A Weak Spectroscopy Game to Characterize Behavioral Equivalences

Lisa Annett Barthel Benjamin Bisping Leonard Moritz Hübner
Caroline Lemke Karl Parvis Philipp Mattes Lenard Mollenkopf

January 3, 2025

Abstract

We provide an Isabelle/HOL formalization of Bisping and Jansen's [1] weak spectroscopy game, which can be used to characterize a range of behavioral equivalences simultaneously, spanning from stability-respecting branching bisimilarity to weak trace equivalence. We relate distinguishing sublanguages of Hennessy-Milner Logic and attacker-winning budgets in an energy game by an eight-dimensional measure of formula expressiveness.

Contents

1 Introduction

Verification and asking wether a model fulfills its specification or if a program can be replaced with one that has the same behaviour are core problems of reactive systems and programming. For this we have to get an idea of what same behaviour for processes actually means and consider different behavioural equivalences. One possibility for this consideration are games where one player winning the game corresponds to the behavioural equivalences of processes. Alternatively, we could use a modal logic, known as Hennessy-Milner Logic (HML), not only to express the specification but also to build formulas that distinguish processes and thereby characterize behavioural equivalences. These techniques for checking whether two processes have the same behaviour may also be combined.

Previously, it was only possible to decide equivalence problems individually, but recently there have been ideas of deciding many of these problems at once. Therefore, Bisping and Jansen [1] included a measure of expressiveness for HML_{SRBB} formulas as eight-dimensional vectors. These vectors are added as costs to the moves of an extended delay bisimulation game such that the following property is obtained: The attacker wins a play with a certain initial energy e if and only if there is a formula that distinguishes the corresponding processes with a price less than or equal to e. Then the initial energy and the price of a formula encode the satisfied behavioural equivalences. It is therefore possible to decide for a whole spectrum of behavioural equivalences at the same time which of them apply [1].

We formalize the eight-dimensional weak spectroscopy game, which "can be used to decide a wide array of behavi[ou]ral equivalences between stability-respecting branching bisimilarity and weak trace equivalence in one go"[1, Abstract]. We then outline the proof of the correspondence between "attacker-winning energy budgets and distinguishing sublanguages of Hennessy-Milner [L]ogic characterized by eight dimensions of formula expressiveness"[1, Abstract]. With our formalization we try to follow [1] as closely as possible in how we formalize the weak spectroscopy game, HML and their correspondence. In doing so, we point out deviations in our formalization from and small corrections of the paper. This report documents the outcome of a project supervised by Benjamin Bisping at the Technical University of Berlin.

First, we formalize labelled transition systems with special handeling of τ-transitions. Afterwards, we describe our formalization of HML and a subset of HML, which we denote HML_{SRBB}. Within these HML sections, we define the semantics of such formulas and based on this prove several implications and equivalences on HML formulas. Additionally, we treat the notion of distinguishing formulas and especially distinguishing conjunctions. In the following sections, we present our formalization of energies as a data type and a price function for formulas. Before we formalize the weak spectroscopy game, we do the same for its basis in the form of energy games and define winning budgets on them. Following these fundamentals, we state our formalization of the theorem 1 of [1], that "relate[s] attacker-winning energy budgets and distinguishing sublanguages of Hennessy-Milner [L]ogic"[1, Abstract]. Based on the proof in [1] we outline a proof for this theorem through three lemmas. The first lemma states that given a distinguishing formula, the attacker is able to win the corresponding weak spectroscopy game. After introducing strategy formulas, we use induction to prove the second lemma, which claimes that if the attacker wins the weak spectroscopy game with an initial energy e, then there exists a (strategy) formula with a price less than or equal to e. Afterwards, the third lemma completes this cycle by stating that if there is a (strategy) formula, then it is a distingushing formula. Finally, we discuss the minor issues we found in the paper and thus present our contributions to [1] and end this report with a conclusion.

2 LTS

```
theory LTS
  imports Main
begin
```

2.1 Labelled Transition Systems

The locale `LTS` represents a labelled transition system consisting of a set of states $\mathcal{P}$, a set of actions Σ, and a transition relation $\mapsto \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$ (cf. [1, defintion 1]). 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`. Due to technical limitations we use the notation `p ↦α p'` which has same meaing as $p \xrightarrow{\alpha} p'$ has in [1].

```
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`. We define `P ↦S α Q` to be true if and only if for all states `q` in `Q` there exists a state `p` in `P` such that `p ↦ α q` and for all `p` in `P` and for all `q`, if `p ↦ α q` then `q` is in `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 possible α-steps for a set of states `P` are all `q` such that there is a state `p` in `P` with `p ↦ α q`.

```
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⟩
```

For a set of states `P` and an action α there exists exactly one `Q` such that `P ↦S α Q`.

```
lemma exactly_one_step_set: ‹∃!Q. P ↦S α Q›
⟨proof⟩
```

The lifted `step` (`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
```

2.2 Labelled Transition Systems with Silent Steps

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

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

The paper introduces a transition $p \xrightarrow{(\alpha)} p'$ if $p \xrightarrow{\alpha} p'$, or if $\alpha = \tau$ and $p = p'$ (cf. [1, defintion 2]). We define `soft_step` analagously 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)›
```

A state p is silent_reachable, represented by the symbol ↠, from another state p' iff there exists a path of τ-transitions. from p' to p.

```
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⟩
```

In the following, we define weak_step as a new notion of transition relation between states. A state p can reach p' 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_step's from one state p to another p' is called a weak_step_sequence That means that p' can be reached from p with that sequence of steps.

```
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 or all possible sequences of weak transitions that can be performed. In the context of labelled transition systems, weak traces capture the observable behaviour of a state.

```
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⟩
```

Since `weak_step`'s can be proceeded and followed by any number τ-transitions and the empty `weak_step_sequence` also allows τ-transitions, τ can be prepended to a weak trace of a state.

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

If state `p'` is reachable from state `p` via a sequence of τ-transitions and there exists a weak trace `tr` starting from `p'`, then `tr` is also a weak trace starting from `p`.

```
lemma silent_prepend_weak_traces:
  assumes ‹p ↠ p'›
      and ‹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'›
      and ‹tr ∈ weak_traces p'›
    shows ‹(α # tr) ∈ weak_traces p›
  ⟨proof⟩
```

One of the behavioural pre-orders/equivalences that we talk about is trace pre-order/equivalence. This is the modal characterization for one state is weakly trace pre-ordered to the other, `weakly_trace_preordered` denoted by `≲WT`, and two states are weakly trace equivalent, `weakly_trace_equivalent` denoted `≃WT`.

```
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 exactly_one_sreachable_set: ‹∃!Q. P ↠S Q›
⟨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⟩

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

end
```

2.3 Modal Logics on LTS

```
theory LTS_Semantics
  imports
    LTS
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⟩
```

The definition given above is equivalent which means 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⟩
```

If φ is equivalent to φ' and φ distinguishes process `p` from process `q`, the φ' also distinguishes process `p` from process `q`.

```
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}›
```

2.4 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 also `q` must 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 `p` and `q` if this set of formulas 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 `p` and `q`, then no formula in that set may distinguish `p` from `q` nor the other way around.

```
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

end
```

3 Stability-Respecting Branching Bisimilarity (HML_{SRBB})

```
theory HML_SRBB
  imports Main LTS_Semantics
begin
```

This section describes the largest subset of the full HML language in section ?? that we are using for purposes of silent step spectroscopy. It is supposed to characterize the most fine grained behavioural equivalence that we may decide: Stability-Respecting Branching Bisimilarity (SRBB). While there are good reasons to believe that this subset truly characterizes SRBB (c.f.[1]), we do not provide a formal proof. From this sublanguage smaller subsets are derived via the notion of expressiveness prices (5).

The mutually recursive data types `hml_srbb`, `hml_srbb_inner` and `hml_srbb_conjunct` represent the subset of all `hml` formulas, which characterize stability-respecting branching bisimilarity (abbreviated to 'SRBB').

When a parameter is of type `hml_srbb` we typically use φ as a name, for type `hml_srbb_inner` we use χ and for type `hml_srbb_conjunct` we use ψ.
The data constructors are to be interpreted as follows:

- in `hml_srbb`:
  - `TT` encodes $\top$
  - `(Internal` χ`)` encodes $\langle\varepsilon\rangle\chi$
  - `(ImmConj I` ψ`s)` encodes $\bigwedge_{i\in \text{I}} \psi s(i)$
- in `hml_srbb_inner`
  - `(Obs` α φ`)` encodes $(\alpha)\varphi$ (Note the difference to [1])
  - `(Conj I` ψ`s)` encode $\bigwedge_{i\in \text{I}} \psi s(i)$
  - `(StableConj I` ψ`s)` encodes $\neg\langle\tau\rangle\top \wedge \bigwedge_{i\in \text{I}} \psi s(i)$
  - `(BranchConj` α φ `I` ψ`s)` encodes $(\alpha)\varphi \wedge \bigwedge_{i\in \text{I}} \psi s(i)$
- in `hml_srbb_conjunct`
  - `(Pos` χ`)` encodes $\langle\varepsilon\rangle\chi$
  - `(Neg` χ`)` encodes $\neg\langle\varepsilon\rangle\chi$

For justifications regarding the explicit inclusion of `TT` and the encoding of conjunctions via index sets `I` and mapping from indices to conjuncts ψ`s`, reference the `TT` and `Conj` data constructors of the type `hml` in section ??.

```
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›
```

3.1 Semantics of HML$_{\text{SRBB}}$ Formulas

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

3.2 Different Variants of Verum

```
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⟩
```

⋀{(τ)⊤} is trivially true.

```
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⟩
```

3.3 Distinguishing Formulas

Now, we take a look at some basic properties of the `distinguishes` predicate:

⊤ can never distinguish two processes. This is due to the fact that every process satisfies `T`. Therefore, the second part of the definition of `distinguishes` never holds.

```
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⟩
```

If there is one conjunct in that distinguishes `p` from `q` and `p` satisfies all other conjuncts in a conjunction then $\bigwedge_{i \in I} \psi s(i)$ (where ψs ranges over the previously mentioned conjunctions) distinguishes `p` from `q`.

```
lemma srbb_dist_conjunct_implies_dist_imm_conjunction:
  assumes ‹i∈I›
      and ‹hml_srbb_conj.distinguishes (ψs i) p q›
      and ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
    shows ‹distinguishes (ImmConj I ψs) 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_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⟩
```

In the following, we replicate `srbb_dist_conjunct_implies_dist_imm_conjunction` for simple conjunctions in `hml_srbb_inner`.

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

We also replicate `srbb_dist_imm_conjunction_implies_dist_conjunct` for branching conjunctions $(\alpha)\varphi \wedge \bigwedge_{i \in I} \psi s(i)$. Here, either the branching condition distinguishes `p` from `q` or there must be a distinguishing conjunct.

```
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*⟩

In the following, we replicate `srbb_dist_conjunct_implies_dist_imm_conjunction` for branching conjunctions in `hml_srbb_inner`.

```
lemma srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction:
  assumes ‹∀i ∈ I. hml_srbb_conjunct_models p (ψs i)›
      and ‹hml_srbb_inner_models p (Obs α φ)›
      and ‹(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⟩
```

3.4 HML$_{\text{SRBB}}$ Implication

```
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›
```

3.5 HML$_{\text{SRBB}}$ Equivalence

We define HML$_{\text{SRBB}}$ formula equivalence to by appealing to HML$_{\text{SRBB}}$ implication. A HML$_{\text{SRBB}}$ formula is equivalent to another formula if both imply each other.

```
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›
```

3.6 Substitution

```
lemma srbb_internal_subst:
  assumes ‹χl ⟸χ⟹ χr›
      and ‹φ ⟸srbb⟹ (Internal χl)›
    shows ‹φ ⟸srbb⟹ (Internal χr)›
  ⟨proof⟩
```

3.7 Congruence

This section provides means to derive new equivalences by extending both sides with a given prefix.

Prepending ⟨ε⟩ . . . preserves equivalence.

```
lemma internal_srbb_cong:
  assumes ‹χl ⟸χ⟹ χr›
  shows ‹(Internal χl) ⟸srbb⟹ (Internal χr)›
  ⟨proof⟩
```

If equivalent conjuncts are included in an otherwise identical conjunction, the equivalence is preserved.

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

Prepending (α) . . . preserves equivalence.

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

3.8 Known Equivalence Elements

The formula $(\tau)\top$ is equivalent to $\bigwedge\{\}$.

```
lemma srbb_obs_τ_is_χTT: ‹Obs τ TT ⟸χ⟹ Conj {} ψs›
  ⟨proof⟩
```

The formula $(\alpha)\varphi$ is equivalent to $(\alpha)\varphi \wedge \bigwedge\{\}$.

```
lemma srbb_obs_is_empty_branch_conj: ‹Obs α φ ⟸χ⟹ BranchConj α φ {} ψs›
  ⟨proof⟩
```

The formula $\top$ is equivalent to $\langle\varepsilon\rangle \bigwedge\{\}$.

```
lemma srbb_TT_is_χTT: ‹TT ⟸srbb⟹ Internal (Conj {} ψs)›
  ⟨proof⟩
```

The formula $\top$ is equivalent to $\bigwedge\{\}$.

```
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:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes ((if distinguishes (Φ q) p q then conjunctify_distinctions
else conjunctify_distinctions_dual) Φ p q) p q›
⟨proof⟩

end

end
```

4 Energy

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

Following the paper [1, p. 5], we define energies as eight-dimensional vectors of natural numbers extended by ∞. But deviate from [1] in also defining an energy `eneg` that represents negative energy. This allows us to express energy updates (cf. [1, p. 8]) as total functions.

```
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›)
```

4.1 Ordering 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)›
```

Next, we show that this yields a reflexive transitive antisymmetric order.

```
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
```

We then use this order to define a predicate that decides if an `e1` may be subtracted from another `e2` without the result being negative. We encode this by `e1` being `somewhere_larger` than `e2`.

```
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⟩
```

4.2 Subtracting Energies

Using `somewhere_larger` we define subtraction as the `minus` operator on energies.

```
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
```

Afterwards, we prove 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⟩
```

We also define abbreviations for performing subtraction.

```
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)›
```

4.3 Minimum Updates

Next, we define 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)›
```

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 that these updates only decrease energies.

```
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 mono_min_1_6:
  shows ‹mono (the ∘ min1_6)›
⟨proof⟩

lemma mono_min_1_7:
  shows ‹mono (the ∘ min1_7)›
⟨proof⟩

lemma gets_smaller_min_1_6:
  shows ‹the (min1_6 x) ≤ x›
  ⟨proof⟩


lemma gets_smaller_min_1_7:
  shows ‹the (min1_7 x) ≤ x›
  ⟨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
```

5 Expressiveness Price Function

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

The expressiveness price function assigns a price - an eight-dimensional vector - to a $\mathrm{HML}_{\mathrm{SRBB}}$ formula. This price is supposed to capture the expressiveness power needed to describe a certain property and will later be used to select subsets of specific expressiveness power associated with the behavioural equivalence characterized by that subset of the $\mathrm{HML}_{\mathrm{SRBB}}$ sublanguage.

The expressiveness price function may be defined as a single function:

$$
\begin{aligned}
\mathit{expr}(\top) := \mathit{expr}^{\varepsilon}(\top) :=& 0\\
\mathit{expr}(\langle\varepsilon\rangle\chi) :=& \mathit{expr}^{\varepsilon}(\chi)\\
\mathit{expr}(\textstyle\bigwedge\Psi) :=& \hat{e}_5 + \mathit{expr}^{\varepsilon}(\textstyle\bigwedge\Psi)\\
\mathit{expr}^{\varepsilon}((\alpha)\varphi) :=& \hat{e}_1 + \mathit{expr}(\varphi)\\
\mathit{expr}^{\varepsilon}(\textstyle\bigwedge(\{(\alpha)\varphi\}\cup\Psi)) :=& \hat{e}_2 + \mathit{expr}^{\varepsilon}(\textstyle\bigwedge(\{\langle\varepsilon\rangle(\alpha)\varphi\}\cup\Psi\setminus\{(\alpha)\varphi\}))\\
\mathit{expr}^{\varepsilon}(\textstyle\bigwedge\Psi) :=& \sup\{\mathit{expr}^{\wedge}(\psi)|\psi\in\Psi\} + \begin{cases}\hat{e}_4 & \text{if} \neg\langle\tau\rangle\top\in\Psi\\ \hat{e}_3 & \text{otherwise}\end{cases}\\
\mathit{expr}^{\wedge}(\neg\langle\tau\rangle\top) :=& 0\\
\mathit{expr}^{\wedge}(\neg\varphi) :=& \sup\{\hat{e}_8 + \mathit{expr}(\varphi), (0,0,0,0,0,0,\mathit{expr}_1(\varphi),0)\}\\
\mathit{expr}^{\wedge}(\varphi) :=& \sup\{\mathit{expr}(\varphi), (0,0,0,0,0,\mathit{expr}_1(\varphi),0,0)\}
\end{aligned}
$$

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

Instead of defining the expressiveness price function in one go, we define eight functions (one for each dimension) and then use them in combination to build the the result vector.

Note that since all these functions stem from the above singular function, they all look very similar, but differ mostly in where the 1+ is placed.

5.1 Modal Depth

The (maximal) modal depth (of observations $\langle\alpha\rangle$, (α)) is increased on each:

- `Obs`
- `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›

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⟩
```

5.2 Depth of Branching Conjunctions

The depth of branching conjunctions (with one observation clause not starting with ⟨ε⟩) is increased on each:

- `BranchConj` if there are other conjuncts besides the branching conjunct

Note that if the `BranchConj` is empty (has no other conjuncts), then it is treated like a simple `Obs`.

```
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›
```

5.3 Depth of Stable Conjunctions

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_conju
∘ ψs) ` I))› |

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

5.4 Depth of Instable Conjunctions

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` if there are other conjuncts besides the branching conjunct

Note that if the `BranchConj` is empty (has no other conjuncts), then it is treated like a simple `Obs`.

```
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›
```

5.5 Depth of Immediate Conjunctions

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›
```

5.6 Maximal Modal Depth of Positive Clauses in Conjunctions

Now, we take a look at the maximal modal depth of positive clauses in conjunctions.
This counter 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_
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›
```

```
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⟩
```

5.7 Maximal Modal Depth of Negative Clauses in Conjunctions

We take a look at the maximal modal depth of negative clauses in conjunctions.
This counter 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›
```

```
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⟩
```

5.8 Depth of Negations

The depth of negations (occurrences of `Neg` χ 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›
```

5.9 Expressiveness Price Function

The `expressiveness_price` function combines the eight 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⟩
```

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⟩
```

Formalizing HML$_{SRBB}$ by mutually recursive data types leads to expressiveness price functions of these other types, namely `expr_pr_inner` and `expr_pr_conjunct`, 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)›

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

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)›

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

5.10 Prices of Certain Formulas

We demonstrate the pricing mechanisms for various formulas. These proofs operate under the assumption of an expressiveness price e for a given formula χ and proceed to determine the price of a derived formula such as `Pos χ`. The proofs all are of a similar nature. They decompose the expression function(s) into their constituent parts and apply their definitions to the constructed formula (`(Pos χ)`).

```
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⟩
```

If the price of a formula χ is not greater than the minimum update `min1_6` apllied to some energy e, then `Pos χ` is not greater than `e`.

```
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⟩

end
```

Here, we show the prices for some specific formulas.

```
locale Inhabited_LTS = LTS step
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80) +
  fixes left :: 's
    and right :: 's
  assumes left_right_distinct: ‹(left::'s) ≠ (right::'s)›

begin
```

```
lemma example_φ_cp:
  fixes op::‹'a› and a::‹'a› and b::‹'a›
  defines φ: ‹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›
  and ‹branching_conjunction_depth φ = 0›
  and ‹unstable_conjunction_depth  φ = 1›
  and ‹stable_conjunction_depth    φ = 0›
  and ‹immediate_conjunction_depth φ = 0›
  and ‹max_positive_conjunct_depth φ = 1›
  and ‹max_negative_conjunct_depth φ = 0›
  and ‹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⟩

end

context LTS_Tau
begin

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

5.11 Characterizing Equivalence by Energy Coordinates

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 $\mathcal{O}$ derived from `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 $\mathcal{O}$.

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

5.12 Relational Effects of 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
```

6 Weak Traces

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

The inductive `is_trace_formula` represents the modal-logical characterization of weak traces HML_{WT}. In particular:

- $\top \in \text{HML}_{WT}$ encoded by `is_trace_formula TT`, `is_trace_formula ImmConj I ψs` if `I = {}` and `is_trace_formula Conj I ψs` if `I = {}`..
- $\langle\varepsilon\rangle\chi \in \text{HML}_{WT}$ if $\varphi \in \text{HML}_{WT}$ encoded by `is_trace_formula Internal χ` if `is_trace_formula χ`.
- $(\alpha)\varphi \in \text{HML}_{WT}$ if $\varphi \in \text{HML}_{WT}$ encoded by `is_trace_formula Obs α φ` if `is_trace_formula φ`.
- $\bigwedge\{(\alpha)\varphi\} \cup \Psi \in \text{HML}_{WT}$ if $\varphi \in \text{HML}_{WT}$ and `Ψ = {}` encoded by `is_trace_formula BranchConj α φ I ψs` if `is_trace_formula φ` and `I = {}`.

```
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 ⊨ φ` 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
```

`wtrace_to_srbb trace` is in our modal-logical characterization of weak traces.

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

The following three lemmas show that the modal-logical characterization of HML_{WT} corresponds to the sublanguage of HML_{SRBB}, obtain by the energy coordinates `(∞, 0, 0, 0, 0, 0, 0, 0)`.

```
lemma trace_formula_to_expressiveness:
  fixes φ :: ‹('act, 'i) hml_srbb›
  fixes 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›
  fixes 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 formula φ is in HML_{WT} and a state `p` models φ, 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
      trace_case: ‹is_trace_formula φ ⟹ p ⊨SRBB φ ⟹ (∃tr ∈ weak_traces p. wtrace_to_srbb
tr ⇚srbb⇛ φ)›
    and conj_case: ‹is_trace_formula_inner χ ⟹ hml_srbb_inner_models q χ ⟹ (∃tr ∈ weak_traces
q. wtrace_to_inner tr ⇚χ⇛ χ)›
    and           True
⟨proof⟩
```

`t` is a weak trace of a state `p` if and only if `p` models the formula obtained from `wtrace_to_srbb t`.

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

If a state `p` weakly trace-pre-orders another state `q`, φ is in our modal-logical characterization HML_{WT}, and `p` models φ then `q` models φ.

```
lemma aux:
  fixes φ :: ‹('a, 's) hml_srbb›
  fixes χ :: ‹('a, 's) hml_srbb_inner›
  fixes ψ :: ‹('a, 's) hml_srbb_conjunct›
  shows ‹p ≲WT q ⟹ is_trace_formula φ ⟹ p ⊨SRBB φ ⟹ q ⊨SRBB φ›
⟨proof⟩
```

These are the main lemmas of this theory. They establish that the colloquial, relational notion of of weak trace pre-order/equivalence has the same distinctive power as the one derived from the coordinate (∞, `0, 0, 0, 0, 0, 0, 0`).

A state `p` weakly trace-pre-orders a state `q` iff and only if it also pre-orders `q` with respect to the coordinate (∞, `0, 0, 0, 0, 0, 0, 0`).

```
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 (∞, `0, 0, 0, 0, 0, 0, 0`).

```
lemma ‹(p ≃WT q) = (p ∼ (E ∞ 0 0 0 0 0 0 0) q)›
  ⟨proof⟩

end

end
```

7 η-Bisimilarity

```
theory Eta_Bisimilarity
  imports Expressiveness_Price
begin
```

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

```
context LTS_Tau
begin
```

— Following Def 2.1 in Divide and congruence

```
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⟩
```

7.2 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 ‹(p ~η q) = (p ∼ (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) q)›
  ⟨proof⟩
lemma modal_eta_sim: ‹eta_simulation (preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)))›
⟨proof⟩

theorem ‹(p ≼ (E ∞ ∞ ∞ 0 0 ∞ 0 0) q) ⟹ (∃R. eta_simulation R ∧ R p q)›
  ⟨proof⟩

end

end
```

8 Branching Bisimilarity

```
theory Branching_Bisimilarity
  imports Eta_Bisimilarity
begin
```

8.1 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›
```

8.2 Properties of Branching Bisimulation Equivalences

```
lemma branching_bisimilarity_branching_sim: ‹branching_simulation sr_branching_bisimulated›
  ⟨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 sr_branching_bisimulated›
  ⟨proof⟩

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

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

lemma sr_branching_bisimulated_sym:
  assumes
    ‹sr_branching_bisimulated p q›
  shows
    ‹sr_branching_bisimulated q 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⟩
```

8.3 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 ‹(p ~SRBB q) = (p ≼ (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞) q)›
  ⟨proof⟩

end
```

```
end
```

9 Energy Games

```
theory Energy_Games
  imports Main Misc
begin
```

In this theory, we introduce energy games and give basic definitions such as winning budgets. Energy games are the foundation for the later introduced weak spectroscopy game, which is an energy game itself, characterizing equivalence problems.

9.1 Fundamentals

We use an abstract concept of energies and only later consider eight-dimensional energy games. Through our later given definition of energies as a data type, we obtain certain properties that we enforce for all energy games. We therefore assume that an energy game has a partial order on energies such that all updates are monotonic and have sink where the defender wins.

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

An energy game is played by two players on a directed graph labelled by energy updates. These updates represent the costs of choosing a certain move. Since we will only consider cases in which the attacker's moves may actually have non-zero costs, only they can run out of energy. This is the case when the energy level reaches the `defender_win_level`. In contrast to other definitions of games, we do not fix a starting position.

```
locale energy_game =
fixes
  weight_opt :: ‹'gstate ⇒ 'gstate ⇒ 'energy update option› and
  defender :: ‹'gstate ⇒ bool› (‹Gd›) 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
```

In the following, we introduce some abbreviations for attacker positions and moves.

```
abbreviation attacker :: ‹'gstate ⇒ bool› (‹Ga›) where ‹Ga p ≡ ¬ Gd 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)›
```

9.1.1 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. The needed amount of energy is described by winning budgets: `e` is in the winning budget of `g` if and only if there exists a winning strategy for the attacker when starting in `g` with energy `e`. In more detail, this yields the following definition:

- If `g` is an attacker position and `e` is not the `defender_win_level` then `e` is in the winning budget of `g` if and only if there exists a position `g'` the attacker can move to. In other words, if the updated energy level is in the winning budget of `g'`. (This corresponds to the second case of the following definition.)
- If `g` is a defender position and `e` is not the `defender_win_level` then `e` is in the winning budget of `g` if and only if for all successors `g'` the accordingly updated energy is in the winning budget of `g'`. In other words, if the attacker will win from every successor the defender can move to.

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

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. This is proven using the inductive definition of winning budgets and the given properties of the partial order `ord`.

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

end

end
```

9.2 Instantiation of an Energy Game

```
theory Example_Instantiation
  imports Energy_Games "HOL-Library.Extended_Nat"
begin
```

In this theory, we create an instantiation of a two-dimensional energy game to test our definitions.

We first define energies in a similar manner to our definition of energies with two dimensions. We define component-wise subtraction.

```
datatype energy = E (one: ‹enat›) (two: ‹enat›)

abbreviation ‹direct_minus e1 e2 ≡ E ((one e1) - (one e2)) ((two e1) - (two e2))›

instantiation energy :: order
begin

fun less_eq_energy:: ‹energy ⇒ energy ⇒ bool› where
  ‹less_eq_energy (E ea1 ea2) (E eb1 eb2) = (ea1 ≤ eb1 ∧ ea2 ≤ eb2)›

fun less_energy:: ‹energy ⇒ energy ⇒ bool› where
  ‹less_energy eA eB = (eA ≤ eB ∧ ¬ eB ≤ eA)›

instance ⟨proof⟩
end
```

```
fun order_opt:: ‹energy option ⇒ energy option ⇒ bool› where
  ‹order_opt (Some eA) (Some eB) = (eA ≤ eB)› |
  ‹order_opt None _ = True› |
  ‹order_opt (Some eA) None = False›

definition minus_energy_def[simp]: ‹minus_energy e1 e2 ≡ if (¬e2 ≤ e1) then None
                                                  else Some (direct_minus e1 e2)›
lemma energy_minus[simp]:
  assumes ‹E c d ≤ E a b›
  shows ‹minus_energy (E a b) (E c d) = Some (E (a - c) (b - d))› ⟨proof⟩

definition min_update_def[simp]: ‹min_update e1 ≡ Some (E (min (one e1) (two e1)) (two e1))›
```

In preparation for our instantiation, we define our states, the updates for our energy levels and which states are defender positions.

```
datatype state = a | b1 | b2 | c | d1 | d2 | e

fun weight_opt :: ‹state ⇒ state ⇒ energy update option› where
  ‹weight_opt a b1 = Some (λx. minus_energy x (E 1 0))› |
  ‹weight_opt a b2 = Some (λx. minus_energy x (E 0 1))› |
  ‹weight_opt a _  = None›  |
  ‹weight_opt b1 c = Some Some› |
  ‹weight_opt b1 _  = None›  |
  ‹weight_opt b2 c = Some min_update› |
  ‹weight_opt b2 _  = None›  |
  ‹weight_opt c d1 = Some (λx. minus_energy x (E 0 1))› |
  ‹weight_opt c d2 = Some (λx. minus_energy x (E 1 0))› |
  ‹weight_opt c _  = None›  |
  ‹weight_opt d1 e = Some Some› |
  ‹weight_opt d1 _  = None›  |
  ‹weight_opt d2 e = Some Some› |
  ‹weight_opt d2 _  = None› |
  ‹weight_opt e _  = None›

find_theorems weight_opt

fun defender :: ‹state ⇒ bool› where
  ‹defender b1 = True› |
  ‹defender b2 = True› |
  ‹defender c = True› |
  ‹defender e = True› |
  ‹defender _ = False›
```

Now, we can state our energy game example.

```
interpretation Game: energy_game ‹weight_opt› ‹defender› ‹(≤)›
⟨proof⟩

notation Game.moves (infix ‹↣› 70)

lemma moves:
  shows ‹a ↣ b1› ‹a ↣ b2›
        ‹b1 ↣ c› ‹b2 ↣ c›
        ‹c ↣ d1› ‹c ↣ d2›
        ‹d1 ↣ e› ‹d2 ↣ e›
        ‹¬(c ↣ e)› ‹¬(e ↣ d1)›
  ⟨proof⟩
```

Our definition of winning budgets.

```
lemma wina_of_e:
  shows ‹Game.attacker_wins (E 9 8) e›
  ⟨proof⟩
```

```
lemma wina_of_e_exist:
  shows ‹∃e1. Game.attacker_wins e1 e›
  ⟨proof⟩

lemma attacker_wins_at_e:
  shows ‹∀e'. Game.attacker_wins e' e›
  ⟨proof⟩

lemma wina_of_d1:
  shows ‹Game.attacker_wins (E 9 8) d1›
⟨proof⟩

lemma wina_of_d2:
  shows ‹Game.attacker_wins (E 8 9) d2›
⟨proof⟩

lemma wina_of_c:
  shows ‹Game.attacker_wins (E 9 9) c›
⟨proof⟩

lemma not_wina_of_c:
  shows ‹¬Game.attacker_wins (E 0 0) c›
⟨proof⟩

end
```

10 Weak Spectroscopy Game

```
theory Spectroscopy_Game
  imports Energy_Games Energy LTS
begin
```

In this theory, we define the weak spectroscopy game as a locale. This game is an energy game constructed by adding stable and branching conjunctions to a delay bisimulation game that depends on a LTS. We play the weak spectroscopy game to compare the behaviour of processes and analyze which behavioural equivalences apply. The moves of a weak spectroscopy game depend on the transitions of the processes and the available energy. So in other words: If the defender wins the weak spectroscopy game starting with a certain energy, the corresponding behavioural equivalence applies.
Since we added adding stable and branching conjunctions to a delay bisimulation game, we differentiate the positions accordingly.

```
datatype ('s, 'a) spectroscopy_position =
         Attacker_Immediate (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
         Attacker_Branch (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
         Attacker_Clause (attacker_state: ‹'s›) (defender_state: ‹'s›) |
         Attacker_Delayed (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›) |
         Defender_Conj (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
         Defender_Stable_Conj (attacker_state: ‹'s›) (defender_states: ‹'s set›)

context LTS_Tau begin
```

We also define the moves of the weak spectroscopy game. Their names indicate the respective HML formulas they correspond to. This correspondence will be shown in section 11.2.

```
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 Some Some else None)› |

  procrastination:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q')
      = (if (Q' = Q ∧ p ≠ p' ∧ p ↦ τ p') then Some Some 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 Some Some else None)› |

  conj_answer:
    ‹spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause 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_Clause 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 Some Some else None)› |

  conj_s_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Clause 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 Some Some
         else None)› |

  br_answer:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Clause 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_Clause _ _) = False› |
  ‹spectroscopy_defender (Attacker_Delayed _ _) = False› |
  ‹spectroscopy_defender (Defender_Branch _ _ _ _ _) = True› |
  ‹spectroscopy_defender (Defender_Conj _ _) = True› |
  ‹spectroscopy_defender (Defender_Stable_Conj _ _) = True›

interpretation Game: energy_game ‹spectroscopy_moves› ‹spectroscopy_defender› ‹(≤)›
⟨proof⟩

end
```

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

```
locale weak_spectroscopy_game =
  LTS_Tau step τ
  + energy_game ‹spectroscopy_moves› ‹spectroscopy_defender› ‹less_eq›
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦_ _› [70, 70, 70] 80) and
      τ :: 'a

end
```

11 Correctness

As in the main theorem of [1], we state in what sense winning energy levels and equivalences coincide as the theorem `spectroscopy_game_correctness`: 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 the winning budget of `Attacker_Immediate p Q`.
The proof is split into three lemmas. The forward direction is given by the lemma `distinction_implies_winning_budg` 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`. From lemma `strategy_formulas_distinguish` we then know that this formula actually distinguishes `p` from `Q`.

11.1 Distinction Implies Winning Budgets

```
theory Distinction_Implies_Winning_Budgets
  imports Spectroscopy_Game Expressiveness_Price
begin

context weak_spectroscopy_game
begin
```

In this section, we prove that if a formula distinguishes a process `p` from a set of process `Q`, then the price of this formula is in the attackers-winning budget. This is the same statement as that of lemma 1 in the paper [1, p. 20]. We likewise also prove it in the same manner.
First, we show that the statement holds if `Q = {}`. This is the case, as the attacker can move, at no cost, from the starting position, `Attacker_Immediate p {}`, to the defender position `Defender_Conj p {}`. In this position the defender is then unable to make any further moves. Hence, the attacker wins the game with any budget.

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

Next, we show the statement for the case that `Q` $\neq$ `{}`. Following the proof of [1, p. 20], we do this by induction on a more complex property.

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

end

end
```

11.2 Strategy Formulas

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

In this section, we introduce strategy formulas as a tool of proving the corresponding lemma, `spectroscopy_game_correctness`, in section 11.3. We first define strategy formulas, creating a bridge between HML formulas, the spectroscopy game and winning budgets. We then show that for some energy `e` in a winning budget there exists a strategy formula with expressiveness price $\leq$ `e`. Afterwards, we prove that this formula actually distinguishes the corresponding processes.

```
context weak_spectroscopy_game
begin
```

We define strategy formulas inductively. For example for $\langle\alpha\rangle\varphi$ to be a strategy formula for some attacker delayed position `g` with energy `e` the following must hold: φ is a strategy formula at the from `g` through an observation move reached attacker (immediate) position with the energy `e` updated according to the move. Then the function `strategy_formula_inner g e` $\langle\alpha\rangle\varphi$ returns true. Similarly, every derivation rule for strategy formulas corresponds to possible moves in the spectroscopy game. To account for the three different data types a HML_{SRBB} formula can have in our formalization, we define three functions at the same time:

```
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')
        = (Some Some)) ∧ (attacker_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)
         = (Some Some) ∧ attacker_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)
          ∧ attacker_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 ↦aα 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)
                 ∧ attacker_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)
         = (Some Some) ∧ (attacker_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_Clause p q)
          = (subtract 0 0 1 0 0 0 0 0)
            ∧ (attacker_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Clause p q))
            ∧ strategy_formula_conjunct (Attacker_Clause 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_Clause p q)
        = (subtract 0 0 1 0 0 0 0 0)
          ∧ (attacker_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Clause p q))
          ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 0 1 0 0 0 0 0)) (Φ
q)› |

  pos:
  ‹strategy_formula_conjunct (Attacker_Clause p q) e (Pos χ)›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p Q')
      = Some min1_6 ∧ attacker_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_Clause p q) e (Neg χ)›
    if ‹∃P'. (spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed q P')
      = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
        ∧ attacker_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')
      = (Some Some) ∧ attacker_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_Clause p q)
       = (subtract 0 0 0 1 0 0 0 0)
         ∧ attacker_wins (e - (E 0 0 0 1 0 0 0 0)) (Attacker_Clause p q)
         ∧ strategy_formula_conjunct (Attacker_Clause 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)
      = (Some Some) ∧ attacker_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
          ∧ (attacker_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_Clause p q)
         = (subtract 0 1 1 0 0 0 0 0)
         ∧ attacker_wins (e - (E 0 1 1 0 0 0 0 0)) (Attacker_Clause p q)
         ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 1 1 0 0 0 0 0)) (Φ
q)›
```

To prove `spectroscopy_game_correctness` we need the following implication: If `e` is in the winning budget of `Attacker_Immediate p Q`, there is a strategy formula φ for `Attacker_Immediate p Q` with energy `e` with expressiveness price $\leq$ `e`.

We prove a more detailed result for all possible game positions `g` by induction over the structure of winning budgets (Cases $1-3$:

1. We first show that the statement holds if `g` has no outgoing edges. This can only be the case for defender positions.
2. If `g` is an attacker position, by `e` being in the winning budget of `g`, we know there exists a successor of `g` that the attacker can move to. If by induction the property holds true for that successor, we show that it then holds for `g` as well.
3. If a defender position `g` has outgoing edges and the statement holds true for all successors, we show that the statement holds true for `g` as well.

```
lemma winning_budget_implies_strategy_formula:
  fixes g e
  assumes ‹attacker_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_Clause 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⟩
```

To prove `spectroscopy_game_correctness` we need the following implication: If φ is a strategy formula for `Attacker_Immediate p Q` with energy `e`, then φ distinguishes `p` from `Q`.

We prove a more detailed result for all possible game positions `g` by induction. Note that the case of `g` being an attacker branching position is not explicitly needed as part of the induction hypothesis but is proven as a part of case `branch_conj`. The induction relies on the inductive structure of strategy formulas.
Since our formalization differentiates immediate conjunctions and conjunctions, two `Defender_Conj` cases are necessary. Specifically, the strategy construction rule `early_conj` uses immediate conjunctions, while `late_conj` uses conjunctions.

```
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_Clause p q ⇒ hml_srbb_conj.distinguishes ψ p q
      | _ ⇒ True))›
⟨proof⟩

end

end
```

11.3 Correctness Theorem

```
theory Silent_Step_Spectroscopy
  imports
    Distinction_Implies_Winning_Budgets
    Strategy_Formulas
begin

context weak_spectroscopy_game
begin

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

end

end
```

12 Conclusion

We were able to formalize the majority of the paper, including the weak spectroscopy game as introduced by Bisping and Jansen in [1], and to prove one direction of the theorem stating correctness, namely 'if the attacker wins the weak spectroscopy game, given an energy e, then there exists a formula $\varphi \in \mathrm{HML}_{\mathrm{SRBB}}$ with price $\mathrm{expr}(\varphi) \leq e$'(c.f. [1, lemma 2, 3]). For the other direction, we provide a comprehensive proof skeleton, including proofs for individual induction cases.

Due to the nature of Isabelle, the formalization differs from [1]. The gravest change is to the definition of $\mathrm{HML}_{\mathrm{SRBB}}$. We have implemented this definition using three mutually recursive data types. As a result, we had two definitions for a conjunction $\bigwedge \psi$, `ImmConj` and `Conj`, each with a different type. The other difference to the $\mathrm{HML}_{\mathrm{SRBB}}$ definition of [1] concerns the observation of actions. We argue that both definitions have the same distinguishing power. These changes led to necessary adaptations of our definition of the weak spectroscopy game and thereby affected the following definitions and proofs. An overview of these and other deviations can be found in appendix **??**.

A major change compared to [1] is the addition of new game move $(p, \emptyset)_d^s \overset{\hat{e}_4}{\rightarrowtail} (p, \emptyset)_d$ from `Defender_Stable_Conj` to `Defender_Conj` if $Q = \emptyset$. Without this move, the attacker could use an empty stability conjunction `StableConj` without having the proper budget. We formalized a weak spectroscopy game closely related to [1] that can decide (almost) all behavioural equivalences between stability-respecting branching bisimilarity and weak trace equivalence at once. Provided our definition of energies as eight-dimensional vectors corresponds to these equivalences, we implemented a (mostly) machine-checkable proof for the correctness of this spectroscopy game.

To further increase confidence in the results of [1], additional proofs are necessary. Firstly, the proof for 'given an energy e, if there exists a formula $\varphi \in \mathrm{HML}_{\mathrm{SRBB}}$ with price $\mathrm{expr}(\varphi) \leq e$, then the attacker wins the weak spectroscopy game' is senseful (c.f. [1, lemma 1]). Secondly, [1] uses coordinates of energies to define equivalences. One can show that the HML sublanguages obtained from these coordinates correspond to the desired equivalences. Since our formalization of the model relation `hml_models` is only defined on the parameterization of HML by the state type `'s`, one could also show that this formalization sufficiently captures the expressiveness power of HML on labelled transition systems. Finally, [1, proposition 1] claims that their slightly different modal characterization of $\mathrm{HML}_{\mathrm{SRBB}}$ corresponds to the modal characterization of [2]. The proof for proposition 1 in [1] could be turned into a machine-checkable proof.

References

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

[2] W. Fokkink, R. van Glabbeek, and B. Luttik. Divide and congruence iii: From decomposition of modal formulas to preservation of stability and divergence. *Information and Computation*, 268:104435, 2019.