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 **4**
2.1 Trivial notions of equality 4
2.1.1 Identity 4
2.1.2 Universal equality 4
2.2 Trace Equality 4
2.3 Isomorphism 5
2.4 Simulation preorder and equivalence 5

3 Hennessy–Milner Logic **7**

4 Reachability Games **8**

5 The Bisimulation Game **10**

6 Priced Spectrum **13**
6.1 Enabledness 14
6.2 Traces 14
6.3 Simulation 14
[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›
  ⟨proof⟩

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›
  ⟨proof⟩

lemma path_implies_trace:
  assumes ‹path pp›
  shows ‹∃tr. (hd pp) ⟼$ tr (last pp)›
  ⟨proof⟩

lemma trace_implies_path:
  assumes ‹p ⟼$ tr p'›
```

```
  shows ‹∃pp. path pp ∧ hd pp = p ∧ last pp = p'›
  ⟨proof⟩

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''›
  ⟨proof⟩

lemma silent_reachable_preorder:
   ‹reflp (↠)›
   ‹transp (↠)›
  ⟨proof⟩
```

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›
```

```
  ⟨proof⟩

lemma weak_seq_tau_transparent:
  assumes ‹p ↠⟼$ tr p'›
  shows ‹p ↠⟼$ (filter (λα. α ≠ τ) tr) p'›
  ⟨proof⟩

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›
  ⟨proof⟩

lemma non_identity:
  assumes ‹p ≠ q›
  shows ‹¬ identical p q›
  ⟨proof⟩
```

It's an equivalence.

```
lemma identity_equivalence:
  shows ‹equivp identical›
⟨proof⟩
```

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›
  ⟨proof⟩
```

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)›
  ⟨proof⟩

lemma trace_equivalence_equiv:
  shows ‹equivp trace_equivalent›
⟨proof⟩
```

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⟩
```

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'›
⟨proof⟩
```

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›
  ⟨proof⟩

corollary iso_implies_trace_eq:
  assumes ‹is_isomorphic_to p q›
  shows ‹trace_equivalent p q›
  ⟨proof⟩
```

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›
  ⟨proof⟩

lemma simulation_composition:
  assumes
    ‹simulation R1›
    ‹simulation R2›
  shows
    ‹simulation (λp q. ∃p’. R1 p p’ ∧ R2 p’ q)›
  ⟨proof⟩

lemma simulation_union:
  assumes
    ‹simulation R1›
    ‹simulation R2›
  shows
    ‹simulation (λp q. R1 p q ∨ R2 p q)›
  ⟨proof⟩

lemma simulation_preorder_transitive:
  shows ‹transp (≲S)›
  ⟨proof⟩

lemma iso_is_sim:
  shows ‹simulation is_isomorphic_to›
  ⟨proof⟩

corollary iso_implies_sim:
  assumes ‹is_isomorphic_to p q›
  shows ‹simulated_by p q›
  ⟨proof⟩

lemma sim_implies_trace_preord:
  assumes ‹p ≲S q›
  shows ‹p ≲T q›
  ⟨proof⟩


lemma sim_eq_implies_trace_eq:
  assumes ‹p ≃S q›
  shows ‹p ≃T q›
  ⟨proof⟩
```

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›
  ⟨proof⟩

lemma bisimilarity_equiv:
  shows ‹equivp (≃B)›
⟨proof⟩
```

Bisimilarity is a bisimulation.

```
lemma bisim_bisim:
  shows ‹simulation bisimilar ∧ symp bisimilar›
  ⟨proof⟩

lemma bisim_implies_sim:
  assumes ‹p ≃B q›
  shows ‹p ≃S q›
  ⟨proof⟩

lemma iso_implies_bisim:
  assumes ‹p ≃ISO q›
  shows ‹p ≃B q›
⟨proof⟩

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
  ⟨proof⟩
interpretation hml_conj: lts_semantics where satisfies = satisfies_conj
  ⟨proof⟩

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›
  ⟨proof⟩

lemma conjunction_wrapping:
  shows ‹p ⊨ (HML_conj_pos φ) ⟷ p ⊨ φ›
  ⟨proof⟩
```

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›
  ⟨proof⟩


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›
  ⟨proof⟩
```

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'])›
  ⟨proof⟩

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›
  ⟨proof⟩

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›
  ⟨proof⟩

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)›
  ⟨proof⟩

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⟩

theorem bisim_game_characterization:
  shows
    ‹(∃strat. defender_winning_strategy strat (Bisim_Attack p q)) =
      bisimilar p q›
  ⟨proof⟩

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)›
  ⟨proof⟩

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⟩
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)›
  ⟨proof⟩

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›
  ⟨proof⟩

lemma emin_le_mono:
  assumes ‹n ≤ (m::enat)›
  shows ‹min a n ≤ min a m›
  ⟨proof⟩

lemma conjuncts_require_observations:
  assumes ‹conjunct_of_price pr ψ› ‹obs_depth pr = 0›
  shows ‹False›
⟨proof⟩

lemma only_true_for_free:
  assumes ‹formula_of_price pr φ› ‹obs_depth pr = 0›
  shows ‹φ = HML_true›
  ⟨proof⟩

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⟩

lemma neg_conjuncts_require_negations:
  assumes ‹conjunct_of_price pr (HML_neg φ)› ‹neg_depth pr = 0›
  shows ‹False›
  ⟨proof⟩

lemma price_closure:
  assumes
    ‹p01 ≤ p02›
    ‹formula_of_price p01 φ0›
  shows
    ‹formula_of_price p02 φ0›
⟨proof⟩

end
```

6 Priced Spectrum

```
theory Priced_Spectrum
imports
  Priced_HML
  HML_Spectrum

begin



context lts
begin
```

```
interpretation hml: lts_semantics where satisfies = satisfies
  ⟨proof⟩
interpretation hml_conj: lts_semantics where satisfies = satisfies_conj
  ⟨proof⟩

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)))›
  ⟨proof⟩
```

6.2 Traces

```
lemma trace_observations_respect_prices:
  assumes
    ‹observations_traces φ›
  shows
    ‹formula_of_price (Price ∞ 0 0 0 0 0) φ›
  ⟨proof⟩

lemma trace_prices_imply_observations:
  assumes
    ‹formula_of_price (Price ∞ 0 0 0 0 0) φ›
  shows
    ‹observations_traces φ›
  ⟨proof⟩

theorem traces_priced_characterization:
    ‹(≲T) = price_preordered (Price ∞ 0 0 0 0 0)›
  ⟨proof⟩
```

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⟩

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⟩

theorem simulation_priced_characterization:
    ‹(≲S) = price_preordered (Price ∞ ∞ ∞ ∞ 0 0)›
  ⟨proof⟩

end

end
```

References

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