The **Weak Spectroscopy Game** to Characterize Behavioral Equivalences

Lisa A. Barthel[1,2], Leonard M. Hübner[1], Caroline Lemke[1,3], Karl P. P. Mattes[1], Lenard Mollenkopf[1], and Benjamin Bisping[1,4]

[1]Technische Universität Berlin, Germany
[2]Technische Universität Ilmenau, Germany
[3]Carl von Ossietzky Universität Oldenburg, Germany
[4]Télécom SudParis, Institut Polytechnique de Paris, France

February 7, 2026

Abstract

We provide an Isabelle/HOL formalization of Bisping and Jansen's weak spectroscopy game [4], which can be used to simultaneously characterize and decide a hierarchy of behavioral equivalences for systems with internal behavior. This is valuable for applications in concurrency theory and formal verification where equivalences and distinctions of the "linear-time–branching-time spectrum" are a recurring topic.

This entry contains a game characterization of most behavioral equivalences from stability-respecting branching bisimilarity to weak trace equivalence. Technically, the results link distinguishing sublanguages of Hennessy–Milner logic to winning attacker budgets in an energy game through an eight-dimensional measurement of syntactic features appearing in formulas.

Overview. This formalization provides theoretical underpinnings of https://equiv.io, a tool to *decide all behavioral equivalences at once*. By phrasing *equivalences as energy games*, one obtains a uniform way to handle a wide range of equivalences in van Glabbeek's *linear-time–branching-time spectrum* [6, 7]. In particular, we treat systems with *silent* τ-steps, which usually arise because of abstraction from internal behavior, for instance, when modeling communication protocols or distributed systems using transition systems.

This formalization follows Bisping and Jansen's *weak spectroscopy game* [4], respectively the proofs from the arXiv version [3].

- Section 1 provides some basics on transition systems with internal behavior.
- Sections 2 and 3 define a version of Hennessy–Milner logic for systems with internal behavior and a syntactic metric to select sublanguages of it through coordinates.
- Sections 4 to 6 prove certain coordinates to correspond to weak trace equivalence, η-bisimilarity, η-similarity and branching bisimilarity as well as their stable variants. (As the relationship is established through modal logics, the results can be understood as Hennessy–Milner theorems [5].)
- Sections 7 to 9 introduce the weak spectroscopy game and prove that winning attacker energies in the game correspond to coordinates of distinguishing formulas according to the syntactic expressiveness metric.

The broader project of *deciding all equivalences at once* is outlined in Bisping's PhD thesis [2]. There, one can also find more gentle introductions to the topic and to the game-theoretic approach in general. The energy game approach is due to [1].

Acknowledgments. Several proofs in this document follow pen-and-paper proofs by David N. Jansen.

Contents

1 Labeled Transition Systems

```
theory Labeled_Transition_Systems
  imports Main
begin
```

1.1 Base LTS

The locale `LTS` represents a labeled transition system consisting of a set of states $\mathcal{P}$, a set of actions Σ, and a transition relation $\mapsto \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$. We formalize the sets of states and actions by the type variables `'s` and `'a`. An LTS is then determined by the transition relation `step`.

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

One may lift `step` to sets of states, written as `P ↦S α Q`.

```
abbreviation step_setp (‹_ ↦S _ _› [70,70,70] 80) where
  ‹P ↦S α Q ≡ (∀q ∈ Q. ∃p ∈ P. p ↦ α q) ∧ (∀p ∈ P. ∀q. p ↦ α q ⟶ q ∈ Q)›
```

The set of α-derivatives for a set of states `P`.

```
definition step_set :: ‹'s set ⇒ 'a ⇒ 's set› where
  ‹step_set P α ≡ { q . ∃p ∈ P. p ↦ α q }›
```

The set of possible α-steps for a set of states `P` is an instance of `step` lifted to sets of steps.

```
lemma step_set_is_step_set: ‹P ↦S α (step_set P α)›
  ⟨proof⟩
```

The lifted `step_setp` (`P ↦S α Q`) is therefore this set `Q`.

```
lemma step_set_eq:
  assumes ‹P ↦S α Q›
  shows ‹Q = step_set P α›
  ⟨proof⟩
```

```
end — of locale lts
```

1.2 Labeled Transition Systems with Silent Steps

We formalize labeled transition systems with silent steps as an extension of ordinary labeled transition systems with a fixed internal action τ.

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

The paper [3] introduces a transition $p \xrightarrow{(\alpha)} p'$ if $p \xrightarrow{\alpha} p'$, or if $\alpha = \tau$ and $p = p'$. We define `soft_step` analogously and provide the notation `p ↦a α p'`.

```
abbreviation soft_step (‹_ ↦a _ _› [70,70,70] 80) where
  ‹p ↦a α q ≡ p ↦α q ∨ (α = τ ∧ p = q)›

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

If `p'` is silent-reachable from `p` and there is a τ-transition from `p'` to `p''` then `p''` is silent reachable from `p`.

```
lemma silent_reachable_append_τ: ‹p ↠ p' ⟹ p' ↦ τ p'' ⟹ p ↠ p''›
```
⟨*proof*⟩

The relation `(↠)` is transitive.

```
lemma silent_reachable_trans:
  assumes
    ‹p ↠ p'›
    ‹p' ↠ p''›
  shows
    ‹p ↠ p''›
```
⟨*proof*⟩

The relation `silent_reachable_loopless` is a variation of `(↠)` that does not use self-loops.

```
inductive silent_reachable_loopless :: ‹'s ⇒ 's ⇒ bool›  (infix ‹↠L› 80)
  where
    ‹p ↠L p› |
    ‹p ↠L p''› if ‹p ↦ τ p'› and ‹p' ↠L p''› and ‹p ≠ p'›
```

If a state `p'` is `(↠)` from `p` it is also `(↠L)`.

```
lemma silent_reachable_impl_loopless:
  assumes ‹p ↠ p'›
  shows ‹p ↠L p'›
  ⟨proof⟩

lemma tau_chain_reachabilty:
  assumes ‹∀i < length pp - 1.  pp!i ↦ τ pp!(Suc i)›
  shows ‹∀j < length pp. ∀i ≤ j. pp!i ↠ pp!j›
```
⟨*proof*⟩

A state `p` can reach `p'` weakly by performing an α-transition, possibly proceeded and followed by any number of τ-transitions.

```
definition weak_step (‹_ ↠↦↠ _ _› [70, 70, 70] 80) where
  ‹p  ↠↦↠ α p' ≡ if α = τ
                   then p ↠ p'
                   else ∃p1 p2. p ↠ p1 ∧ p1 ↦ α p2 ∧ p2 ↠ p'›

lemma silent_prepend_weak_step: ‹p ↠ p' ⟹ p' ↠↦↠ α p'' ⟹ p ↠↦↠ α p''›
  ⟨proof⟩
```

A sequence of `weak_steps` from one state `p` to another `p'`.

```
inductive weak_step_sequence :: ‹'s ⇒ 'a list ⇒ 's ⇒ bool› (‹_ ↠↦↠$ _ _› [70,70,70]
80) where
  ‹p ↠↦↠$ [] p'› if ‹p ↠ p'› |
  ‹p ↠↦↠$ (α#rt) p'''› if ‹p ↠↦↠ α p'› ‹p' ↠↦↠$ rt p'''›

lemma weak_step_sequence_trans:
  assumes ‹p ↠↦↠$ tr_1 p'› and ‹p' ↠↦↠$ tr_2 p''›
  shows ‹p ↠↦↠$ (tr_1 @ tr_2) p''›
  ⟨proof⟩
```

The weak traces of a state are all possible sequences of weak transitions that can be performed.

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

The empty trace is in `weak_traces` for all states.

```
lemma empty_trace_allways_weak_trace:
  shows ‹[] ∈ weak_traces p›
  ⟨proof⟩
```

τ can be prepended to any weak trace.

```
lemma prepend_τ_weak_trace:
  assumes ‹tr ∈ weak_traces p›
  shows ‹(τ # tr) ∈ weak_traces p›
  ⟨proof⟩

lemma silent_prepend_weak_traces:
  assumes
    ‹p ↠ p'›
    ‹tr ∈ weak_traces p'›
  shows
    ‹tr ∈ weak_traces p›
  ⟨proof⟩
```

If there is an α-transition from `p` to `p'`, and `p'` has a weak trace `tr`, then the sequence `(α # tr)` is a valid (weak) trace of `p`.

```
lemma step_prepend_weak_traces:
  assumes
    ‹p ↦ α p'›
    ‹tr ∈ weak_traces p'›
  shows
    ‹(α # tr) ∈ weak_traces p›
  ⟨proof⟩
```

A state is weakly trace pre-ordered to another other, `weakly_trace_preordered` denoted by $\lesssim$`WT` if all its traces can also be observed from the second process.

```
definition weakly_trace_preordered (infix ‹≲WT› 60) where
  ‹p ≲WT q ≡ weak_traces p ⊆ weak_traces q›

definition weakly_trace_equivalent (infix ‹≃WT› 60) where
  ‹p ≃WT q ≡ p ≲WT q ∧ q ≲WT p›
```

Just like `step_setp`, one can lift `(↠)` to sets of states.

```
abbreviation silent_reachable_setp (infix ‹↠S› 80) where
  ‹P ↠S P' ≡ ((∀p' ∈ P'. ∃p ∈ P. p ↠ p') ∧ (∀p ∈ P. ∀p'. p ↠ p' ⟶ p' ∈ P'))›

definition silent_reachable_set :: ‹'s set ⇒ 's set› where
  ‹silent_reachable_set P ≡ { q . ∃p ∈ P. p ↠ q }›

lemma sreachable_set_is_sreachable: ‹P ↠S (silent_reachable_set P)›
  ⟨proof⟩

lemma sreachable_set_eq:
  assumes ‹P ↠S Q›
  shows ‹Q = silent_reachable_set P›
  ⟨proof⟩
```

We likewise lift `soft_step` to sets of states.

```
abbreviation soft_step_setp (‹_ ↦aS _ _› [70,70,70] 80) where
  ‹P ↦aS α Q ≡ (∀q ∈ Q. ∃p ∈ P. p ↦a α q) ∧ (∀p ∈ P. ∀q. p ↦a α q ⟶ q ∈ Q)›

definition soft_step_set :: ‹'s set ⇒ 'a ⇒ 's set› where
  ‹soft_step_set P α ≡ { q . ∃p ∈ P. p ↦a α q }›

lemma soft_step_set_is_soft_step_set:
  ‹P ↦aS α (soft_step_set P α)›
  ⟨proof⟩

lemma exactly_one_soft_step_set:
```

```
  ‹∃!Q. P ↦aS α Q›
```
⟨*proof*⟩

```
lemma soft_step_set_eq:
  assumes ‹P ↦aS α Q›
  shows ‹Q = soft_step_set P α›
```
⟨*proof*⟩

A state is stable if it cannot make any further internal steps.

```
abbreviation ‹stable_state p ≡ ∀p'. ¬(p ↦ τ p')›

lemma stable_state_stable:
  assumes ‹stable_state p› ‹p ↠ p'›
  shows ‹p = p'›
```
⟨*proof*⟩

```
definition stability_respecting :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹stability_respecting R ≡ ∀ p q. R p q ∧ stable_state p ⟶
    (∃q'. q ↠ q' ∧ R p q' ∧ stable_state q')›

end — of locale lts_tau

end
```

1.3 Modal Logics on LTS

We here supply abstract definitions that would work for all modal logics one might define over an LTS. In particular, this contains mechanisms to derive equivalences from sublogics.

```
theory LTS_Semantics
  imports
    Labeled_Transition_Systems
begin

locale lts_semantics = lts step
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80) +
  fixes models :: ‹'s ⇒ 'formula ⇒ bool›
begin

definition entails :: ‹'formula ⇒ 'formula ⇒ bool› where
  entails_def[simp]: ‹entails φl φr ≡ (∀p. (models p φl) ⟶ (models p φr))›

definition logical_eq :: ‹'formula ⇒ 'formula ⇒ bool› where
  logical_eq_def[simp]: ‹logical_eq φl φr ≡ entails φl φr ∧ entails φr φl›
```

Formula implication is a pre-order.

```
lemma entails_preord: ‹reflp (entails)› ‹transp (entails)›
```
⟨*proof*⟩

```
lemma eq_equiv: ‹equivp logical_eq›
```
⟨*proof*⟩

Formula equivalence is a biimplication on the models predicate.

```
lemma eq_equality[simp]: ‹(logical_eq φl φr) = (∀p. models p φl ⟷ models p φr)›
```
⟨*proof*⟩

```
lemma logical_eqI[intro]:
  assumes
    ‹⋀s. models s φl ⟹ models s φr›
    ‹⋀s. models s φr ⟹ models s φl›
```

```
  shows
    ‹logical_eq φl φr›
  ⟨proof⟩

definition distinguishes :: ‹'formula ⇒ 's ⇒ 's ⇒ bool› where
  distinguishes_def[simp]:
  ‹distinguishes φ p q ≡ models p φ ∧ ¬(models q φ)›

definition distinguishes_from :: ‹'formula ⇒ 's ⇒ 's set ⇒ bool› where
  distinguishes_from_def[simp]:
  ‹distinguishes_from φ p Q ≡ models p φ ∧ (∀q ∈ Q. ¬(models q φ))›

lemma distinction_unlifting:
  assumes
    ‹distinguishes_from φ p Q›
  shows
    ‹∀q∈Q. distinguishes φ p q›
  ⟨proof⟩

lemma no_distinction_fom_self:
  assumes
    ‹distinguishes φ p p›
  shows
    ‹False›
  ⟨proof⟩

lemma dist_equal_dist:
  assumes ‹logical_eq φl φr›
      and ‹distinguishes φl p q›
    shows ‹distinguishes φr p q›
  ⟨proof⟩

abbreviation model_set :: ‹'formula ⇒ 's set› where
  ‹model_set φ ≡ {p. models p φ}›
```

1.4 Preorders and Equivalences on Processes Derived from Formula Sets

A set of formulas pre-orders two processes p and q if, for all formulas in this set, the fact that p satisfies a formula means that q must also satisfy this formula.

```
definition preordered :: ‹'formula set ⇒ 's ⇒ 's ⇒ bool› where
  preordered_def[simp]:
  ‹preordered φs p q ≡ ∀φ ∈ φs. models p φ ⟶ models q φ›
```

If a set of formulas pre-orders two processes p and q, then no formula in that set may distinguish p from q.

```
lemma preordered_no_distinction:
  ‹preordered φs p q = (∀φ ∈ φs. ¬(distinguishes φ p q))›
  ⟨proof⟩
```

A formula set derived pre-order is a pre-order.

```
lemma preordered_preord:
  ‹reflp (preordered φs)›
  ‹transp (preordered φs)›
  ⟨proof⟩
```

A set of formulas equates two processes if it pre-orders these two processes in both directions.

```
definition equivalent :: ‹'formula set ⇒ 's ⇒ 's ⇒ bool› where
  equivalent_def[simp]:
  ‹equivalent φs p q ≡ preordered φs p q ∧ preordered φs q p›
```

If a set of formulas equates two processes, then no formula in that set may distinguish them in any direction.

```
lemma equivalent_no_distinction: ‹equivalent φs p q
     = (∀φ ∈ φs. ¬(distinguishes φ p q) ∧ ¬(distinguishes φ q p))›
```
⟨*proof*⟩

A formula-set-derived equivalence is an equivalence.

```
lemma equivalent_equiv: ‹equivp (equivalent φs)›
```
⟨*proof*⟩

`end` — of context `lts_semantics`

`end`

2 Hennessy–Milner Logic for Stability-Respecting Branching Bisimilarity

```
theory HML_SRBB
  imports LTS_Semantics
begin
```

This section describes a variant of Hennessy–Milner logic that characterizes stability-respecting branching bisimilarity (SRBB).

The following mutually-recursive datatype family describes a grammar of `HML_SRBB` formulas.

```
datatype
  ('act, 'i) hml_srbb =
    TT |
    Internal <('act, 'i) hml_srbb_inner> |
    ImmConj <'i set> <'i ⇒ ('act, 'i) hml_srbb_conjunct>
and
  ('act, 'i) hml_srbb_inner =
    Obs 'act <('act, 'i) hml_srbb> |
    Conj <'i set> <'i ⇒ ('act, 'i) hml_srbb_conjunct> |
    StableConj <'i set> <'i ⇒ ('act, 'i) hml_srbb_conjunct> |
    BranchConj 'act <('act, 'i) hml_srbb>
               <'i set> <'i ⇒ ('act, 'i) hml_srbb_conjunct>
and
  ('act, 'i) hml_srbb_conjunct =
    Pos <('act, 'i) hml_srbb_inner> |
    Neg <('act, 'i) hml_srbb_inner>
```

The constructors correspond to more conventional notation of HML as follows:

- `hml_srbb` (members usually referred to as φ):
  - `TT` encodes $\top$
  - `Internal` χ encodes $\langle\varepsilon\rangle\chi$
  - `ImmConj I` ψ`s` encodes $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
- `hml_srbb_inner` (usually χ):
  - `Obs` α φ encodes $(\alpha)\varphi$
  - `Conj I` ψ`s` encodes $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
  - `StableConj I` ψ`s` encodes $\neg\langle\tau\rangle\top \wedge \bigwedge_{i\in \mathtt{I}} \psi s(i)$
  - `BranchConj` α φ `I` ψ`s` encodes $(\alpha)\varphi \wedge \bigwedge_{i\in \mathtt{I}} \psi s(i)$
- `hml_srbb_conjunct` (usually ψ):
  - `Pos` χ encodes $\langle\varepsilon\rangle\chi$
  - `Neg` χ encodes $\neg\langle\varepsilon\rangle\chi$

2.1 Semantics of HML_{SRBB} Formulas

This section describes how semantic meaning is assigned to HML_{SRBB} formulas in the context of a LTS. We define what it means for a process `p` to satisfy an HML_{SRBB} formula φ, written as `p` $\models$`SRBB` φ.

```
context lts_tau
begin
```

```
primrec
      hml_srbb_models :: ‹'s ⇒ ('a, 's) hml_srbb ⇒ bool› (infixl ‹⊨SRBB› 60)
  and hml_srbb_inner_models :: ‹'s ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  and hml_srbb_conjunct_models :: ‹'s ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool› where
  ‹hml_srbb_models state TT =
    True› |
  ‹hml_srbb_models state (Internal χ) =
    (∃p'. state ↠ p' ∧ (hml_srbb_inner_models p' χ))› |
  ‹hml_srbb_models state (ImmConj I ψs) =
    (∀i∈I. hml_srbb_conjunct_models state (ψs i))› |

  ‹hml_srbb_inner_models state (Obs a φ) =
    ((∃p'. state ↦ a p' ∧ hml_srbb_models p' φ) ∨ a = τ ∧ hml_srbb_models state φ)› |
  ‹hml_srbb_inner_models state (Conj I ψs) =
    (∀i∈I. hml_srbb_conjunct_models state (ψs i))› |
  ‹hml_srbb_inner_models state (StableConj I ψs) =
    ((∄p'. state ↦ τ p') ∧ (∀i∈I. hml_srbb_conjunct_models state (ψs i)))› |
  ‹hml_srbb_inner_models state (BranchConj a φ I ψs) =
    (((∃p'. state ↦ a p' ∧ hml_srbb_models p' φ) ∨ a = τ ∧ hml_srbb_models state φ)
    ∧ (∀i∈I. hml_srbb_conjunct_models state (ψs i)))› |

  ‹hml_srbb_conjunct_models state (Pos χ) =
    (∃p'. state ↠ p' ∧ hml_srbb_inner_models p' χ)› |
  ‹hml_srbb_conjunct_models state (Neg χ) =
    (∄p'. state ↠ p' ∧ hml_srbb_inner_models p' χ)›

sublocale lts_semantics ‹step› ‹hml_srbb_models› ⟨proof⟩
sublocale hml_srbb_inner: lts_semantics where models = hml_srbb_inner_models ⟨proof⟩
sublocale hml_srbb_conj: lts_semantics where models = hml_srbb_conjunct_models ⟨proof⟩
```

2.2 Distinguishing Formulas

```
lemma verum_never_distinguishes:
  ‹¬ distinguishes TT p q›
  ⟨proof⟩
```

If $\bigwedge_{i \in I} \psi s(i)$ distinguishes `p` from `q`, then there must be at least one conjunct in this conjunction that distinguishes `p` from `q`.

```
lemma srbb_dist_imm_conjunction_implies_dist_conjunct:
  assumes ‹distinguishes (ImmConj I ψs) p q›
  shows ‹∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
  ⟨proof⟩

lemma srbb_dist_conjunction_implies_dist_conjunct:
  assumes ‹hml_srbb_inner.distinguishes (Conj I ψs) p q›
  shows ‹∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
  ⟨proof⟩

lemma srbb_dist_branch_conjunction_implies_dist_conjunct_or_branch:
  assumes
    ‹hml_srbb_inner.distinguishes (BranchConj α φ I ψs) p q›
  shows
    ‹(∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q)
       ∨ hml_srbb_inner.distinguishes (Obs α φ) p q›
  ⟨proof⟩

lemma srbb_dist_conjunct_implies_dist_imm_conjunction:
  assumes
    ‹i∈I›
    ‹hml_srbb_conj.distinguishes (ψs i) p q›
```

```
    ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
  shows
    ‹distinguishes (ImmConj I ψs) p q›
  ⟨proof⟩

lemma srbb_dist_conjunct_implies_dist_conjunction:
  assumes
    ‹i∈I›
    ‹hml_srbb_conj.distinguishes (ψs i) p q›
    ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
  shows
    ‹hml_srbb_inner.distinguishes (Conj I ψs) p q›
  ⟨proof⟩

lemma srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction:
  assumes
    ‹∀i ∈ I. hml_srbb_conjunct_models p (ψs i)›
    ‹hml_srbb_inner_models p (Obs α φ)›
    ‹(i∈I ∧ hml_srbb_conj.distinguishes (ψs i) p q)
         ∨ (hml_srbb_inner.distinguishes (Obs α φ) p q)›
  shows
    ‹hml_srbb_inner.distinguishes (BranchConj α φ I ψs) p q›
  ⟨proof⟩
```

2.3 HML_{SRBB} Implication and Equivalence

```
abbreviation hml_srbb_impl
  :: ‹('a, 's) hml_srbb ⇒ ('a, 's) hml_srbb ⇒ bool›  (infixr ‹⇛› 70)
where
  ‹hml_srbb_impl ≡ entails›

abbreviation
  hml_srbb_impl_inner
  :: ‹('a, 's) hml_srbb_inner ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  (infix ‹χ⇛› 70)
where
  ‹(χ⇛) ≡ hml_srbb_inner.entails›

abbreviation
  hml_srbb_impl_conjunct
  :: ‹('a, 's) hml_srbb_conjunct ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool›
  (infix ‹ψ⇛› 70)
where
  ‹(ψ⇛) ≡ hml_srbb_conj.entails›

abbreviation
  hml_srbb_eq
  :: ‹('a, 's) hml_srbb ⇒ ('a, 's) hml_srbb ⇒ bool›
  (infix ‹⇚srbb⇛› 70)
where
  ‹(⇚srbb⇛) ≡ logical_eq›

abbreviation
  hml_srbb_eq_inner
  :: ‹('a, 's) hml_srbb_inner ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  (infix ‹⇚χ⇛› 70)
where
  ‹(⇚χ⇛) ≡ hml_srbb_inner.logical_eq›

abbreviation
  hml_srbb_eq_conjunct
```

```
  :: ‹('a, 's) hml_srbb_conjunct ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool›
  (infix ‹⇚ψ⇛› 70)
  where
  ‹(⇚ψ⇛) ≡ hml_srbb_conj.logical_eq›
```

2.4 Substitution and Congruence

```
lemma srbb_internal_subst:
  assumes
    ‹χl ⇚χ⇛ χr›
    ‹φ ⇚srbb⇛ (Internal χl)›
  shows
    ‹φ ⇚srbb⇛ (Internal χr)›
  ⟨proof⟩

lemma internal_srbb_cong:
  assumes ‹χl ⇚χ⇛ χr›
  shows ‹(Internal χl) ⇚srbb⇛ (Internal χr)›
  ⟨proof⟩


lemma immconj_cong:
  assumes
    ‹ψsl ‘ I = ψsr ‘ I›
    ‹ψsl s ⇚ψ⇛ ψsr s›
  shows
    ‹ImmConj (I ∪ {s}) ψsl ⇚srbb⇛ ImmConj (I ∪ {s}) ψsr›
  ⟨proof⟩

lemma obs_srbb_cong:
  assumes ‹φl ⇚srbb⇛ φr›
  shows ‹(Obs α φl) ⇚χ⇛ (Obs α φr)›
  ⟨proof⟩
```

2.5 Trivial and Equivalent Formulas

```
lemma empty_conj_trivial[simp]:
  ‹state ⊨SRBB ImmConj {} ψs›
  ‹hml_srbb_inner_models state (Conj {} ψs)›
  ‹hml_srbb_inner_models state (Obs τ TT)›
  ⟨proof⟩

lemma empty_branch_conj_tau:
  ‹hml_srbb_inner_models state (BranchConj τ TT {} ψs)›
  ⟨proof⟩

lemma stable_conj_parts:
  assumes
    ‹hml_srbb_inner_models p (StableConj I Ψ)›
    ‹i ∈ I›
  shows
    ‹hml_srbb_conjunct_models p (Ψ i)›
  ⟨proof⟩

lemma branching_conj_parts:
  assumes
    ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
    ‹i ∈ I›
  shows
    ‹hml_srbb_conjunct_models p (Ψ i)›
  ⟨proof⟩
```

```
lemma branching_conj_obs:
  assumes ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
  shows ‹hml_srbb_inner_models p (Obs α φ)›
  ⟨proof⟩

lemma srbb_obs_τ_is_χTT: ‹Obs τ TT ⇚χ⇛ Conj {} ψs›
  ⟨proof⟩

lemma srbb_obs_is_empty_branch_conj: ‹Obs α φ ⇚χ⇛ BranchConj α φ {} ψs›
  ⟨proof⟩

lemma srbb_TT_is_χTT: ‹TT ⇚srbb⇛ Internal (Conj {} ψs)›
  ⟨proof⟩

lemma srbb_TT_is_empty_conj: ‹TT ⇚srbb⇛ ImmConj {} ψs›
  ⟨proof⟩
```

Positive conjuncts in stable conjunctions can be replaced by negative ones.

```
lemma srbb_stable_Neg_normalizable:
  assumes
    ‹i ∈ I› ‹Ψ i = Pos χ›
    ‹Ψ' = Ψ(i:= Neg (StableConj {left} (λ_. Neg χ)))›
  shows
    ‹Internal (StableConj I Ψ) ⇚srbb⇛ Internal (StableConj I Ψ')›
⟨proof⟩
```

All positive conjuncts in stable conjunctions can be replaced by negative ones at once.

```
lemma srbb_stable_Neg_normalizable_set:
  assumes
    ‹Ψ' = (λi. case (Ψ i) of
      Pos χ ⇒ Neg (StableConj {left} (λ_. Neg χ)) |
      Neg χ ⇒ Neg χ)›
  shows
    ‹Internal (StableConj I Ψ) ⇚srbb⇛ Internal (StableConj I Ψ')›
⟨proof⟩

definition conjunctify_distinctions ::
  ‹('s ⇒ ('a, 's) hml_srbb) ⇒ 's ⇒ ('s ⇒ ('a, 's) hml_srbb_conjunct)› where
  ‹conjunctify_distinctions Φ p ≡ λq.
    case (Φ q) of
      TT ⇒ undefined
    | Internal χ ⇒ Pos χ
    | ImmConj I Ψ ⇒ Ψ (SOME i. i∈I ∧ hml_srbb_conj.distinguishes (Ψ i) p q)›

lemma distinction_conjunctification:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes ((conjunctify_distinctions Φ p) q) p q›
  ⟨proof⟩

lemma distinction_combination:
  fixes p q
  defines
    ‹Qα ≡ {q'. q ↠ q' ∧ (∄φ. distinguishes φ p q')}›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q''. q' ↦a α q'' ⟶ (distinguishes (Φ q'') p' q'')›
```

```
  shows
    ‹∀q'∈Qα.
      hml_srbb_inner.distinguishes (Obs α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
                                                   (conjunctify_distinctions Φ p'))) p q'›
⟨proof⟩

definition conjunctify_distinctions_dual ::
  ‹('s ⇒ ('a, 's) hml_srbb) ⇒ 's ⇒ ('s ⇒ ('a, 's) hml_srbb_conjunct)› where
  ‹conjunctify_distinctions_dual Φ p ≡ λq.
    case (Φ q) of
      TT ⇒ undefined
    | Internal χ ⇒ Neg χ
    | ImmConj I Ψ ⇒
      (case Ψ (SOME i. i∈I ∧ hml_srbb_conj.distinguishes (Ψ i) q p) of
        Pos χ ⇒ Neg χ | Neg χ ⇒ Pos χ)›

lemma dual_conjunct:
  assumes
    ‹hml_srbb_conj.distinguishes ψ p q›
  shows
    ‹hml_srbb_conj.distinguishes (case ψ of
              hml_srbb_conjunct.Pos χ ⇒ hml_srbb_conjunct.Neg χ
              | hml_srbb_conjunct.Neg χ ⇒ hml_srbb_conjunct.Pos χ) q p›
  ⟨proof⟩

lemma distinction_conjunctification_dual:
  assumes
    ‹∀q∈I. distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes (conjunctify_distinctions_dual Φ p q) p q›
  ⟨proof⟩

lemma distinction_conjunctification_two_way:
  fixes Φ p I
  defines
    ‹conjfy q ≡
      (if distinguishes (Φ q) p q
      then conjunctify_distinctions Φ
      else conjunctify_distinctions_dual Φ) p q›
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes (conjfy q) p q›
⟨proof⟩

end — of lts_tau

end
```

3 Expressiveness Prices

```
theory Energy
  imports "HOL-Library.Extended_Nat"
begin
```

We intend to work on eight-dimensional vectors in an energy game. The dimensions will encode expressiveness prices to $\mathrm{HML}_{\mathrm{SRBB}}$ formulas. This price is supposed to capture syntactic features needed to describe a certain property and will later be used to select sublogics of specific expressiveness to characterize behavioural equivalences.

The eight dimensions are intended to measure the following properties of formulas:

1. Modal depth (of observations $\langle\alpha\rangle$, (α)),
2. Depth of branching conjunctions (with one observation clause not starting with $\langle\varepsilon\rangle$),
3. Depth of stable conjunctions (that do enforce stability by a $\neg\langle\tau\rangle\top$-conjunct),
4. Depth of unstable conjunctions (that do not enforce stability by a $\neg\langle\tau\rangle\top$-conjunct),
5. Depth of immediate conjunctions (that are not preceded by $\langle\varepsilon\rangle$),
6. Maximal modal depth of positive clauses in conjunctions,
7. Maximal modal depth of negative clauses in conjunctions,
8. Depth of negations

```
datatype energy =
  E (modal_depth: ‹enat›) (br_conj_depth: ‹enat›) (conj_depth: ‹enat›)
    (st_conj_depth: ‹enat›) (imm_conj_depth: ‹enat›)
    (pos_conjuncts: ‹enat›) (neg_conjuncts: ‹enat›) (neg_depth: ‹enat›)
```

3.1 Comparing and Subtracting Energies

In order to define subtraction on energies, we first lift the orderings $\leq$ and `<` from `enat` to `energy`.

```
instantiation energy :: order begin

definition ‹e1 ≤ e2 ≡
  (case e1 of E a1 b1 c1 d1 e1 f1 g1 h1 ⇒ (
    case e2 of E a2 b2 c2 d2 e2 f2 g2 h2 ⇒
      (a1 ≤ a2 ∧ b1 ≤ b2 ∧ c1 ≤ c2 ∧ d1 ≤ d2 ∧ e1 ≤ e2 ∧ f1 ≤ f2 ∧ g1 ≤ g2 ∧ h1 ≤ h2)
    ))›

definition ‹(x::energy) < y = (x ≤ y ∧ ¬ y ≤ x)›

instance ⟨proof⟩

lemma leq_components[simp]:
  shows ‹e1 ≤ e2 ≡
      (modal_depth e1 ≤ modal_depth e2 ∧ br_conj_depth e1 ≤ br_conj_depth e2
      ∧ conj_depth e1 ≤ conj_depth e2 ∧ st_conj_depth e1 ≤ st_conj_depth e2
      ∧ imm_conj_depth e1 ≤ imm_conj_depth e2 ∧ pos_conjuncts e1 ≤ pos_conjuncts e2
      ∧ neg_conjuncts e1 ≤ neg_conjuncts e2 ∧ neg_depth e1 ≤ neg_depth e2)›
  ⟨proof⟩

lemma energy_leq_cases:
  assumes
    ‹modal_depth e1 ≤ modal_depth e2› ‹br_conj_depth e1 ≤ br_conj_depth e2›
    ‹conj_depth e1 ≤ conj_depth e2› ‹st_conj_depth e1 ≤ st_conj_depth e2›
```

```
    ‹imm_conj_depth e1 ≤ imm_conj_depth e2› ‹pos_conjuncts e1 ≤ pos_conjuncts e2›
    ‹neg_conjuncts e1 ≤ neg_conjuncts e2› ‹neg_depth e1 ≤ neg_depth e2›
  shows
    ‹e1 ≤ e2›
  ⟨proof⟩

end

abbreviation somewhere_larger where ‹somewhere_larger e1 e2 ≡ ¬(e1 ≥ e2)›

lemma somewhere_larger_eq:
  assumes
    ‹somewhere_larger e1 e2›
  shows
    ‹modal_depth e1 < modal_depth e2 ∨ br_conj_depth e1 < br_conj_depth e2
    ∨ conj_depth e1 < conj_depth e2 ∨ st_conj_depth e1 < st_conj_depth e2
    ∨ imm_conj_depth e1 < imm_conj_depth e2 ∨ pos_conjuncts e1 < pos_conjuncts e2
    ∨ neg_conjuncts e1 < neg_conjuncts e2 ∨ neg_depth e1 < neg_depth e2›
  ⟨proof⟩

instantiation energy :: minus
begin

definition minus_energy_def[simp]: ‹e1 - e2 ≡ E
  ((modal_depth e1) - (modal_depth e2))
  ((br_conj_depth e1) - (br_conj_depth e2))
  ((conj_depth e1) - (conj_depth e2))
  ((st_conj_depth e1) - (st_conj_depth e2))
  ((imm_conj_depth e1) - (imm_conj_depth e2))
  ((pos_conjuncts e1) - (pos_conjuncts e2))
  ((neg_conjuncts e1) - (neg_conjuncts e2))
  ((neg_depth e1) - (neg_depth e2))›

instance ⟨proof⟩

end
```

Some lemmas to ease the manipulation of expressions using subtraction on energies.

```
lemma energy_minus[simp]:
  shows ‹E a1 b1 c1 d1 e1 f1 g1 h1 - E a2 b2 c2 d2 e2 f2 g2 h2
         = E (a1 - a2) (b1 - b2) (c1 - c2) (d1 - d2)
             (e1 - e2) (f1 - f2) (g1 - g2) (h1 - h2)›
  ⟨proof⟩

lemma minus_component_leq:
  assumes
    ‹s ≤ x›
    ‹x ≤ y›
  shows
    ‹modal_depth (x - s) ≤ modal_depth (y - s)›
    ‹br_conj_depth (x - s) ≤ br_conj_depth (y - s)›
    ‹conj_depth (x - s) ≤ conj_depth (y - s)›
    ‹st_conj_depth (x - s) ≤ st_conj_depth (y - s)›
    ‹imm_conj_depth (x - s) ≤ imm_conj_depth (y - s)›
    ‹pos_conjuncts (x - s) ≤ pos_conjuncts (y - s)›
    ‹neg_conjuncts (x - s) ≤ neg_conjuncts (y - s)›
    ‹neg_depth (x - s) ≤ neg_depth (y - s)›
  ⟨proof⟩

lemma enat_diff_mono:
  assumes ‹(i::enat) ≤ j›
```

```
  shows ‹i - k ≤ j - k›
⟨proof⟩
```

We further show that the subtraction of energies is decreasing.

```
lemma energy_diff_mono:
  fixes s :: energy
  shows ‹mono_on UNIV (λx. x - s)›
  ⟨proof⟩

lemma gets_smaller:
  fixes s :: energy
  shows ‹(λx. x - s) x ≤ x›
  ⟨proof⟩

lemma mono_subtract:
  assumes ‹x ≤ x'›
  shows ‹(λx. x - (E a b c d e f g h)) x ≤ (λx. x - (E a b c d e f g h)) x'›
  ⟨proof⟩
```

Abbreviations for performing subtraction in the energy games.

```
abbreviation ‹subtract_fn a b c d e f g h ≡
  (λx. if somewhere_larger x (E a b c d e f g h) then None else Some (x - (E a b c d e f g h)))›

abbreviation ‹subtract a b c d e f g h ≡ Some (subtract_fn a b c d e f g h)›
```

3.2 Minimum Updates

Two energy updates that replace the first component with the minimum of two other components.

```
definition ‹min1_6 e ≡ case e of E a b c d e f g h ⇒ Some (E (min a f) b c d e f g h)›
definition ‹min1_7 e ≡ case e of E a b c d e f g h ⇒ Some (E (min a g) b c d e f g h)›
```

Abbreviations for identity update.

```
abbreviation ‹id_up ≡ Some Some›
```

lift order to options

```
instantiation option :: (order) order
begin

definition less_eq_option_def[simp]:
  ‹less_eq_option (optA :: 'a option) optB ≡
    case optA of
      (Some a) ⇒
        (case optB of
          (Some b) ⇒ a ≤ b |
          None ⇒ False) |
      None ⇒ True›

definition less_option_def[simp]:
  ‹less_option (optA :: 'a option) optB ≡ (optA ≤ optB ∧ ¬ optB ≤ optA)›

instance ⟨proof⟩

end
```

Again, we prove some lemmas to ease the manipulation of expressions using mininum updates.

```
lemma min_1_6_simps[simp]:
  shows ‹modal_depth (the (min1_6 e)) = min (modal_depth e) (pos_conjuncts e)›
        ‹br_conj_depth (the (min1_6 e)) = br_conj_depth e›
```

```
        ‹conj_depth (the (min1_6 e)) = conj_depth e›
        ‹st_conj_depth (the (min1_6 e)) = st_conj_depth e›
        ‹imm_conj_depth (the (min1_6 e)) = imm_conj_depth e›
        ‹pos_conjuncts (the (min1_6 e)) = pos_conjuncts e›
        ‹neg_conjuncts (the (min1_6 e)) = neg_conjuncts e›
        ‹neg_depth (the (min1_6 e)) = neg_depth e›
  ⟨proof⟩

lemma min_1_7_simps[simp]:
  shows ‹modal_depth (the (min1_7 e)) = min (modal_depth e) (neg_conjuncts e)›
        ‹br_conj_depth (the (min1_7 e)) = br_conj_depth e›
        ‹conj_depth (the (min1_7 e)) = conj_depth e›
        ‹st_conj_depth (the (min1_7 e)) = st_conj_depth e›
        ‹imm_conj_depth (the (min1_7 e)) = imm_conj_depth e›
        ‹pos_conjuncts (the (min1_7 e)) = pos_conjuncts e›
        ‹neg_conjuncts (the (min1_7 e)) = neg_conjuncts e›
        ‹neg_depth (the (min1_7 e)) = neg_depth e›
  ⟨proof⟩

lemma min_1_6_some:
  shows ‹min1_6 e ≠ None›
  ⟨proof⟩

lemma min_1_7_some:
  shows ‹min1_7 e ≠ None›
  ⟨proof⟩

lemma min_1_7_lower_end:
  assumes ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) = None›
  shows ‹neg_depth e = 0›
  ⟨proof⟩

lemma min_1_7_subtr_simp:
  shows ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
    = (if neg_depth e = 0 then None
       else Some (E (min (modal_depth e) (neg_conjuncts e)) (br_conj_depth e) (conj_depth e)
                    (st_conj_depth e) (imm_conj_depth e) (pos_conjuncts e)
                    (neg_conjuncts e) (neg_depth e - 1)))›
  ⟨proof⟩

lemma min_1_7_subtr_mono:
  shows ‹mono (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
⟨proof⟩

lemma min_1_6_subtr_simp:
  shows ‹(Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)
    = (if br_conj_depth e = 0 ∨ conj_depth e = 0 then None
       else Some (E (min (modal_depth e) (pos_conjuncts e)) (br_conj_depth e - 1)
                    (conj_depth e - 1) (st_conj_depth e) (imm_conj_depth e)
                    (pos_conjuncts e) (neg_conjuncts e) (neg_depth e)))›
  ⟨proof⟩

instantiation energy :: Sup
begin

definition ‹Sup ee ≡ E
  (Sup (modal_depth ‘ ee)) (Sup (br_conj_depth ‘ ee )) (Sup (conj_depth ‘ ee))
  (Sup (st_conj_depth ‘ ee)) (Sup (imm_conj_depth ‘ ee)) (Sup (pos_conjuncts ‘ ee))
  (Sup (neg_conjuncts ‘ ee)) (Sup (neg_depth ‘ ee))›

instance ⟨proof⟩
```

```
end
```

```
end
```

3.3 Components of Expressiveness Prices

```
theory Expressiveness_Price
  imports HML_SRBB Energy
begin
```

The (maximal) modal depth (of observations $\langle\alpha\rangle$, (α)) is increased on each `Obs` and `BranchConj`.

```
primrec
      modal_depth_srbb :: ‹('act, 'i) hml_srbb ⇒ enat›
  and modal_depth_srbb_inner :: ‹('act, 'i) hml_srbb_inner ⇒ enat›
  and modal_depth_srbb_conjunct :: ‹('act, 'i) hml_srbb_conjunct ⇒ enat› where
 ‹modal_depth_srbb TT = 0› |
 ‹modal_depth_srbb (Internal χ) = modal_depth_srbb_inner χ› |
 ‹modal_depth_srbb (ImmConj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |

 ‹modal_depth_srbb_inner (Obs α φ) = 1 + modal_depth_srbb φ› |
 ‹modal_depth_srbb_inner (Conj I ψs) =
    Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |
 ‹modal_depth_srbb_inner (StableConj I ψs) =
    Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |
 ‹modal_depth_srbb_inner (BranchConj a φ I ψs) =
    Sup ({1 + modal_depth_srbb φ} ∪ ((modal_depth_srbb_conjunct ∘ ψs) ‘ I))› |

 ‹modal_depth_srbb_conjunct (Pos χ) = modal_depth_srbb_inner χ› |
 ‹modal_depth_srbb_conjunct (Neg χ) = modal_depth_srbb_inner χ›
```

The depth of branching conjunctions (with one observation clause not starting with $\langle\varepsilon\rangle$) is increased on each: `BranchConj`.

```
primrec
      branching_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and branch_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and branch_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹branching_conjunction_depth TT = 0› |
  ‹branching_conjunction_depth (Internal χ) = branch_conj_depth_inner χ› |
  ‹branching_conjunction_depth (ImmConj I ψs) =
    Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I)› |

  ‹branch_conj_depth_inner (Obs _ φ) = branching_conjunction_depth φ› |
  ‹branch_conj_depth_inner (Conj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹branch_conj_depth_inner (StableConj I ψs) =
    Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹branch_conj_depth_inner (BranchConj _ φ I ψs) =
     1 + Sup ({branching_conjunction_depth φ} ∪ ((branch_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹branch_conj_depth_conjunct (Pos χ) = branch_conj_depth_inner χ› |
  ‹branch_conj_depth_conjunct (Neg χ) = branch_conj_depth_inner χ›
```

The depth of stable conjunctions (that do enforce stability by a $\neg\langle\tau\rangle\top$-conjunct) is increased on each `StableConj`. Note that if the `StableConj` is empty (has no other conjuncts), it is still counted.

```
primrec
      stable_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and st_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and st_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹stable_conjunction_depth TT = 0› |
  ‹stable_conjunction_depth (Internal χ) = st_conj_depth_inner χ› |
```

```
  ‹stable_conjunction_depth (ImmConj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)› |

  ‹st_conj_depth_inner (Obs _ φ) = stable_conjunction_depth φ› |
  ‹st_conj_depth_inner (Conj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹st_conj_depth_inner (StableConj I ψs) = 1 + Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹st_conj_depth_inner (BranchConj _ φ I ψs) =
    Sup ({stable_conjunction_depth φ} ∪ ((st_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹st_conj_depth_conjunct (Pos χ) = st_conj_depth_inner χ› |
  ‹st_conj_depth_conjunct (Neg χ) = st_conj_depth_inner χ›
```

The depth of unstable conjunctions (that do not enforce stability by a $\neg\langle\tau\rangle\top$-conjunct) is increased on each:

- `ImmConj` if there are conjuncts (i.e. $\bigwedge\{\}$ is not counted)
- `Conj` if there are conjuncts, (i.e. the conjunction is not empty)
- `BranchConj`.

```
primrec
      unstable_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and inst_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and inst_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹unstable_conjunction_depth TT = 0› |
  ‹unstable_conjunction_depth (Internal χ) = inst_conj_depth_inner χ› |
  ‹unstable_conjunction_depth (ImmConj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹inst_conj_depth_inner (Obs _ φ) = unstable_conjunction_depth φ› |
  ‹inst_conj_depth_inner (Conj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I))› |
  ‹inst_conj_depth_inner (StableConj I ψs) =
    Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹inst_conj_depth_inner (BranchConj _ φ I ψs) =
    1 + Sup ({unstable_conjunction_depth φ} ∪ ((inst_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹inst_conj_depth_conjunct (Pos χ) = inst_conj_depth_inner χ› |
  ‹inst_conj_depth_conjunct (Neg χ) = inst_conj_depth_inner χ›
```

The depth of immediate conjunctions (that are not preceded by $\langle\varepsilon\rangle$) is increased on each `ImmConj` if there are conjuncts (i.e. $\bigwedge\{\}$ is not counted).

```
primrec
      immediate_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and imm_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and imm_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹immediate_conjunction_depth TT = 0› |
  ‹immediate_conjunction_depth (Internal χ) = imm_conj_depth_inner χ› |
  ‹immediate_conjunction_depth (ImmConj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹imm_conj_depth_inner (Obs _ φ) = immediate_conjunction_depth φ› |
  ‹imm_conj_depth_inner (Conj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹imm_conj_depth_inner (StableConj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I)› |
```

```
  ‹imm_conj_depth_inner (BranchConj _ φ I ψs) =
    Sup ({immediate_conjunction_depth φ} ∪ ((imm_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹imm_conj_depth_conjunct (Pos χ) = imm_conj_depth_inner χ› |
  ‹imm_conj_depth_conjunct (Neg χ) = imm_conj_depth_inner χ›
```

The maximal modal depth of positive clauses in conjunctions calculates the modal depth for every positive clause in a conjunction (`Pos` χ).

```
primrec
      max_positive_conjunct_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and max_pos_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and max_pos_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹max_positive_conjunct_depth TT = 0› |
  ‹max_positive_conjunct_depth (Internal χ) = max_pos_conj_depth_inner χ› |
  ‹max_positive_conjunct_depth (ImmConj I ψs) =
    Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)› |

  ‹max_pos_conj_depth_inner (Obs _ φ) = max_positive_conjunct_depth φ› |
  ‹max_pos_conj_depth_inner (Conj I ψs) =
    Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹max_pos_conj_depth_inner (StableConj I ψs) =
    Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹max_pos_conj_depth_inner (BranchConj _ φ I ψs) =
    Sup ({1 + modal_depth_srbb φ, max_positive_conjunct_depth φ}
        ∪ ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹max_pos_conj_depth_conjunct (Pos χ) = modal_depth_srbb_inner χ› |
  ‹max_pos_conj_depth_conjunct (Neg χ) = max_pos_conj_depth_inner χ›
```

The maximal modal depth of negative clauses in conjunctions calculates the modal depth for every negative clause in a conjunction (`Neg` χ).

```
primrec
      max_negative_conjunct_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and max_neg_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and max_neg_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹max_negative_conjunct_depth TT = 0› |
  ‹max_negative_conjunct_depth (Internal χ) = max_neg_conj_depth_inner χ› |
  ‹max_negative_conjunct_depth (ImmConj I ψs) =
    Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)› |

  ‹max_neg_conj_depth_inner (Obs _ φ) = max_negative_conjunct_depth φ› |
  ‹max_neg_conj_depth_inner (Conj I ψs) =
    Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹max_neg_conj_depth_inner (StableConj I ψs) =
    Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)› |
  ‹max_neg_conj_depth_inner (BranchConj _ φ I ψs) =
    Sup ({max_negative_conjunct_depth φ} ∪ ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹max_neg_conj_depth_conjunct (Pos χ) = max_neg_conj_depth_inner χ› |
  ‹max_neg_conj_depth_conjunct (Neg χ) = modal_depth_srbb_inner χ›
```

The depth of negations on a path of the syntax tree) is increased on each `Neg` χ.

```
primrec
      negation_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and neg_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and neg_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹negation_depth TT = 0› |
  ‹negation_depth (Internal χ) = neg_depth_inner χ› |
  ‹negation_depth (ImmConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)› |
```

```
  ‹neg_depth_inner (Obs _ φ) = negation_depth φ› |
  ‹neg_depth_inner (Conj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)› |
  ‹neg_depth_inner (StableConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)› |
  ‹neg_depth_inner (BranchConj _ φ I ψs) =
    Sup ({negation_depth φ} ∪ ((neg_depth_conjunct ∘ ψs) ‘ I))› |

  ‹neg_depth_conjunct (Pos χ) = neg_depth_inner χ› |
  ‹neg_depth_conjunct (Neg χ) = 1 + neg_depth_inner χ›
```

3.4 Properties of Price Components

```
lemma ‹modal_depth_srbb TT = 0›
  ⟨proof⟩

lemma ‹modal_depth_srbb (Internal (Obs α (Internal (BranchConj β TT {} ψs2)))) = 2›
  ⟨proof⟩

fun observe_n_alphas :: ‹'a ⇒ nat ⇒ ('a, nat) hml_srbb› where
  ‹observe_n_alphas α 0 = TT› |
  ‹observe_n_alphas α (Suc n) = Internal (Obs α (observe_n_alphas α n))›

lemma obs_n_α_depth_n: ‹modal_depth_srbb (observe_n_alphas α n) = n›
⟨proof⟩

lemma sup_nats_in_enats_infinite: ‹(SUP x∈ℕ. enat x) = ∞›
  ⟨proof⟩

lemma sucs_of_nats_in_enats_sup_infinite: ‹(SUP x∈ℕ. 1 + enat x) = ∞›
  ⟨proof⟩

lemma ‹modal_depth_srbb (ImmConj ℕ (λn. Pos (Obs α (observe_n_alphas α n)))) = ∞›
  ⟨proof⟩

lemma modal_depth_dominates_pos_conjuncts:
  fixes
    φ::‹('a, 's) hml_srbb› and
    χ::‹('a, 's) hml_srbb_inner› and
    ψ::‹('a, 's) hml_srbb_conjunct›
  shows
    ‹(max_positive_conjunct_depth φ ≤ modal_depth_srbb φ)
    ∧ (max_pos_conj_depth_inner χ ≤ modal_depth_srbb_inner χ)
    ∧ (max_pos_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ)›
  ⟨proof⟩

lemma modal_depth_dominates_neg_conjuncts:
  fixes
    φ::‹('a, 's) hml_srbb› and
    χ::‹('a, 's) hml_srbb_inner› and
    ψ::‹('a, 's) hml_srbb_conjunct›
  shows
    ‹(max_negative_conjunct_depth φ ≤ modal_depth_srbb φ)
    ∧ (max_neg_conj_depth_inner χ ≤ modal_depth_srbb_inner χ)
    ∧ (max_neg_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ)›
  ⟨proof⟩
```

3.5 Expressiveness Price Function

The `expressiveness_price` function combines the eight component functions into one.

```
fun expressiveness_price :: ‹('a, 's) hml_srbb ⇒ energy› where
  ‹expressiveness_price φ =
```

```
    E (modal_depth_srbb            φ)
      (branching_conjunction_depth φ)
      (unstable_conjunction_depth  φ)
      (stable_conjunction_depth    φ)
      (immediate_conjunction_depth φ)
      (max_positive_conjunct_depth φ)
      (max_negative_conjunct_depth φ)
      (negation_depth              φ)›
```

Here, we can see the decomposed price of an immediate conjunction:

```
lemma expressiveness_price_ImmConj_def:
  shows ‹expressiveness_price (ImmConj I ψs) = E
    (Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I))
    (Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I))
    (if I = {} then 0 else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I))
    (if I = {} then 0 else 1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((neg_depth_conjunct ∘ ψs) ‘ I))› ⟨proof⟩

lemma expressiveness_price_ImmConj_non_empty_def:
  assumes ‹I ≠ {}›
  shows ‹expressiveness_price (ImmConj I ψs) = E
    (Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I))
    (Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I))
    (1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I))
    (1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I))
    (Sup ((neg_depth_conjunct ∘ ψs) ‘ I))› ⟨proof⟩

lemma expressiveness_price_ImmConj_empty_def:
  assumes ‹I = {}›
  shows ‹expressiveness_price (ImmConj I ψs) = E 0 0 0 0 0 0 0 0› ⟨proof⟩
```

Formalizing HML_{SRBB} by mutually recursive data types leads to expressiveness price functions of these other types and corresponding definitions and lemmas.

```
fun expr_pr_inner :: ‹('a, 's) hml_srbb_inner ⇒ energy› where
  ‹expr_pr_inner χ =
    E (modal_depth_srbb_inner χ)
      (branch_conj_depth_inner χ)
      (inst_conj_depth_inner χ)
      (st_conj_depth_inner χ)
      (imm_conj_depth_inner χ)
      (max_pos_conj_depth_inner χ)
      (max_neg_conj_depth_inner χ)
      (neg_depth_inner χ)›

fun expr_pr_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ energy› where
  ‹expr_pr_conjunct ψ =
    E (modal_depth_srbb_conjunct ψ)
      (branch_conj_depth_conjunct ψ)
      (inst_conj_depth_conjunct ψ)
      (st_conj_depth_conjunct ψ)
      (imm_conj_depth_conjunct ψ)
      (max_pos_conj_depth_conjunct ψ)
      (max_neg_conj_depth_conjunct ψ)
      (neg_depth_conjunct ψ)›
```

3.6 Prices of Certain Formulas

```
context lts_tau
begin
```

For example, here, we establish that the expressiveness price of `Internal` χ is equal to the expressiveness price of χ.

```
lemma expr_internal_eq:
  shows ‹expressiveness_price (Internal χ) = expr_pr_inner χ›
  ⟨proof⟩

lemma expr_pos:
  assumes ‹expr_pr_inner χ ≤ the (min1_6 e)›
  shows ‹expr_pr_conjunct (Pos χ) ≤ e›
  ⟨proof⟩

lemma expr_neg:
  assumes
    ‹expr_pr_inner χ ≤ e'›
    ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) = Some e'›
  shows ‹expr_pr_conjunct (Neg χ) ≤ e›
⟨proof⟩

lemma expr_obs:
  assumes
    ‹expressiveness_price φ ≤ e'›
    ‹subtract_fn 1 0 0 0 0 0 0 0 e = Some e'›
  shows ‹expr_pr_inner (Obs α φ) ≤ e›
  ⟨proof⟩

lemma expr_st_conj:
  assumes
    ‹subtract_fn  0 0 0 1 0 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹∀q ∈ I. expr_pr_conjunct (ψs q) ≤ e'›
  shows
    ‹expr_pr_inner (StableConj I ψs) ≤ e›
⟨proof⟩

lemma expr_imm_conj:
  assumes
    ‹subtract_fn  0 0 0 0 1 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹expr_pr_inner (Conj I ψs) ≤ e'›
  shows ‹expressiveness_price (ImmConj I ψs) ≤ e›
⟨proof⟩

lemma expr_conj:
  assumes
    ‹subtract_fn 0 0 1 0 0 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹∀q ∈ I. expr_pr_conjunct (ψs q) ≤ e'›
  shows ‹expr_pr_inner (Conj I ψs) ≤ e›
⟨proof⟩

lemma expr_br_conj:
  assumes
    ‹subtract_fn 0 1 1 0 0 0 0 0 e = Some e'›
    ‹min1_6 e' = Some e''›
    ‹subtract_fn 1 0 0 0 0 0 0 0 e'' = Some e'''›
    ‹expressiveness_price φ ≤ e'''›
```

```
    ‹∀q ∈ Q. expr_pr_conjunct (Φ q) ≤ e'›
    ‹1 + modal_depth_srbb φ ≤ pos_conjuncts e›
  shows ‹expr_pr_inner (BranchConj α φ Q Φ) ≤ e›
⟨proof⟩

lemma expressiveness_price_ImmConj_geq_parts:
  assumes ‹i ∈ I›
  shows ‹expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0 ≥ expr_pr_conjunct (ψs i)›
⟨proof⟩

lemma expressiveness_price_ImmConj_geq_parts':
  assumes ‹i ∈ I›
  shows
    ‹(expressiveness_price (ImmConj I ψs) - E 0 0 0 0 1 0 0 0) - E 0 0 1 0 0 0 0 0
    ≥ expr_pr_conjunct (ψs i)›
  ⟨proof⟩
```

Here, we show the prices for some specific formulas.

```
lemma example_φ_cp:
  fixes op a b::‹'a› and left right::‹'s›
  defines ‹φ ≡
    (Internal
      (Obs op
        (Internal
          (Conj {left, right}
                (λi. (if i = left
                      then (Pos (Obs a TT))
                      else if i = right
                           then (Pos (Obs b TT))
                           else undefined))))))›
  shows
    ‹modal_depth_srbb            φ = 2›
    ‹branching_conjunction_depth φ = 0›
    ‹unstable_conjunction_depth  φ = 1›
    ‹stable_conjunction_depth    φ = 0›
    ‹immediate_conjunction_depth φ = 0›
    ‹max_positive_conjunct_depth φ = 1›
    ‹max_negative_conjunct_depth φ = 0›
    ‹negation_depth              φ = 0›
  ⟨proof⟩

lemma ‹expressiveness_price (Internal
      (Obs op
        (Internal
          (Conj {left, right}
                (λi. (if i = left
                      then (Pos (Obs a TT))
                      else if i = right
                           then (Pos (Obs b TT))
                           else undefined)))))) = E 2 0 1 0 0 1 0 0›
  ⟨proof⟩

lemma ‹expressiveness_price TT = E 0 0 0 0 0 0 0 0›
  ⟨proof⟩

lemma ‹expressiveness_price (ImmConj {} ψs) = E 0 0 0 0 0 0 0 0›
  ⟨proof⟩

lemma ‹expressiveness_price (Internal (Conj {} ψs)) = E 0 0 0 0 0 0 0 0›
  ⟨proof⟩
```

```
lemma ‹expressiveness_price (Internal (BranchConj α TT {} ψs)) = E 1 1 1 0 0 1 0 0›
  ⟨proof⟩

lemma expr_obs_phi:
  ‹subtract_fn 1 0 0 0 0 0 0 0 (expr_pr_inner (Obs α φ)) = Some (expressiveness_price φ)›
  ⟨proof⟩

end — pause lts_tau context
```

3.7 Characterizing Equivalence by Energy Coordinates

We can now define a sublanguage of Hennessy–Milner Logic $\mathcal{O}$ by the set of formulas with prices below an energy coordinate.

```
definition 𝒪 :: ‹energy ⇒ (('a, 's) hml_srbb) set› where
  ‹𝒪 energy ≡ {φ . expressiveness_price φ ≤ energy}›

lemma 𝒪_sup: ‹UNIV = 𝒪 (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞)› ⟨proof⟩

lemma price_hierarchy_entails_modal_hierarchy:
  assumes ‹e1 ≤ e2›
  shows ‹𝒪 e1 ⊆ 𝒪 e2›
  ⟨proof⟩

definition 𝒪_inner :: ‹energy ⇒ (('a, 's) hml_srbb_inner) set› where
  ‹𝒪_inner energy ≡ {χ . expr_pr_inner χ ≤ energy}›

definition 𝒪_conjunct :: ‹energy ⇒ (('a, 's) hml_srbb_conjunct) set› where
  ‹𝒪_conjunct energy ≡ {χ . expr_pr_conjunct χ ≤ energy}›

context lts_tau
begin
```

A state `p` pre-orders another state `q` with respect to some energy `e` if and only if `p` HML pre-orders `q` with respect to the HML sublanguage $\mathcal{O}$ `e`.

```
definition expr_preord :: ‹'s ⇒ energy ⇒ 's ⇒ bool› (‹_ ⪯ _ _› 60) where
  ‹(p ⪯ e q) ≡ preordered (𝒪 e) p q›
```

Conversely, `p` and `q` are equivalent with respect to `e` if and only if they are equivalent with respect to that HML sublanguage $\mathcal{O}$ `e`.

```
definition expr_equiv :: ‹'s ⇒ energy ⇒ 's ⇒ bool› (‹_ ∼ _ _› 60) where
  ‹(p ∼ e q) ≡ equivalent (𝒪 e) p q›

lemma price_hierachy_preorder_dual:
  assumes
    ‹e1 ≤ e2›
    ‹p ⪯ e2 q›
  shows
    ‹p ⪯ e1 q›
  ⟨proof⟩
```

3.8 Relational Effects of Prices

Certain properties of prices influence the preorder/equivalence relations that are characterized by price coordinates. (This will be important for some behavioral equivalences that we will prove to be characterized by specific prices.)

```
lemma distinction_combination_eta:
  fixes p q
  defines
```

```
    ‹Qα ≡ {q'. q ↠ q' ∧  (∄φ. φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes φ p q')}›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ distinguishes (Φ q''') p' q'''›
  shows
    ‹∀q'∈ Qα. hml_srbb_inner.distinguishes (Obs α (Internal (Conj
      {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
      (conjunctify_distinctions Φ p')))) p q'›
⟨proof⟩

lemma distinction_conjunctification_two_way_price:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
    ‹∀q∈I. Φ q ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
  shows
    ‹∀q∈I.
      (if distinguishes (Φ q) p q
        then conjunctify_distinctions
        else conjunctify_distinctions_dual
      ) Φ p q ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
⟨proof⟩

lemma distinction_combination_eta_two_way:
  fixes p q p' Φ
  defines
    ‹Qα ≡ {q'. q ↠ q' ∧  (∄φ. φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)
          ∧ (distinguishes φ p q' ∨ distinguishes φ q' p))}› and
    ‹Ψα ≡ λq'''. (
        if distinguishes (Φ q''') p' q'''
        then conjunctify_distinctions
        else conjunctify_distinctions_dual
      ) Φ p' q'''›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q'''
        ⟶ distinguishes (Φ q''') p' q''' ∨ distinguishes (Φ q''') q''' p'›
  shows
    ‹∀q'∈ Qα. hml_srbb_inner.distinguishes (Obs α (Internal (Conj
      {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
      Ψα))) p q'›
⟨proof⟩

lemma distinction_conjunctification_price:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q›
    ‹∀q∈I. Φ q ∈ 𝒪 pr›
    ‹modal_depth pr ≤ pos_conjuncts pr›
  shows
    ‹∀q∈I. ((conjunctify_distinctions Φ p) q) ∈ 𝒪_conjunct pr›
⟨proof⟩

lemma modal_stability_respecting:
  ‹stability_respecting (preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)))›
  ⟨proof⟩

end

end
```

4 Weak Traces

```
theory Weak_Traces
  imports Main HML_SRBB Expressiveness_Price
begin
```

The point of this theory is to prove that the coordinate `E ∞ 0 0 0 0 0 0 0` precisely characterizes weak trace preorder and equivalence.

4.1 Weak Traces as Modal Constructs

```
inductive
  is_trace_formula :: ‹('act, 'i) hml_srbb ⇒ bool› and
  is_trace_formula_inner :: ‹('act, 'i) hml_srbb_inner ⇒ bool›
where
  ‹is_trace_formula TT› |
  ‹is_trace_formula (Internal χ)› if ‹is_trace_formula_inner χ› |
  ‹is_trace_formula (ImmConj I ψs)› if ‹I = {}› |

  ‹is_trace_formula_inner (Obs α φ)› if ‹is_trace_formula φ› |
  ‹is_trace_formula_inner (Conj I ψs)› if ‹I = {}›
```

We define a function that translates a (weak) trace `tr` to a formula φ such that a state `p` models φ, `p` $\models \varphi$ if and only if `tr` is a (weak) trace of `p`.

```
fun
  wtrace_to_srbb :: ‹'act list ⇒ ('act, 'i) hml_srbb› and
  wtrace_to_inner :: ‹'act list ⇒ ('act, 'i) hml_srbb_inner› and
  wtrace_to_conjunct :: ‹'act list ⇒ ('act, 'i) hml_srbb_conjunct›
where
  ‹wtrace_to_srbb [] = TT› |
  ‹wtrace_to_srbb tr = (Internal (wtrace_to_inner tr))› |

  ‹wtrace_to_inner [] = (Conj {} (λ_. undefined))› | — Should never happen
  ‹wtrace_to_inner (α # tr) = (Obs α (wtrace_to_srbb tr))› |

  ‹wtrace_to_conjunct tr = Pos (wtrace_to_inner tr)› — Should never happen

lemma trace_to_srbb_is_trace_formula:
  ‹is_trace_formula (wtrace_to_srbb trace)›
  ⟨proof⟩
```

4.2 Weak Trace Observations through Coordinates

The following three lemmas show that the modal-logical characterization of weak traces corresponds to the sublanguage of $\mathrm{HML}_{\mathrm{SRBB}}$, obtain by the energy coordinates `(∞, 0, 0, 0, 0, 0, 0, 0)`.

```
lemma trace_formula_to_expressiveness:
  fixes
    φ :: ‹('act, 'i) hml_srbb› and
    χ :: ‹('act, 'i) hml_srbb_inner›
  shows  ‹(is_trace_formula φ       ⟶ (φ ∈ 𝒪       (E ∞ 0 0 0 0 0 0 0)))
        ∧ (is_trace_formula_inner χ ⟶ (χ ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)))›
  ⟨proof⟩

lemma expressiveness_to_trace_formula:
  fixes
    φ :: ‹('act, 'i) hml_srbb› and
    χ :: ‹('act, 'i) hml_srbb_inner›
  shows ‹(φ ∈ 𝒪 (E ∞ 0 0 0 0 0 0 0) ⟶ is_trace_formula φ)
       ∧ (χ ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0) ⟶ is_trace_formula_inner χ)
```

```
      ∧ True›
```
⟨*proof*⟩

```
lemma modal_depth_only_is_trace_form:
  ‹(is_trace_formula φ) = (φ ∈ 𝒪 (E ∞ 0 0 0 0 0 0 0))›
```
⟨*proof*⟩

```
context lts_tau
begin
```

If a trace formula φ is satisfied by a state `p` then there exists a weak trace `tr` of `p` such that `wtrace_to_srbb tr` is equivalent to φ.

```
lemma trace_formula_implies_trace:
  fixes
    ψ ::‹('a, 's) hml_srbb_conjunct›
  shows
    ‹is_trace_formula φ ⟹ p ⊨SRBB φ
      ⟹ ∃tr ∈ weak_traces p. wtrace_to_srbb tr ⇐srbb⇒ φ›
    ‹is_trace_formula_inner χ ⟹ hml_srbb_inner_models q χ
      ⟹ ∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ χ›
    True
```
⟨*proof*⟩

```
lemma trace_equals_trace_to_formula:
  ‹t ∈ weak_traces p ⟷ p ⊨SRBB (wtrace_to_srbb t)›
```
⟨*proof*⟩

```
lemma expr_preorder_characterizes_relational_preorder_traces:
  ‹(p ≲WT q) = (p ≼ (E ∞ 0 0 0 0 0 0 0) q)›
```
⟨*proof*⟩

Two states `p` and `q` are weakly trace equivalent if and only if they they are equivalent with respect to the coordinate $(\infty, 0, 0, 0, 0, 0, 0, 0)$.

```
theorem weak_traces_coordinate: ‹(p ≃WT q) = (p ∼ (E ∞ 0 0 0 0 0 0 0) q)›
```
⟨*proof*⟩

```
end

end
```

5 η-Bisimilarity and η-Similarity

```
theory Eta_Bisimilarity
  imports Expressiveness_Price
begin
```

η-Bisimilarity and η-Similarity are comparably arcane notions of behavioral equivalence. We show that they are characterized by coordinates `E` ∞ ∞ ∞ `0 0` ∞ ∞ ∞ and `E` ∞ ∞ ∞ `0 0` ∞ `0 0` as an illustration of how to connect coordinates and relational characterizations of equivalences.

5.1 Definition and Properties of η-(Bi-)Similarity

```
context lts_tau
begin
```

We characterize η-bisimilarity through symmetric η-simulations.

```
definition eta_simulation :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹eta_simulation R ≡ ∀p α p' q. R p q ⟶ p ↦ α p' ⟶
    (α = τ ∧ R p' q)
    ∨ (∃q' q'' q'''. q ↠ q' ∧ q' ↦ α q'' ∧ q'' ↠ q''' ∧ R p q' ∧ R p' q''')›

definition eta_bisimulated :: ‹'s ⇒ 's ⇒ bool› (infix ‹~η› 40) where
  ‹p ~η q ≡ ∃R. eta_simulation R ∧ symp R ∧ R p q›

lemma eta_bisim_sim:
  shows ‹eta_simulation (~η)›
  ⟨proof⟩

lemma eta_bisim_sym:
  assumes ‹p ~η q›
  shows ‹q ~η p›
  ⟨proof⟩

lemma silence_retains_eta_sim:
assumes
  ‹eta_simulation R›
  ‹R p q›
  ‹p ↠ p'›
shows ‹∃q'. R p' q' ∧ q ↠ q'›
  ⟨proof⟩

lemma eta_bisimulated_silently_retained:
  assumes
    ‹p ~η q›
    ‹p ↠ p'›
  shows
    ‹∃q'. q ↠ q' ∧ p' ~η q'› ⟨proof⟩
```

5.2 Logical Characterization of η-Bisimilarity through Expressiveness Price

```
lemma logic_eta_bisim_invariant:
  assumes
    ‹p0 ~η q0›
    ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
⟨proof⟩

lemma modal_eta_sim_eq: ‹eta_simulation (equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)))›
```

⟨*proof*⟩

```
theorem eta_bisim_coordinate: ‹(p ~η q) = (p ∼ (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) q)›
```
⟨*proof*⟩

5.3 η-Similarity

```
lemma logic_eta_sim_invariant:
  assumes
    ‹∃R. eta_simulation R ∧  R p0 q0›
    ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
```
⟨*proof*⟩

```
lemma modal_eta_sim: ‹eta_simulation (preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)))›
```
⟨*proof*⟩

```
theorem eta_sim_coordinate:
    ‹(p ≼ (E ∞ ∞ ∞ 0 0 ∞ 0 0) q) = (∃R. eta_simulation R ∧ R p q)›
```
⟨*proof*⟩

```
end

end
```

6 Branching Bisimilarity

```
theory Branching_Bisimilarity
  imports Eta_Bisimilarity
begin
```

The whole of the modal logic of `hml_srbb` precisely characterizes stability-respecting branching bisimilarity.

6.1 Definitions of (Stability-Respecting) Branching Bisimilarity

```
context lts_tau
begin

definition branching_simulation :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹branching_simulation R ≡ ∀p α p' q. R p q ⟶ p ↦ α p' ⟶
    ((α = τ ∧ R p' q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ R p q' ∧ R p' q''))›

lemma branching_simulation_intro:
  assumes
    ‹⋀p α p' q. R p q ⟹ p ↦ α p' ⟹
      ((α = τ ∧ R p' q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ R p q' ∧ R p' q''))›
  shows
    ‹branching_simulation R›
  ⟨proof⟩

definition branching_simulated :: ‹'s ⇒ 's ⇒ bool› where
  ‹branching_simulated p q ≡ ∃R. branching_simulation R ∧ R p q›

definition branching_bisimulated :: ‹'s ⇒ 's ⇒ bool› where
  ‹branching_bisimulated p q ≡ ∃R. branching_simulation R ∧ symp R ∧ R p q›

definition sr_branching_bisimulated :: ‹'s ⇒ 's ⇒ bool› (infix ‹~SRBB› 40) where
  ‹p ~SRBB q ≡ ∃R. branching_simulation R ∧ symp R ∧ stability_respecting R ∧ R p q›
```

6.2 Properties of Branching Bisimulation Equivalences

```
lemma branching_bisimilarity_branching_sim: ‹branching_simulation (~SRBB)›
  ⟨proof⟩

lemma branching_sim_eta_sim:
  assumes ‹branching_simulation R›
  shows ‹eta_simulation R›
  ⟨proof⟩

lemma silence_retains_branching_sim:
assumes
  ‹branching_simulation R›
  ‹R p q›
  ‹p ↠ p'›
shows ‹∃q'. R p' q' ∧ q ↠ q'›
  ⟨proof⟩

lemma branching_bisimilarity_stability: ‹stability_respecting (~SRBB)›
  ⟨proof⟩

lemma sr_branching_bisimulation_silently_retained:
  assumes
    ‹p ~SRBB q›
    ‹p ↠ p'›
  shows
```

‹∃q'. q ↠ q' ∧ p' ~SRBB q'› ⟨*proof*⟩

lemma sr_branching_bisimulation_sim:
assumes
‹p ~SRBB q›
‹p ↠ p'› ‹p' ↦a α p''›
shows
‹∃q' q''. q ↠ q' ∧ q' ↦a α q'' ∧ p' ~SRBB q' ∧ p'' ~SRBB q''›
⟨*proof*⟩

lemma sr_branching_bisimulated_sym:
assumes
‹p ~SRBB q›
shows
‹q ~SRBB p›
⟨*proof*⟩

lemma sr_branching_bisimulated_symp:
shows ‹symp (~SRBB)›
⟨*proof*⟩

lemma sr_branching_bisimulated_reflp:
shows ‹reflp (~SRBB)›
⟨*proof*⟩

lemma establish_sr_branching_bisim:
assumes
‹∀α p'. p ↦ α p' ⟶
((α = τ ∧ p' ~SRBB q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ p ~SRBB q' ∧ p' ~SRBB q''))›
‹∀α q'. q ↦ α q' ⟶
((α = τ ∧ p ~SRBB q') ∨ (∃p' p''. p ↠ p' ∧ p' ↦ α p'' ∧ p' ~SRBB q ∧ p'' ~SRBB q'))›
‹stable_state p ⟶ (∃q'. q ↠ q' ∧ p ~SRBB q' ∧ stable_state q')›
‹stable_state q ⟶ (∃p'. p ↠ p' ∧ p' ~SRBB q ∧ stable_state p')›
shows ‹p ~SRBB q›
⟨*proof*⟩

lemma sr_branching_bisimulation_stuttering:
assumes
‹pp ≠ []›
‹∀i < length pp - 1. pp!i ↦ τ pp!(Suc i)›
‹hd pp ~SRBB last pp›
‹i < length pp›
shows
‹hd pp ~SRBB pp!i›
⟨*proof*⟩

lemma sr_branching_bisimulation_stabilizes:
assumes
‹sr_branching_bisimulated p q›
‹stable_state p›
shows
‹∃q'. q ↠ q' ∧ sr_branching_bisimulated p q' ∧ stable_state q'›
⟨*proof*⟩

lemma sr_branching_bisim_stronger:
assumes
‹sr_branching_bisimulated p q›
shows
‹branching_bisimulated p q›
⟨*proof*⟩

6.3 hml_srbb as Modal Characterization of Stability-Respecting Branching Bisimilarity

```
lemma modal_sym: ‹symp (preordered UNIV)›
```
⟨*proof*⟩

```
lemma modal_branching_sim: ‹branching_simulation (preordered UNIV)›
```
⟨*proof*⟩

```
lemma logic_sr_branching_bisim_invariant:
  assumes
    ‹sr_branching_bisimulated p0 q0›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
```
⟨*proof*⟩

```
lemma sr_branching_bisim_is_hmlsrbb: ‹sr_branching_bisimulated p q = preordered UNIV p q›
```
⟨*proof*⟩

```
lemma sr_branching_bisimulated_transitive:
  assumes
    ‹p ~SRBB q›
    ‹q ~SRBB r›
  shows
    ‹p ~SRBB r›
```
⟨*proof*⟩

```
lemma sr_branching_bisimulated_equivalence: ‹equivp (~SRBB)›
```
⟨*proof*⟩

```
lemma sr_branching_bisimulation_stuttering_all:
  assumes
    ‹pp ≠ []›
    ‹∀i < length pp - 1.  pp!i ↦ τ pp!(Suc i)›
    ‹hd pp ~SRBB last pp›
    ‹i ≤ j› ‹j < length pp›
  shows
    ‹pp!i ~SRBB pp!j›
```
⟨*proof*⟩

```
theorem sr_branching_bisim_coordinate: ‹(p ~SRBB q) = (p ≼ (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞) q)›
```
⟨*proof*⟩

```
end

end
```

7 Energy Games

```
theory Energy_Games
  imports Main
begin
```

Energy games are the foundation for the weak spectroscopy game. We introduce them through a recursive definition of attacker's winning budgets in energy reachability games.

7.1 Fundamentals

```
type_synonym 'energy update = ‹'energy ⇒ 'energy option›
```

An energy game is played by two players on a directed graph labeled by energy updates. These updates represent the costs of choosing a certain move, in our case, for the attacker.

```
locale energy_game =
fixes
  weight_opt :: ‹'gstate ⇒ 'gstate ⇒ 'energy update option› and
  defender :: ‹'gstate ⇒ bool› and
  ord:: ‹'energy ⇒ 'energy ⇒ bool›
assumes
  antisim: ‹⋀e e'. (ord e e') ⟹ (ord e' e) ⟹ e = e'› and
  monotonicity: ‹⋀g g' e e' eu eu'.
    weight_opt g g' ≠ None ⟹ the (weight_opt g g') e = Some eu
    ⟹ the (weight_opt g g') e' = Some eu' ⟹ ord e e' ⟹ ord eu eu'› and
  defender_win_min: ‹⋀g g' e e'. ord e e' ⟹ weight_opt g g' ≠ None
    ⟹ the (weight_opt g g') e' = None ⟹ the (weight_opt g g') e = None›
begin

abbreviation attacker :: ‹'gstate ⇒ bool› where
  ‹attacker p ≡ ¬ defender p›

abbreviation moves :: ‹'gstate ⇒ 'gstate ⇒ bool› (infix ‹↣› 70) where
  ‹g1 ↣ g2 ≡ weight_opt g1 g2 ≠ None›

abbreviation weighted_move
  :: ‹'gstate ⇒ 'energy update ⇒ 'gstate ⇒  bool› (‹_ ↣wgt _ _› [60,60,60] 70) where
  ‹weighted_move g1 u g2 ≡ g1 ↣ g2 ∧ (the (weight_opt g1 g2) = u)›

abbreviation ‹weight g1 g2 ≡ the (weight_opt g1 g2)›

abbreviation ‹updated g g' e ≡ the (weight g g' e)›
```

7.2 Winning Budgets

The attacker wins a game if and only if they manage to force the defender to get stuck before running out of energy.

```
inductive attacker_wins:: ‹'energy ⇒ 'gstate ⇒ bool› where
  Attack: ‹attacker_wins e g› if
    ‹attacker g› ‹g ↣ g'› ‹weight g g' e = Some e'› ‹attacker_wins e' g'› |
  Defense: ‹attacker_wins e g› if
    ‹defender g› ‹⋀g'. (g ↣ g') ⟶ (∃e'. weight g g' e = Some e' ∧ attacker_wins e' g')›

lemma attacker_wins_Ga_with_id_step:
  assumes ‹attacker_wins e g'› ‹g ↣wgt Some g'› ‹attacker g›
  shows ‹attacker_wins e g›
  ⟨proof⟩
```

If from a certain starting position `g` a game is won by the attacker with some energy `e` (i.e. `e` is in the winning budget of `g`), then the game is also won by the attacker with more energy.

```
lemma win_a_upwards_closure:
  assumes
    ‹attacker_wins e g›
    ‹ord e e'›
  shows
    ‹attacker_wins e' g›
```
⟨*proof*⟩

```
end
```
— context `energy_game`

```
end
```

8 Weak Spectroscopy Game

```
theory Spectroscopy_Game
  imports Energy_Games Energy Labeled_Transition_Systems
begin
```

The weak spectroscopy game is an energy game played over an LTS. The attacker's moves in the weak spectroscopy game depend on the transitions of the processes and the available energy. Intuitively, each move type corresponds to a production in the construction of distinguishing formulas; and each attacker position to a non-terminal in the underlying grammar.

8.1 Game Rules

```
datatype ('s, 'a) spectroscopy_position =
  Attacker_Immediate (attacker_state: <'s>) (defender_states: <'s set>) |
  Attacker_Delayed (attacker_state: <'s>) (defender_states: <'s set>) |
  Attacker_Conjunct (attacker_state: <'s>) (defender_state: <'s>) |
  Attacker_Branch (attacker_state: <'s>) (defender_states: <'s set>) |

  Defender_Conj (attacker_state: <'s>) (defender_states: <'s set>) |
  Defender_Stable_Conj (attacker_state: <'s>) (defender_states: <'s set>) |
  Defender_Branch (attacker_state: <'s>) (attack_action: <'a>)
                  (attacker_state_succ: <'s>) (defender_states: <'s set>)
                  (defender_branch_states: <'s set>)

context lts_tau
begin
```

The names of moves of the weak spectroscopy game indicate the respective HML constructs they correspond to.

```
fun spectroscopy_moves :: <('s, 'a) spectroscopy_position ⇒ ('s, 'a) spectroscopy_position
  ⇒ energy update option>
where
  delay:
    <spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p' Q')
     = (if p' = p ∧ Q →»S Q' then id_up else None)> |

  procrastination:
    <spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q')
      = (if (Q' = Q ∧ p ≠ p' ∧ p ↦ τ p') then id_up else None)> |

  observation:
    <spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
      = (if (∃a. p ↦a a p' ∧ Q ↦aS a Q') then (subtract 1 0 0 0 0 0 0 0)
         else None)> |

  f_or_early_conj:
    <spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p' Q')
     =(if (Q≠{} ∧ Q = Q' ∧ p = p') then (subtract 0 0 0 0 1 0 0 0)
       else None)> |

  late_inst_conj:
    <spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p' Q')
      = (if p = p' ∧ Q = Q' then id_up else None)> |

  conj_answer:
    <spectroscopy_moves (Defender_Conj p Q) (Attacker_Conjunct p' q)
      = (if p = p' ∧ q ∈ Q then (subtract 0 0 1 0 0 0 0 0) else None)> |

  pos_neg_clause:
```

```
    ‹spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed p' Q')
      = (if (p = p') then
          (if {q} →↠S Q' then Some min1_6 else None)
        else (if {p} →↠S Q' ∧ q=p'
              then Some (λe. Option.bind (subtract_fn 0 0 0 0 0 0 0 1 e) min1_7)
              else None))› |

  late_stbl_conj:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p' Q')
      = (if (p = p' ∧ Q' = { q ∈ Q. (∄q'. q ↦τ q')} ∧ (∄p''. p ↦τ p''))
          then id_up else None)› |

  conj_s_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Conjunct p' q)
      = (if p = p' ∧ q ∈ Q then (subtract 0 0 0 1 0 0 0 0)
         else None)› |

  empty_stbl_conj_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Defender_Conj p' Q')
      = (if Q = {} ∧ Q = Q' ∧ p = p' then (subtract 0 0 0 1 0 0 0 0)
         else None)› |

  br_conj:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p' α p'' Q' Qα)
      = (if (p = p' ∧ Q' = Q - Qα ∧ p ↦a α p'' ∧ Qα ⊆ Q) then id_up
         else None)› |

  br_answer:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Conjunct p'' q)
      = (if (p = p'' ∧ q ∈ Q) then (subtract 0 1 1 0 0 0 0 0) else None)› |

  br_obsv:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Branch p'' Q')
      = (if (p' = p'' ∧ Qα ↦aS α Q')
         then Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6) else None)› |

  br_acct:
    ‹spectroscopy_moves (Attacker_Branch p Q) (Attacker_Immediate p' Q')
      = (if p = p' ∧ Q = Q' then subtract 1 0 0 0 0 0 0 0 else None)› |

  others: ‹spectroscopy_moves _ _ = None›

fun spectroscopy_defender where
  ‹spectroscopy_defender (Attacker_Immediate _ _) = False› |
  ‹spectroscopy_defender (Attacker_Branch _ _) = False› |
  ‹spectroscopy_defender (Attacker_Conjunct _ _) = False› |
  ‹spectroscopy_defender (Attacker_Delayed _ _) = False› |
  ‹spectroscopy_defender (Defender_Branch _ _ _ _ _) = True› |
  ‹spectroscopy_defender (Defender_Conj _ _) = True› |
  ‹spectroscopy_defender (Defender_Stable_Conj _ _) = True›
```

8.2 Energy Game Properties

Now, we are able to define the weak spectroscopy game on an arbitrary LTS.

```
sublocale weak_spectroscopy_game:
  energy_game ‹spectroscopy_moves› ‹spectroscopy_defender› ‹(≤)›
```
⟨*proof*⟩

```
abbreviation ‹spectro_att_wins ≡ weak_spectroscopy_game.attacker_wins›
```

```
end — of lts_tau

end
```

9 Correctness

Energy levels where the defender wins in the spectroscopy game and equivalences coincide in the following sense: There exists a formula φ distinguishing a process `p` from a set of processes `Q` with expressiveness price of at most `e` if and only if `e` is in attacker's winning budget of `Attacker_Immediate p Q`.

The proof is split into two directions, closely following the structure of [3]. The forward direction is given by the lemma `distinction_implies_winning_budgets` combined with the upwards closure of winning budgets. To show the other direction, one can construct a (strategy) formula with an appropriate price using the constructive proof of `winning_budget_implies_strategy_formula`.

9.1 Distinction Implies Winning Budgets

```
theory Distinction_Implies_Winning_Budgets
  imports Spectroscopy_Game Expressiveness_Price
begin

context lts_tau
begin
```

We prove that if a formula distinguishes process `p` from a set of process `Q`, then the price of this formula is in attacker's winning budgets.

```
lemma distinction_implies_winning_budgets_empty_Q:
  assumes
    ‹distinguishes_from φ p {}›
  shows
    ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p {})›
  ⟨proof⟩

lemma distinction_implies_winning_budgets:
  assumes
    ‹distinguishes_from φ p Q›
  shows
    ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p Q)›
⟨proof⟩

end

end
```

9.2 Strategy Formulas

```
theory Strategy_Formulas
    imports Spectroscopy_Game Expressiveness_Price
begin
```

Strategy formulas express attacker strategies in HML. They bridge between HML formulas, the spectroscopy game and winning budgets. We show that, if some energy `e` suffices for the attacker to win, there exists a strategy formula with expressiveness price $\leq$ `e`. We also prove that this formula actually distinguishes the processes of the attacker position.

```
context lts_tau
begin

inductive
  strategy_formula
  :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's) hml_srbb ⇒ bool›
and
  strategy_formula_inner
```

```
  :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
and
  strategy_formula_conjunct
  :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool›
where
  delay:  ‹strategy_formula (Attacker_Immediate p Q) e (Internal χ)›
    if ‹∃Q'.
      spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p Q') = id_up
      ∧ spectro_att_wins e (Attacker_Delayed p Q')
      ∧ strategy_formula_inner (Attacker_Delayed p Q') e χ›
|
  procrastination:  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹∃p'.
       spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q) = id_up
       ∧ spectro_att_wins e (Attacker_Delayed p' Q)
       ∧ strategy_formula_inner (Attacker_Delayed p' Q) e χ›
|
  observation:  ‹strategy_formula_inner (Attacker_Delayed p Q) e (Obs α φ)›
    if ‹∃p' Q'. spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
       = (subtract 1 0 0 0 0 0 0 0)
       ∧ spectro_att_wins (e - E 1 0 0 0 0 0 0 0) (Attacker_Immediate p' Q')
       ∧ strategy_formula (Attacker_Immediate p' Q') (e - E 1 0 0 0 0 0 0 0) φ
       ∧ p ↦a α p' ∧ Q ↦aS α Q'›
|
  early_conj:  ‹strategy_formula (Attacker_Immediate p Q) e φ›
    if ‹∃p'. spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p' Q')
          = (subtract 0 0 0 0 1 0 0 0)
            ∧ spectro_att_wins (e - E 0 0 0 0 1 0 0 0) (Defender_Conj p' Q')
            ∧ strategy_formula (Defender_Conj p' Q') (e - E 0 0 0 0 1 0 0 0) φ›
|
  late_conj:  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹(spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p Q)
       = id_up ∧ (spectro_att_wins e (Defender_Conj p Q))
         ∧ strategy_formula_inner (Defender_Conj p Q) e χ)›
|
  conj:  ‹strategy_formula_inner (Defender_Conj p Q) e (Conj Q Φ)›
    if ‹∀q ∈ Q. spectroscopy_moves (Defender_Conj p Q) (Attacker_Conjunct p q)
        = (subtract 0 0 1 0 0 0 0 0)
          ∧ (spectro_att_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Conjunct p q))
          ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 0 1 0 0 0 0 0) (Φ q)›
|
  imm_conj:  ‹strategy_formula (Defender_Conj p Q) e (ImmConj Q Φ)›
    if ‹∀q ∈ Q. spectroscopy_moves (Defender_Conj p Q) (Attacker_Conjunct p q)
        = (subtract 0 0 1 0 0 0 0 0)
          ∧ (spectro_att_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Conjunct p q))
          ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 0 1 0 0 0 0 0) (Φ q)›
|
  pos:  ‹strategy_formula_conjunct (Attacker_Conjunct p q) e (Pos χ)›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed p Q')
      = Some min1_6 ∧ spectro_att_wins (the (min1_6 e)) (Attacker_Delayed p Q')
        ∧ strategy_formula_inner (Attacker_Delayed p Q') (the (min1_6 e)) χ)›
|
  neg:  ‹strategy_formula_conjunct (Attacker_Conjunct p q) e (Neg χ)›
    if ‹∃P'. (spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed q P')
      = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
        ∧ spectro_att_wins (the (min1_7 (e - E 0 0 0 0 0 0 0 1))) (Attacker_Delayed q P'))
        ∧ strategy_formula_inner (Attacker_Delayed q P')
                                 (the (min1_7 (e - E 0 0 0 0 0 0 0 1))) χ›
|
  stable:  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p Q')
```

```
      = id_up ∧ spectro_att_wins e (Defender_Stable_Conj p Q')
        ∧ strategy_formula_inner (Defender_Stable_Conj p Q') e χ)›
|
  stable_conj:  ‹strategy_formula_inner (Defender_Stable_Conj p Q) e (StableConj Q Φ)›
    if ‹∀q ∈ Q. spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Conjunct p q)
      = (subtract 0 0 0 1 0 0 0 0)
        ∧ spectro_att_wins (e - (E 0 0 0 1 0 0 0 0)) (Attacker_Conjunct p q)
        ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 0 0 1 0 0 0 0) (Φ q)›
|
  branch:  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹∃p' Q' α Qα. spectroscopy_moves (Attacker_Delayed p Q)
                      (Defender_Branch p α p' Q' Qα) = id_up
          ∧ spectro_att_wins e (Defender_Branch p α p' Q' Qα)
          ∧ strategy_formula_inner (Defender_Branch p α p' Q' Qα) e χ›
|
  branch_conj:
    ‹strategy_formula_inner (Defender_Branch p α p' Q Qα) e (BranchConj α φ Q Φ)›
    if ‹∃Q'. spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Branch p' Q')
          = Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)
            ∧ spectroscopy_moves (Attacker_Branch p' Q') (Attacker_Immediate p' Q')
            = subtract 1 0 0 0 0 0 0 0
            ∧ (spectro_att_wins (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - E 1 0 0 0 0 0 0 0)
                  (Attacker_Immediate p' Q'))
            ∧ strategy_formula (Attacker_Immediate p' Q')
                               (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - E 1 0 0 0 0 0 0 0) φ›
        ‹∀q ∈ Q. spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Conjunct p q)
          = (subtract 0 1 1 0 0 0 0 0)
          ∧ spectro_att_wins (e - (E 0 1 1 0 0 0 0 0)) (Attacker_Conjunct p q)
          ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 1 1 0 0 0 0 0) (Φ q)›

lemma winning_budget_implies_strategy_formula:
  assumes
    ‹spectro_att_wins e g›
  shows
    ‹case g of
      Attacker_Immediate p Q ⇒ ∃φ. strategy_formula g e φ ∧ expressiveness_price φ ≤ e
    | Attacker_Delayed p Q ⇒ ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner χ ≤ e
    | Attacker_Conjunct p q ⇒
        ∃ψ. strategy_formula_conjunct g e ψ ∧ expr_pr_conjunct ψ ≤ e
    | Defender_Conj p Q ⇒ ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner χ ≤ e
    | Defender_Stable_Conj p Q ⇒ ∃χ. strategy_formula_inner g e χ  ∧ expr_pr_inner χ ≤ e
    | Defender_Branch p α p' Q Qa ⇒
        ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner χ ≤ e
    | Attacker_Branch p Q ⇒
        ∃φ. strategy_formula (Attacker_Immediate p Q) (e - E 1 0 0 0 0 0 0 0) φ
          ∧ expressiveness_price φ ≤ e - E 1 0 0 0 0 0 0 0›
  ⟨proof⟩

lemma strategy_formulas_distinguish:
  shows
    ‹(strategy_formula g e φ ⟶
      (case g of
        Attacker_Immediate p Q ⇒  distinguishes_from φ p Q
      | Defender_Conj p Q ⇒ distinguishes_from φ p Q
      | _ ⇒ True))
      ∧
      (strategy_formula_inner g e χ ⟶
      (case g of
         Attacker_Delayed p Q ⇒ (Q ↠S Q) ⟶ distinguishes_from (Internal χ) p Q
      | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
      | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦ τ q)
```

```
      ⟶ hml_srbb_inner.distinguishes_from χ p Q
  | Defender_Branch p α p' Q Qa ⇒(p ↦a α p')
      ⟶ hml_srbb_inner.distinguishes_from χ p (Q∪Qa)
  | _ ⇒ True))
  ∧
  (strategy_formula_conjunct g e ψ ⟶
    (case g of
    Attacker_Conjunct p q ⇒ hml_srbb_conj.distinguishes ψ p q
  | _ ⇒ True))›
⟨proof⟩

end

end
```

9.3 Correctness Theorem

```
theory Silent_Step_Spectroscopy
  imports
    Distinction_Implies_Winning_Budgets
    Strategy_Formulas
begin
```

We now only combine the results of `Distinction_Implies_Winning_Budgets` and `Strategy_Formulas` to obtain the main characterization theorem of the weak spectroscopy game characterizing a whole spectrum of weak equivalences.

```
context lts_tau
begin

theorem spectroscopy_game_correctness:
  fixes e p Q
  shows ‹(∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e)
      ⟷ spectro_att_wins e (Attacker_Immediate p Q)›
⟨proof⟩
```

An implicit result of the correctness theorem is that attacker wins on bigger `Q` imply wins on smaller ones.

```
proposition attacker_subet_wins:
  assumes
    ‹spectro_att_wins e (Attacker_Immediate p Q)›
    ‹Q' ⊆ Q›
  shows
    ‹spectro_att_wins e (Attacker_Immediate p Q')›
  ⟨proof⟩

end

end
```

References

[1] Benjamin Bisping. Process equivalence problems as energy games. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 85–106, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-37706-8_5.

[2] Benjamin Bisping. *Generalized Equivalence Checking of Concurrent Programs*. PhD thesis, Technische Universität Berlin, 2025. URL: https://generalized-equivalence-checking.equiv.io/.

[3] Benjamin Bisping and David N. Jansen. Linear-time–branching-time spectroscopy accounting for silent steps, 2023. arXiv:2305.17671, doi:10.48550/arXiv.2305.17671.

[4] Benjamin Bisping and David N. Jansen. One energy game for the spectrum between branching bisimilarity and weak trace semantics. In Georgiana Caltais and Cinzia Di Giusto, editors, Proceedings Combined 31st International Workshop on *Expressiveness in Concurrency* and 21st Workshop on *Structural Operational Semantics,* Calgary, Canada, 9th September 2024, volume 412, pages 71–88. Open Publishing Association, 2024. doi:10.4204/EPTCS.412.6.

[5] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85, pages 299–309. Springer, Berlin, 1980. doi:10.1007/3-540-10003-2_79.

[6] Rob J. van Glabbeek. The linear time–branching time spectrum: extended abstract. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90*, volume 458, pages 278–297. Springer, Berlin, 1990. doi:10.1007/BFb0039066.

[7] Rob J. van Glabbeek. The linear time–branching time spectrum II: The semantics of sequential systems with silent moves; extended abstract. In Eike Best, editor, *CONCUR'93*, volume 715, pages 66–81. Springer, Berlin, 1993. doi:10.1007/3-540-57208-2_6.