons(1) sstep by blast
  qed
qed
```

— Actually, this is more like a corollary of later lemmas.

```
corollary iso_implies_trace_eq:
  assumes ‹is_isomorphic_to p q›
  shows ‹trace_equivalent p q›
  using assms iso_implies_trace_preord iso_equivalence_equiv
  unfolding equivp_def by simp
```

2.4 Simulation preorder and equivalence

Two states are simulation preordered if they can be related by a simulation relation.

```
definition simulation
  where ‹simulation R ≡
    ∀p q a p'. p ⟼ a p' ∧ R p q ⟶ (∃q'. q ⟼ a q' ∧ R p' q')›
```

```
definition simulated_by (infix ‹≲S› 80)
  where ‹p ≲S q ≡ ∃R. R p q ∧ simulation R›

abbreviation similar (infix ‹≃S› 80)
  where ‹p ≃S q ≡ p ≲S q ∧ q ≲S p›

lemma id_sim:
  shows ‹simulation identical›
  unfolding simulation_def identical_def by blast

lemma simulation_composition:
  assumes
    ‹simulation R1›
    ‹simulation R2›
  shows
    ‹simulation (λp q. ∃p’. R1 p p’ ∧ R2 p’ q)›
  using assms unfolding simulation_def by blast

lemma simulation_union:
  assumes
    ‹simulation R1›
    ‹simulation R2›
  shows
    ‹simulation (λp q. R1 p q ∨ R2 p q)›
  using assms unfolding simulation_def by blast

lemma simulation_preorder_transitive:
  shows ‹transp (≲S)›
  unfolding transp_def simulated_by_def
  using simulation_composition
  by (metis (mono_tags, lifting))

lemma iso_is_sim:
  shows ‹simulation is_isomorphic_to›
  using iso_sim unfolding simulated_by_def simulation_def by blast

corollary iso_implies_sim:
  assumes ‹is_isomorphic_to p q›
  shows ‹simulated_by p q›
  using assms iso_is_sim unfolding simulated_by_def by blast

lemma sim_implies_trace_preord:
  assumes ‹p ≲S q›
  shows ‹p ≲T q›
  unfolding trace_preordered_def
proof safe
  fix tr p’
  assume ‹p ⟼$tr p’›
  thus ‹∃p’. q ⟼$tr p’› using assms
  proof (induct tr arbitrary: p q p’)
    case Nil
    then show ?case using empty_trace_trivial by blast
  next
    case (Cons a tr p q p’)
    from ‹p ⟼$(a # tr) p’›
```

```
      obtain p'' where p''_spec: ‹p ⟼a p''› ‹p'' ⟼$ tr p'›
      by (cases, auto)
    then obtain R q' where ‹simulation R› ‹R p q› ‹R p'' q'›
      using Cons unfolding simulated_by_def simulation_def by blast
    then show ?case
      using Cons step_sequence.sstep p''_spec
        simulated_by_def simulation_def by metis
  qed
qed


lemma sim_eq_implies_trace_eq:
  assumes ‹p ≃S q›
  shows ‹p ≃T q›
  using assms sim_implies_trace_preord by blast
```

Two states are bisimilar if they can be related by a symmetric simulation.

```
definition bisimilar (infix ‹≃B› 80) where
  ‹bisimilar p q ≡ ∃R. simulation R ∧ symp R ∧ R p q›
```

Bisimilarity is a simulation.

```
lemma bisim_sim:
  shows ‹simulation bisimilar›
  unfolding bisimilar_def simulation_def by blast

lemma bisimilarity_equiv:
  shows ‹equivp (≃B)›
proof (rule equivpI)
  show ‹reflp (≃B)›
    using id_sim DEADID.rel_symp
    unfolding bisimilar_def identical_def
    by (metis (mono_tags, lifting) reflpI)
next
  show ‹symp (≃B)›
    unfolding bisimilar_def
    by (smt (verit, best) sympE sympI)
next
  show ‹transp (≃B)›
    unfolding transp_def bisimilar_def
  proof safe
    fix p p' q R1 R2
    assume case_assms:
      ‹simulation R1› ‹symp R1› ‹R1 p p'›
      ‹simulation R2› ‹symp R2› ‹R2 p' q›
    hence ‹simulation (λp q. (∃p'. R1 p p' ∧ R2 p' q) ∨ (∃p'. R2 p p' ∧ R1 p'
q))›
      using simulation_composition simulation_union by blast
    moreover have
      ‹symp (λp q. (∃p'. R1 p p' ∧ R2 p' q) ∨ (∃p'. R2 p p' ∧ R1 p' q))›
      using case_assms(2,5) unfolding symp_def by blast
    ultimately show ‹∃R. simulation R ∧ symp R ∧ R p q›
      using case_assms(3,6) by blast
  qed
qed
```

Bisimilarity is a bisimulation.

```
lemma bisim_bisim:
  shows ‹simulation bisimilar ∧ symp bisimilar›
  using bisim_sim bisimilarity_equiv equivpE by blast

lemma bisim_implies_sim:
  assumes ‹p ≃B q›
  shows ‹p ≃S q›
  using assms bisim_bisim
  unfolding simulated_by_def
  by (metis sympE)

lemma iso_implies_bisim:
  assumes ‹p ≃ISO q›
  shows ‹p ≃B q›
using assms iso_is_sim equivpE[OF iso_equivalence_equiv] bisimilar_def by blast

end

end
```

3 Hennessy–Milner Logic

```
theory Hennessy_Milner_Logic
imports
  LTS_Semantics
begin
```

HML formulas can be the trivial formula, conjunctions, negations and observations of possible transitions.

```
datatype ('a,'i) hml_formula =
  HML_true
| HML_conj ‹'i set› ‹'i ⇒ ('a,'i) hml_conjunct›  (‹AND _ _›)
| HML_obs ‹'a› ‹('a,'i) hml_formula›             (‹⟨_⟩_› [60] 60)
and ('a,'i) hml_conjunct =
  HML_pos ‹('a,'i) hml_formula›                 (‹+_› [20] 60)
| HML_neg ‹('a,'i) hml_formula›                 (‹-_› [20] 60)

context lts
begin
```

The model relation

```
primrec satisfies :: ‹'s ⇒ ('a, 's) hml_formula ⇒ bool›    (‹_ ⊨ _› [50, 50]
40)
  and satisfies_conj :: ‹'s ⇒ ('a, 's) hml_conjunct ⇒ bool›
  where
    ‹(p ⊨ HML_true) = True› |
    ‹(p ⊨ HML_conj I F) = (∀ i ∈ I. satisfies_conj p (F i))› |
    ‹(p ⊨ HML_obs α φ) = (∃ p'. p ⟼α p' ∧ p' ⊨ φ)› |
    ‹satisfies_conj p (HML_pos φ) = (p ⊨ φ)› |
    ‹satisfies_conj p (HML_neg φ) = (¬p ⊨ φ)›

interpretation hml: lts_semantics where satisfies = satisfies
  by unfold_locales
interpretation hml_conj: lts_semantics where satisfies = satisfies_conj
  by unfold_locales
```

```
abbreviation hml_entails (infixr "⇛" 60) where ‹hml_entails ≡ hml.entails›
abbreviation hml_logical_eq (infix "⇚⇛" 60) where ‹hml_logical_eq ≡ hml.logical_eq›

abbreviation hml_conj_entails (infixr "∧⇛" 60) where ‹hml_conj_entails ≡ hml_conj.entails›
abbreviation hml_conj_logical_eq (infix "⇚∧⇛" 60) where ‹hml_conj_logical_eq ≡
hml_conj.logical_eq›

declare lts_semantics.entails_def[simp]
declare lts_semantics.eq_equality[simp]

abbreviation ‹HML_conj_neg φ ≡ (AND {undefined} (λi. HML_neg φ))›
abbreviation ‹HML_conj_pos φ ≡ (AND {undefined} (λi. HML_pos φ))›

lemma distinguishes_invertible:
  assumes ‹hml.distinguishes φ p q›
  shows ‹hml.distinguishes (HML_conj_neg φ) q p›
  using assms by auto

lemma conjunction_wrapping:
  shows ‹p ⊨ (HML_conj_pos φ) ⟷ p ⊨ φ›
  by auto
```

If two states are not HML equivalent then there must be a distinguishing formula.

```
lemma hml_distinctions:
  assumes ‹¬ hml.equivalent 𝒪 p q›
  shows ‹∃φ. hml.distinguishes φ p q›
  using assms distinguishes_invertible
  unfolding hml.equivalent_def hml.preordered_no_distinction
  by blast


end

end
```

4 Reachability Games

```
theory Equivalence_Games
  imports Strong_Equivalences
begin
```

A game is an unlabeled graph where vertices are partitioned into defender and attacker positions.

```
locale game =
  fixes
    game_move :: ‹'g ⇒ 'g ⇒ bool› (infix ‹↣› 80) and
    defender_position :: ‹'g ⇒ bool›
begin

abbreviation ‹attacker_position g ≡ ¬defender_position g›
abbreviation ‹options g ≡ {g'. g ↣ g'}›
```

Each player loses at a position if it were their turn but they are stuck.

```
definition ‹defender_loses g ≡ defender_position g ∧ options g = {}›
definition ‹attacker_loses g ≡ attacker_position g ∧ options g = {}›
```

A (positional) strategy is a function to select among the options at a position. That only possible moves are valid choices cannot be expressed in a HOL type. We express this by soundness predicates for attacker/defender strategies.

```
type_synonym ('g0) strategy = ‹'g0 ⇒ 'g0›

definition ‹sound_defender_strategy strat g ≡
  defender_position g ∧ options g ≠ {} ⟶ strat g ∈ options g›
definition ‹sound_attacker_strategy strat g ≡
  attacker_position g ∧ options g ≠ {} ⟶ strat g ∈ options g›
```

A play (fragment) is a sequence of game positions that follow a path of game moves.

```
inductive play :: ‹'g list ⇒ bool› where
  init: ‹play [g]› |
  step: ‹play (g # (g' # gg))› if ‹g ↣ g'› ‹play (g' # gg)›
```

We have defined plays in a way where there are no empty plays.

```
lemma no_empty_play:
  assumes ‹play []›
  shows ‹False›
  using assms play.cases by blast
```

A play follows a defender strategy if every every defender-controlled move obeys the strategy. (The type here, does not really ensure the position sequences to be plays and the strategies to be sound. This information should come from the context.)

```
fun play_follows_defender_strategy :: ‹'g list ⇒ ('g ⇒ 'g) ⇒ bool›
  where
  ‹play_follows_defender_strategy (g0 # g1 # pl) strat =
    ((if defender_position g0 then strat g0 = g1 else True)
    ∧ play_follows_defender_strategy (g1 # pl) strat)› |
  ‹play_follows_defender_strategy _ _ = True›

lemma play_extension:
  assumes
    ‹last pl ↣ g'›
    ‹play pl›
  shows
    ‹play (pl @ [g'])›
  using assms no_empty_play
  by (induct pl, auto,
      smt (verit, del_insts) append_Cons append_self_conv2
        list.sel(3) play.simps)

lemma play_follows_defender_strategy_extension_atk:
  assumes
    ‹play_follows_defender_strategy pl strat›
    ‹last pl ↣ g'›
    ‹attacker_position (last pl)›
  shows
    ‹play_follows_defender_strategy (pl @ [g']) strat›
  using assms
  by (induct pl, auto, smt (z3) append_self_conv2 hd_append list.distinct(1)
        list.sel(1,3) play_follows_defender_strategy.elims(1))
```

```
lemma play_follows_defender_strategy_extension_dfn:
  assumes
    ‹play_follows_defender_strategy pl strat›
    ‹defender_position (last pl)›
  shows
    ‹play_follows_defender_strategy (pl @ [strat (last pl)]) strat›
  using assms
  by (induct pl, force, smt (z3) append_Cons append_Nil
    game.play_follows_defender_strategy.simps(3) last.simps
    list.inject play_follows_defender_strategy.elims(1))

fun play_follows_attacker_strategy :: ‹'g list ⇒ ('g ⇒ 'g) ⇒ bool›
  where
  ‹play_follows_attacker_strategy (g0 # g1 # pl) strat =
    ((if attacker_position g0 then strat g0 = g1 else True)
    ∧ play_follows_attacker_strategy (g1 # pl) strat)›
| ‹play_follows_attacker_strategy _ _ = True›
```

A defender strategy is winning from a position if all plays following the strategy from there only lead to positions where the defender has moves and the strategy is sound. (In particular, the defender also wins if the game goes on forever.)

```
definition defender_winning_strategy :: ‹'g strategy ⇒ 'g ⇒ bool›
  where ‹defender_winning_strategy def_strat g ≡
  (∀pl. play pl ∧ hd pl = g ∧ play_follows_defender_strategy pl def_strat
  ⟶ sound_defender_strategy def_strat (last pl) ∧ ¬defender_loses (last pl))›

end
```

5 The Bisimulation Game

```
datatype ('a, 's) bisim_game_pos =
  Bisim_Attack 's 's
| Bisim_Defense 'a 's 's

fun (in lts) bisim_game_move ::
    ‹('a, 's) bisim_game_pos ⇒ ('a, 's) bisim_game_pos ⇒ bool› (infix ‹↣B› 80)
  where
  ‹Bisim_Attack p q ↣B Bisim_Attack p' q' =
    (p' = q ∧ q' = p)›
| ‹Bisim_Attack p q ↣B Bisim_Defense α p' q' =
    (p ⟼ α p' ∧ q' = q)›
| ‹Bisim_Defense α p q ↣B Bisim_Attack p' q' =
    (q ⟼ α q' ∧ p' = p)›
| ‹_ ↣B _ = False›

primrec bisim_defender_position where
  ‹bisim_defender_position (Bisim_Defense _ _ _) = True› |
  ‹bisim_defender_position (Bisim_Attack _ _) = False›

locale bisim_game =
  lts step +
  game ‹(↣B)› bisim_defender_position
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ⟼_ _› [70, 70, 70] 80)
begin
```

```
fun strategy_from_bisim ::
    ‹('s ⇒ 's ⇒ bool) ⇒ ('a, 's) bisim_game_pos strategy› where
  ‹strategy_from_bisim R (Bisim_Defense α p' q) =
    Bisim_Attack p' (SOME q'. R p' q' ∧ q ⟼ α q')›
| ‹strategy_from_bisim _ _ = undefined›

lemma bisim_implies_defender_winning_strategy:
  assumes ‹simulation R› ‹symp R› ‹R p q›
  shows ‹defender_winning_strategy (strategy_from_bisim R) (Bisim_Attack p q)›
  unfolding defender_winning_strategy_def
proof (clarify)
  fix pl
  assume ‹play pl› ‹hd pl = Bisim_Attack p q›
    ‹play_follows_defender_strategy pl (strategy_from_bisim R)›
  thus ‹sound_defender_strategy (strategy_from_bisim R) (last pl)
      ∧ ¬ defender_loses (last pl)›
    using ‹R p q›
  proof (induct pl arbitrary: p q rule: induct_list012)
    case (1 p q)
    then show ?case using no_empty_play by fastforce
  next
    case (2 g p q)
    then show ?case
      unfolding defender_loses_def sound_defender_strategy_def by simp
  next
    case (3 g g' pl p q)
    then have ‹g = Bisim_Attack p q› ‹g ↣B g'› using play.cases
      by (auto, fastforce)
    then obtain α p' where
      g'_spec: ‹g' = Bisim_Defense α p' q ∧ p ⟼ α p' ∨ g' = Bisim_Attack q p›
      by (smt (verit, best) bisim_game_move.elims(1) bisim_game_pos.distinct(1)
          bisim_game_pos.inject(1))
    then show ?case
    proof (cases pl, safe)
      assume g'_def: ‹g' = Bisim_Defense α p' q› ‹p ⟼α p'›
      then obtain q' where q'_spec: ‹q ⟼α q'› ‹R p' q'›
        using ‹simulation R› ‹R p q› unfolding simulation_def by blast
      then show ‹sound_defender_strategy (strategy_from_bisim R)
                  (last [g, Bisim_Defense α p' q])›
        unfolding sound_defender_strategy_def
        by (auto, metis (mono_tags, lifting) someI2_ex)
      show ‹defender_loses (last [g, Bisim_Defense α p' q]) ⟹ False›
        using q'_spec unfolding defender_loses_def
          using bisim_game_move.simps(3) by (auto, blast)
      have ‹play (g' # pl)› using ‹play (g # g' # pl)› using play.cases by blast
      fix g'' plrt
      assume pl_def: ‹pl = g'' # plrt›
      hence ‹play pl› using ‹play (g' # pl)› play.cases by auto
      have ‹g'' = (strategy_from_bisim R) g'› using g'_def(1) pl_def 3(5) by simp
      then obtain q'' where q''_spec:
          ‹g'' = Bisim_Attack p' q''› ‹q ⟼α q''› ‹R p' q''›
        using q'_spec g'_def by (auto, metis (mono_tags, lifting) someI_ex)
      then show
          ‹sound_defender_strategy (strategy_from_bisim R)
            (last (g # Bisim_Defense α p' q # g'' # plrt))›
          ‹defender_loses (last (g # Bisim_Defense α p' q # g'' # plrt)) ⟹ False›
        using 3 ‹play pl› unfolding defender_loses_def pl_def by auto
    next
      assume ‹g' = Bisim_Attack q p›
      then show
          ‹sound_defender_strategy (strategy_from_bisim R)
            (last [g, Bisim_Attack q p])›
          ‹defender_loses (last [g, Bisim_Attack q p]) ⟹ False›
        unfolding sound_defender_strategy_def defender_loses_def by auto
      fix g'' plrt
      assume pl_def: ‹pl = g'' # plrt›
      hence ‹play pl› using ‹play (g # g' # pl)› play.cases by auto
      have ‹R q p› using ‹R p q› ‹symp R› by (meson sympE)
      thus ‹sound_defender_strategy (strategy_from_bisim R)
            (last (g # Bisim_Attack q p # g'' # plrt))›
           ‹defender_loses (last (g # Bisim_Attack q p # g'' # plrt)) ⟹ False›
        using 3 play.cases
        unfolding sound_defender_strategy_def defender_loses_def pl_def
          ‹g' = Bisim_Attack q p›
        by (auto) blast+
    qed
  qed
qed

lemma defender_winning_strategy_implies_bisim:
  assumes
    ‹defender_winning_strategy strat (Bisim_Attack p0 q0)›
  defines
    ‹R == λp q. (∃pl.
      hd pl = (Bisim_Attack p0 q0)
      ∧ play pl
      ∧ play_follows_defender_strategy pl strat
      ∧ last pl = (Bisim_Attack p q))›
  shows
    ‹simulation R› ‹symp R› ‹R p0 q0›
proof -
  show ‹symp R›
    unfolding symp_def R_def
    using no_empty_play
    by (metis bisim_defender_position.simps(2) bisim_game_move.simps(1)
      game.play_extension game.play_follows_defender_strategy_extension_atk
      hd_append2 last_snoc)
  show ‹R p0 q0›
    using assms(1) unfolding defender_winning_strategy_def R_def
    using play.init by force
  show ‹simulation R›
    unfolding simulation_def
  proof safe
    fix p q a p'
    assume ‹p ⟼a p'› ‹R p q›
    then obtain pl where pl_spec:
        ‹hd pl = Bisim_Attack p0 q0›
        ‹play_follows_defender_strategy pl strat›
        ‹last pl = (Bisim_Attack p q)›
        ‹play pl›
      unfolding R_def by blast
    from ‹p ⟼a p'› have ‹Bisim_Attack p q ↣B Bisim_Defense a p' q› by simp
    hence defender_extension:
        ‹play_follows_defender_strategy (pl @ [Bisim_Defense a p' q]) strat›
        ‹play (pl @ [Bisim_Defense a p' q])›
      using play_follows_defender_strategy_extension_atk
        pl_spec play_extension[of pl ‹Bisim_Defense a p' q›] by auto
    hence defender_extension_hd:
      ‹hd (pl @ [Bisim_Defense a p' q]) = Bisim_Attack p0 q0›
      by (metis hd_append no_empty_play pl_spec(1,4))
    hence ‹options (Bisim_Defense a p' q) ≠ {}›
      using assms(1) pl_spec defender_extension
      unfolding defender_winning_strategy_def defender_loses_def
      by force
    then obtain q' where q'_spec:
        ‹strat (Bisim_Defense a p' q) = Bisim_Attack p' q'›
        ‹Bisim_Attack p' q' ∈ options (Bisim_Defense a p' q)›
      using defender_extension assms(1) defender_extension_hd
        bisim_defender_position.simps(1) bisim_game_move.simps(3,5)
      unfolding defender_winning_strategy_def sound_defender_strategy_def
      by (metis last_snoc mem_Collect_eq bisim_game_pos.exhaust)
    define long_play where
      ‹long_play ≡ pl @ [Bisim_Defense a p' q, Bisim_Attack p' q']›
    hence ‹hd long_play = Bisim_Attack p0 q0 ∧ play long_play
        ∧ play_follows_defender_strategy long_play strat
        ∧ last long_play = Bisim_Attack p' q'›
      using play_extension play_follows_defender_strategy_extension_dfn
        defender_extension q'_spec pl_spec no_empty_play last_snoc
      by (auto simp add: hd_append) fastforce+
    thus ‹∃q'. q ⟼a q' ∧ R p' q'›
      using q'_spec unfolding long_play_def R_def by force
  qed
qed

theorem bisim_game_characterization:
  shows
    ‹(∃strat. defender_winning_strategy strat (Bisim_Attack p q)) =
      bisimilar p q›
  using defender_winning_strategy_implies_bisim
    bisim_implies_defender_winning_strategy
  unfolding bisimilar_def
  by blast

end

end
theory Priced_HML
imports
  Hennessy_Milner_Logic
  "HOL-Library.Extended_Nat"
begin

datatype distinction_price =
  Price
    (obs_depth: ‹enat›)
    (conj_depth: ‹enat›)
    (pos_cl_height: ‹enat›)
    (pos_cl_height_2: ‹enat›)
    (neg_cl_height: ‹enat›)
    (neg_depth: ‹enat›)

lemma unfold_distinction_price:
 ‹dp = Price (obs_depth dp) (conj_depth dp) (pos_cl_height dp) (pos_cl_height_2
dp) (neg_cl_height dp) (neg_depth dp)›
  using distinction_price.collapse ..

instantiation distinction_price :: order
begin

fun less_eq_distinction_price :: ‹distinction_price ⇒ distinction_price ⇒ bool›
  where ‹less_eq_distinction_price (Price obs conjs posh posh2 negh negs) (Price
obs' conjs' posh' posh2' negh' negs') =
    (obs ≤ obs' ∧ conjs ≤ conjs' ∧ posh ≤ posh' ∧ posh2 ≤ posh2' ∧ negh ≤ negh'
∧ negs ≤ negs')›

definition less_distinction_price :: ‹distinction_price ⇒ distinction_price ⇒
bool› where
  ‹less_distinction_price pr pr' ≡
    (pr ≤ pr' ∧ ¬(pr' ≤ pr))›

instance
proof
  fix pr pr':: distinction_price
  show ‹(pr < pr') = (pr ≤ pr' ∧ ¬ pr' ≤ pr)›
    unfolding less_distinction_price_def by blast
next
  fix pr:: distinction_price
  show ‹pr ≤ pr›
    by (induct pr, auto)
next
  fix pr1 pr2 pr3:: distinction_price
  assume ‹pr1 ≤ pr2› ‹pr2 ≤ pr3›
  thus ‹pr1 ≤ pr3›
    using unfold_distinction_price
    by (smt (verit, best) dual_order.trans less_eq_distinction_price.simps)
next
  fix pr1 pr2:: distinction_price
  assume ‹pr1 ≤ pr2› ‹pr2 ≤ pr1›
  thus ‹pr1 = pr2›
    using less_eq_distinction_price.elims(2) by fastforce
qed
end

lemma less_eq_distinction_price_def:
  ‹(p1 ≤ p2) = (obs_depth p1 ≤ obs_depth p2 ∧ conj_depth p1 ≤ conj_depth p2 ∧
pos_cl_height p1 ≤ pos_cl_height p2 ∧ pos_cl_height_2 p1 ≤ pos_cl_height_2 p2 ∧
neg_cl_height p1 ≤ neg_cl_height p2 ∧ neg_depth p1 ≤ neg_depth p2)›
  by (induct p1, induct p2, auto)

primrec price_dec_obs where
  ‹price_dec_obs (Price obs conjs posh posh2 negh negs) = (Price (obs-1) conjs posh
posh2 negh negs)›
primrec price_cap_obs where
  ‹price_cap_obs cap (Price obs conjs posh posh2 negh negs) = (Price (min obs cap)
conjs posh posh2 negh negs)›
primrec price_cap_posh where
  ‹price_cap_posh cap (Price obs conjs posh posh2 negh negs) = (Price obs conjs
(min posh cap) posh2 negh negs)›
primrec price_dec_conjs where
  ‹price_dec_conjs (Price obs conjs posh posh2 negh negs) = (Price obs (conjs-1)
posh posh2 negh negs)›
primrec price_dec_negs where
  ‹price_dec_negs (Price obs conjs posh posh2 negh negs) = (Price obs conjs posh
posh2 negh (negs - 1))›

primrec formula_of_price :: ‹distinction_price ⇒ ('a,'i) hml_formula ⇒ bool›
  and conjunct_of_price :: ‹distinction_price ⇒ ('a,'i) hml_conjunct ⇒ bool›
  where
  ‹formula_of_price pr HML_true = True›
| ‹formula_of_price pr (⟨α⟩φ)  =
  (if obs_depth pr > 0 then formula_of_price (price_dec_obs pr) φ else False)›
| ‹formula_of_price pr (HML_conj I F) =
  (if conj_depth pr > 0 then
    (∃i∈I.
      conjunct_of_price (price_dec_conjs pr) (F i)
      ∧ (∀j∈(I-{i}). conjunct_of_price (price_cap_posh (pos_cl_height_2 pr) (price_dec_conjs
pr)) (F j)))
  else False)›
| ‹conjunct_of_price pr (HML_pos φ) =
  (case φ of (⟨α⟩φ') ⇒ formula_of_price (price_cap_obs (pos_cl_height pr) pr) φ
  | _ ⇒ False)›
| ‹conjunct_of_price pr (HML_neg φ) =
  (case φ of (⟨α⟩φ') ⇒
    if neg_depth pr > 0 then
      formula_of_price (price_dec_negs (price_cap_obs (neg_cl_height pr) pr)) φ
else False
  | _ ⇒ False)›

thm hml_formula_hml_conjunct.induct

lemma ediff1_le_mono:
  assumes ‹n ≤ (m::enat)›
  shows ‹n - 1 ≤ m - 1›
  using assms
  by (induct n, induct m, auto simp add: one_enat_def)

lemma emin_le_mono:
  assumes ‹n ≤ (m::enat)›
  shows ‹min a n ≤ min a m›
  using assms
  by (induct n, induct m, auto simp add: min.coboundedI2)

lemma conjuncts_require_observations:
  assumes ‹conjunct_of_price pr ψ› ‹obs_depth pr = 0›
  shows ‹False›
proof (cases ψ)
  case (HML_pos φ)
  then obtain α φ' where ‹φ = ⟨α⟩φ'› using assms(1)
    by (metis conjunct_of_price.simps(1) hml_formula.exhaust hml_formula.simps(10)
        hml_formula.simps(9))
```

```
 then show ?thesis using assms HML_pos
  by (metis conjunct_of_price.simps(1) distinction_price.collapse formula_of_price.simps(2)
    hml_formula.simps(11) less_numeral_extra(3) min_enat_simps(3) price_cap_obs.simps)
next
 case (HML_neg φ)
 then obtain α φ' where ‹φ = ⟨α⟩φ'› using assms(1)
  by (metis conjunct_of_price.simps(2) hml_formula.exhaust hml_formula.simps(10)
    hml_formula.simps(9))
 then show ?thesis using assms HML_neg
  by (metis conjunct_of_price.simps(2) distinction_price.collapse distinction_price.sel(1)
    formula_of_price.simps(2) hml_formula.simps(11) le_zero_eq linorder_neq_iff
min.cobounded1
    price_cap_obs.simps price_dec_negs.simps)
qed

lemma only_true_for_free:
 assumes ‹formula_of_price pr φ› ‹obs_depth pr = 0›
 shows ‹φ = HML_true›
 using assms
 by (induct pr; induct φ; auto;
   metis conjuncts_require_observations distinction_price.sel(1) price_dec_conjs.simps)

lemma deeper_conjuncts_require_observations:
 assumes ‹conjunct_of_price pr ψ› ‹obs_depth pr = 1›
 shows ‹∃α. ψ = (+⟨α⟩ HML_true) ∨ ψ = (-⟨α⟩ HML_true)›
proof (cases ψ)
 case (HML_pos φ)
 then obtain α φ' where
  ‹φ = ⟨α⟩φ'› ‹formula_of_price (price_dec_obs (price_cap_obs (pos_cl_height
pr) pr)) φ'›
  using assms(1)
  by (metis conjunct_of_price.simps(1) formula_of_price.simps(2) hml_formula.exhaust
    hml_formula.simps(10) hml_formula.simps(11) hml_formula.simps(9))
 hence ‹φ' = HML_true›
  using only_true_for_free ‹obs_depth pr = 1›
  by (metis add_diff_cancel_enat distinction_price.exhaust_sel distinction_price.sel(1)
    ediff1_le_mono i1_ne_infinity le_zero_eq min.cobounded1 one_eSuc plus_1_eSuc(1)
    price_cap_obs.simps price_dec_obs.simps)
 then show ?thesis using ‹φ = ⟨α⟩φ'› HML_pos by blast
next
 case (HML_neg φ)
 then obtain α φ' where
  ‹φ = ⟨α⟩φ'› ‹formula_of_price (price_dec_obs (price_dec_negs (price_cap_obs
(neg_cl_height pr) pr))) φ'›
  using assms(1)
  by (metis conjunct_of_price.simps(2) formula_of_price.simps(2) hml_formula.exhaust
    hml_formula.simps(10) hml_formula.simps(11) hml_formula.simps(9))
 hence ‹φ' = HML_true›
  using only_true_for_free ‹obs_depth pr = 1›
  by (metis (no_types, lifting) HML_neg assms(1) conjunct_of_price.simps(2)
    distinction_price.exhaust_sel distinction_price.sel(1) eSuc_minus_1 hml_formula.distinc
    hml_formula.simps(11) iless_eSuc0 min.cobounded1 one_eSuc order_le_less
price_cap_obs.simps
    price_dec_negs.simps price_dec_obs.simps)
 then show ?thesis using ‹φ = ⟨α⟩φ'› HML_neg by blast
qed
```

```
lemma neg_conjuncts_require_negations:
  assumes ‹conjunct_of_price pr (HML_neg φ)› ‹neg_depth pr = 0›
  shows ‹False›
  using assms
  by (auto, metis hml_formula.exhaust hml_formula.simps(10) hml_formula.simps(11)
hml_formula.simps(9) less_numeral_extra(3))

lemma price_closure:
  assumes
    ‹p01 ≤ p02›
    ‹formula_of_price p01 φ0›
  shows
    ‹formula_of_price p02 φ0›
proof -
  {
    fix φ :: ‹('a, 'i) hml_formula› and ψ :: ‹('a, 'i) hml_conjunct›
    have
      ‹⋀p1 p2. p1 ≤ p2 ⟹ formula_of_price p1 φ ⟹ formula_of_price p2 φ›
      ‹⋀p1 p2. p1 ≤ p2 ⟹ conjunct_of_price p1 ψ ⟹ conjunct_of_price p2 ψ›
    proof (induct φ and ψ)
      case HML_true
      then show ?case by simp
    next
      case (HML_conj I Ψ)
      from this(3) have ‹∀i∈I. ∃p0 ≤ p1. conjunct_of_price p0 (Ψ i)› apply (cases
p1) apply auto sorry
      hence ‹∀i∈I. conjunct_of_price p2 (Ψ i)› using HML_conj by auto
      then show ?case
        using HML_conj.prems less_eq_distinction_price.elims(2) apply (induct p2)
apply auto defer
        apply fastforce
        sorry
    next
      case (HML_obs α φ)
      hence ‹formula_of_price (price_dec_obs p1) φ› by (simp, argo)
      moreover have ‹price_dec_obs p1 ≤ price_dec_obs p2›
        using HML_obs(2) ediff1_le_mono by (cases p1, cases p2, auto)
      ultimately have ‹formula_of_price (price_dec_obs p2) φ› using HML_obs by
blast
      then show ?case
        using HML_obs.prems less_eq_distinction_price_def by force
    next
      case (HML_pos φ)
      then show ?case sorry
    next
      case (HML_neg φ)
      then show ?case sorry
    qed
  }
  thus ?thesis using assms by auto
qed

end
```

6 Priced Spectrum

```
theory Priced_Spectrum
imports
  Priced_HML
  HML_Spectrum

begin



context lts
begin

interpretation hml: lts_semantics where satisfies = satisfies
  by unfold_locales
interpretation hml_conj: lts_semantics where satisfies = satisfies_conj
  by unfold_locales

abbreviation ‹price_preordered pr ≡ hml.preordered (Collect (formula_of_price pr))›
```

6.1 Enabledness

A minimal and a way bigger coordinate to characterize enabledness preorder. (One could still increase either one of the last two dimensions, but would arrive at the same language.)

```
lemma enabledness_conjunctions_are_neutral:
  ‹hml.eq_distinctive
    (Collect (formula_of_price (Price 1 0 0 0 0 0)))
    (Collect (formula_of_price (Price 1 ∞ ∞ ∞ 0 0)))›
  unfolding hml.leq_distinctive_def
proof safe
  fix p q φ
  assume case_assms:
    ‹formula_of_price (Price 1 0 0 0 0 0) φ›
    ‹p ⊨ φ› ‹¬ q ⊨ φ›
  hence ‹φ ∈ Collect (formula_of_price (Price 1 ∞ ∞ ∞ 0 0))›
    using price_closure
    by (metis (no_types, opaque_lifting) enat_ord_simps(3) less_eq_distinction_price.simps
mem_Collect_eq order_refl)
  thus ‹∃φ' ∈Collect (formula_of_price (Price 1 ∞ ∞ ∞ 0 0)). hml.distinguishes
φ' p q›
    using case_assms by blast
next
  fix p q φ
  assume case_assms:
    ‹formula_of_price (Price 1 ∞ ∞ ∞ 0 0) φ›
    ‹p ⊨ φ› ‹¬ q ⊨ φ›
  show ‹∃φ'∈Collect (formula_of_price (Price 1 0 0 0 0 0)). hml.distinguishes φ'
p q›
  proof (cases φ)
    case HML_true
    then show ?thesis using case_assms by auto
  next
    case (HML_conj I Ψ)
```

```
      hence ‹∀i ∈ I. ∃ α. Ψ i = +⟨α⟩HML_true›
        using case_assms(1)
        by (simp, metis deeper_conjuncts_require_observations neg_conjuncts_require_negations
            distinction_price.sel(1) distinction_price.sel(6) member_remove remove_def)
      hence ‹∀i ∈ I. ∃ α. (Ψ i) = (+⟨α⟩HML_true) ∧
          formula_of_price (Price 1 0 0 0 0 0) (⟨α⟩HML_true)›
        by auto
      then show ?thesis using case_assms HML_conj
        by (metis mem_Collect_eq satisfies.simps(2) satisfies_conj.simps(1))
    next
      case (HML_obs α φ')
      hence ‹formula_of_price (Price 0 ∞ ∞ ∞ 0 0) φ'› using case_assms(1) by simp
      hence ‹φ' = HML_true› using only_true_for_free
        using distinction_price.sel(1) by blast
      hence ‹formula_of_price (Price 0 0 0 0 0 0) φ'› by simp
      hence ‹formula_of_price (Price 1 0 0 0 0 0) (⟨α⟩φ')› by simp
      then show ?thesis using case_assms(2,3) HML_obs by blast
    qed
  qed
```

6.2 Traces

```
lemma trace_observations_respect_prices:
  assumes
    ‹observations_traces φ›
  shows
    ‹formula_of_price (Price ∞ 0 0 0 0 0) φ›
  using assms by (induct, auto)

lemma trace_prices_imply_observations:
  assumes
    ‹formula_of_price (Price ∞ 0 0 0 0 0) φ›
  shows
    ‹observations_traces φ›
  using assms by (induct φ, auto simp add: observations_traces.intros)

theorem traces_priced_characterization:
    ‹(≲T) = price_preordered (Price ∞ 0 0 0 0 0)›
  using trace_observations_respect_prices trace_prices_imply_observations
  unfolding observations_traces_characterizes_trace_preorder by blast
```

6.3 Simulation

```
lemma simulation_prices_imply_observations:
fixes
  φ :: ‹('a, 's) hml_formula› and
  ψ :: ‹('a, 's) hml_conjunct›
shows
  ‹formula_of_price (Price ∞ ∞ ∞ ∞ 0 0) φ ⟹ observations_simulation φ›
  ‹conjunct_of_price (Price ∞ ∞ ∞ ∞ 0 0) ψ ⟹ observations_simulation_conj
ψ›
proof (induct φ and ψ)
  case HML_true
  then show ?case using observations_simulation_observations_simulation_conj.intros
by simp
next
```

```
  case (HML_conj I Ψ)
  hence ‹I ≠ {}›
    using observations_simulation.cases by auto
  then show ?case using HML_conj
    apply (auto)
    by (metis Diff_iff ‹I ≠ {}› observations_simulation.simps range_eqI singletonD)
next
  case (HML_obs α φ)
  then show ?case
    using observations_simulation_observations_simulation_conj.intros by auto
next
  case (HML_pos φ)
  show ?case
  proof (cases φ)
    case HML_true
    then show ?thesis using HML_pos
      by simp
  next
    case (HML_conj I' Ψ')
    then show ?thesis using HML_pos
      by simp
  next
    case (HML_obs α φ')
    then show ?thesis using HML_pos observations_simulation_conj.simps
      by (metis conjunct_of_price.simps(1) distinction_price.sel(3) hml_formula.distinct(3,5)
        hml_formula.simps(11) observations_simulation.simps min_enat_simps(5)
        price_cap_obs.simps)
  qed
next
  case (HML_neg φ)
  from this(2) have False by (cases φ, auto)
  thus ?case ..
qed

lemma simulation_observations_respect_prices:
fixes
  φ :: ‹('a, 's) hml_formula› and
  ψ :: ‹('a, 's) hml_conjunct›
shows
  ‹observations_simulation φ ⟹ formula_of_price (Price ∞ ∞ ∞ ∞ 0 0) φ›
  ‹observations_simulation_conj ψ ⟹ conjunct_of_price (Price ∞ ∞ ∞ ∞ 0 0)
ψ›
proof (induct φ and ψ)
  case HML_true
  then show ?case by simp
next
  case (HML_conj I Ψ)
  hence ‹I ≠ {}›
    using observations_simulation.cases by auto
  then show ?case using HML_conj
    apply (auto)
    by (metis DiffD1 hml_formula.distinct(1,5) hml_formula.inject(1) observations_simulation.si
range_eqI)
next
  case (HML_obs α φ)
  then show ?case
```

```
    using lts.observations_simulation.cases by auto
next
  case (HML_pos φ)
  then obtain α φ' where α_φ': ‹φ = (⟨α⟩φ') ∧ observations_simulation φ'›
    using lts.observations_simulation_conj.simps
    by (metis hml_conjunct.inject(1))
  hence ‹conjunct_of_price (Price ∞ ∞ ∞ ∞ 0 0) (+(⟨α⟩φ'))›
    using HML_pos.hyps lts.observations_simulation.simps by auto
  thus ?case using α_φ' by blast
next
  case (HML_neg φ)
  from this(2) have False
    using observations_simulation_conj.simps by blast
  thus ?case ..
qed

theorem simulation_priced_characterization:
    ‹(≲S) = price_preordered (Price ∞ ∞ ∞ ∞ 0 0)›
  using simulation_observations_respect_prices simulation_prices_imply_observations
  unfolding observations_simulation_characterize_simulation_preorder by blast

end

end
```

References

[1] B. Bisping and D. N. Jansen. Linear-time–branching-time spectroscopy accounting for silent steps, 2023.