Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

20th February 2024

Contents

Chapter 1

Introduction

In this thesis, I show the correspondence between various equivalences popular in the reactive systems community and coordinates of a formula price function, as introduced by Benjamin Bisping (citation). I formalised the concepts and proofs discussed in this thesis in the interactive proof assistant Isabelle (citation).

Reactive systems are computing systems that continuously interact with their environment, reacting to external stimuli and producing outputs accordingly(Harel). At a high level of abstraction, they can be seen as a collection of interacting processes. Modeling and verifying these processes is often referred to as *Process Theory.*

Verification of these systems involves proving statements regarding the behavior of a system model. Often, verification tasks aim to show that a system's observed behavior aligns with its intended behavior. That requires a criterion of similar behavior, or *semantics of equality.* Depending on the requirements of a particular user, many different such criterions have been defined. For a subset of processes, namely the class of sequential processes lacking internal behavior, (Glaabbeck) classified many such semantics. The processes in this subset can only perform one action at a time. Furthermore, this class is restricted to *concrete* processes; processes in which no internal actions occur. This classification involved partially ordering them by the relation 'makes strictly more identifications on processes than' (Glabbeeck). The resulting complete lattice is referred to as the (infinitary) linear-time–branching-time spectrum. [1] [2]

[1] On Infinity?

[2] linear time describes identification via the order of events, while branching time captures the branching possibilities in system executions.

more on LT BT spectrum?

Systems with this kind of processes can be modeled using labeled transition systems (Kel). An LTS is a triple of a set of processes, or states of the system, a set of possible actions and a transition relation between a process, an action and another process. The outgoing transitions of each process correspond to the actions the system can perform in that state, yielding a subsequent state. In accordance with our restriction to concrete processes, we do not distinguish between different kinds of actions. [3]

In the context of this spectrum, demonstrating that a system model's observed behavior aligns with the behavior of a model of the specification involves finding the finest notions of behavioral equivalence that equate them. Special bisimulation games and algorithms capable of answering equivalence questions by performing a 'spectroscopy' of the differences between two processes have been developed (Deciding all at once)((accounting for silent steps), evtl hier weglassen oder mention: anderes spektrum)(process equiv as energy games)(A game for lt bt spectr). These approaches rechart the linear-time–branching-time spectrum using *formula prices* that capture the expressive capabilities of Hennessy-Milner Logic (HML).

This thesis provides a machine-checkable proof that certain price bounds correspond to the modal-logical characterizations of named equivalences. More precisely, a formula φ is in an observation language $\mathcal{O}_{\mathcal{X}}$ iff its price is within the given price bound. Concretely, for every expressiveness price bound e_X, i derive the sublanguage of Hennessy–Miler logic $\mathcal{O}_{\mathcal{X}}$ and show that a formula φ is in $\mathcal{O}_{\mathcal{X}}$ precisely if its price $\texttt{expr}(\varphi)$ is less than or equal to e_X. Then i show that $\mathcal{O}_{\mathcal{X}}$ has exactly the same distinguishing power as the modal-logical characterization of that equivalence.

For the class of sequential processes, that can at most perform one action at a time, and that do not posses internal behavior. (cite glabbeeck) classified many such semantics by partially ordering them by the relation 'makes strictly more identifications on processes than'. However, The term *reactive system* (citation) describes computing systems that continuously interact with their environment. Unlike sequential systems, the behavior or reactive systems is inherently event-driven and concurrent. They can be modeled by labeled directed graphs called *labeled transition systems* (LTSs) (citation), where the nodes of an LTS describe the states of a reactive system and the edges describe transitions between those states.

Strucutre:

Foundations: LTS, Bismilarity (weil besonders dadurch das es feinste äqui-

[3] A popular notion of identification is internal behavior, LTS capable of modeling internal behavior use a fixed action to express internal behavior. This extension allows for additional semantics that have been investigated, for instance, in (Glabeeck).

valenz ist,verbindung zu HML(HM Theorem), HML
Formula Pricing - capturing expressiveness using formula prices
Korrespondenz zwischen koordinaten und äquivalenzen beweise
diskussion?
appendix?

The semantics of reactive systems can be modeled as equivalences, that determine whether or not two systems behave similarly. In the literature on concurrent systems many different notion of equivalence can be found, the maybe best known being *(strong) bisimilarity.* Rab van Glabbeek's *linear-time–branching-spectrum*(citation) ordered some of the most popular in a hierachy of equivalences. -> New Paper characterizes them different... (HML beschreibung als erstes?!!)

- Reactive Systmes
- modelling (via lts etc)
- Semantics of resysts
- Verification
- different notions of equivalence (because of nondeteminism?) -> van glabbeek
- Different definitions of semantics -> HML/relational/...

–> linear-time–branching-time spectrum understood through properties of HML
–> capture expressiveness capabilities of HML formulas via a function
–> Contribution o Paper: The in (citation) introduced expressiveness function and its coordinates captures the linear time branching time spectrum..

- Isabelle:
- formalization of concepts, proofs
- what is isabelle
- difference between mathematical concepts and their implementation?

Chapter 2

Foundations

In this chapter, relevant concepts will be introduced as well as formalised in Isabelle.

- mention sources (Ben / Max Pohlmann?)

2.1 Labeled Transition Systems

As mentioned in (Introduction), labeled transition systems are formal models used to describe the behavior of reactive systems. A LTS consists of three components: processes, actions, and transitions. Processes represent momentary states or configurations of a system. Actions denote the events or operations that can occur within the system. The outgoing transitions of each process correspond to the actions the system can perform in that state, yielding a subsequent state. A process may have multiple outgoing transitions labeled with the same or different actions. This signifies that the system can choose any of these transitions nondeterministically [1]. The semantic equivalences treated in (Glabbeeck) are defined entirely in terms of action relations. We treat processes as being *sequential*, meaning it can perform at most one action at a time, and instantanious. Note that many modeling methods of systems use a special τ-action to represent internal behavior. However, in our definition of LTS, internal behavior is not considered.

[1] Note that "nondeterministic" has been used differently in some of the literature (citation needed). In the context of reactive systems, all transitions are directly triggered by external actions or events and represent synchronization with the environment. The next state of the system is then uniquely determined by its current state and the external action. In that sense the behavior of the system is deterministic.

Definition 1.1 (Labeled transition Systems)

A *Labeled Transition System* (LTS) is a tuple $\mathcal{S} = (Proc, Act, \rightarrow)$ where $Proc$ is the set of processes, Act is the set of actions and $\rightarrow \subseteq Proc \times Act \times Proc$ is a transition relation. We write $p \rightarrow \alpha p'$ for $(p, \alpha, p') \in\rightarrow$.

Actions and processes are formalized using type variable `'a` and `'s`, respectively. As only actions and states involved in the transition relation are relevant, the set of transitions uniquely defines a specific LTS. We express this relationship using the predicate `tran`. We associate it with a more readable notation (`p ↦α p'` for $p \xrightarrow{\alpha} p'$).

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

Example... (to reuse later?)

We introduce some concepts to better talk about LTS. Note that these Isabelle definitions are only defined in the `context` of LTS.

Definition 1.2

The *α-derivatives* of a state refer to the set of states that can be reached with an α-transition:

$$mathitDer(p, \alpha) = \{p' \mid p \xrightarrow{\alpha} p'\}.$$

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

The set of *initial actions* of a process p is defined by:

$$I(p) = \alpha \in Act \mid \exists p'.p \xrightarrow{\alpha} p'$$

```
abbreviation initial_actions:: ‹'s ⇒ 'a set›
  where
‹initial_actions p ≡ {α|α. (∃p'. p ↦α p')}›
```

The step sequence relation $\xrightarrow{\sigma} *$ for $\sigma \in Act$ is the reflexive transitive closure of $p \xrightarrow{\alpha} p'$. It is defined recursively by:

$$p \xrightarrow{\varepsilon}{}^{*} p$$

$$p \xrightarrow{\alpha} p' \text{ with } \alpha \in \text{Act and } p' \xrightarrow{\sigma}{}^{*} p'' \text{ implies } p' \xrightarrow{\sigma}{}^{*} p''$$

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

p is image-finite if for each $\alpha \in Act$ the set $mathitDer(p, \alpha)$ is finite. An LTS is image-finite if each $p \in Proc$ is image-finite: ”

$$\forall p \in Proc, \alpha \in Act.mathitDer(p, \alpha)$$

is finite.

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

We say that a process is in a *deadlock* if no observation is possible. That is:

$$deadlock p = (\forall \alpha . deadlock p \alpha = \varnothing)$$

```
abbreviation deadlock :: ‹'s ⇒ bool› where
‹deadlock p ≡ (∀α. derivatives p α = {})›
```

nötig?

```
definition image_countable :: ‹bool›
  where ‹image_countable ≡ (∀ p α. countable (derivatives p α))›
```

stimmt definition? definition benötigt nach umstieg auf sets?

```
definition lts_finite where
‹lts_finite ≡ (finite (UNIV :: 's set))›

abbreviation relevant_actions :: ‹'a set›
  where
‹relevant_actions ≡ {a. ∃p p'. p ↦a p'}›


end
```

2.2 Behavioral Equivalence of Processes

As discussed in the previous sections, LTSs model the behaviour of reactive systems. That behaviour is observable by the environment in terms of transitions performed by the system. Depending on different criteria on what constitutes equal behavior has led to a large number of equivalences for concurrent processes. Those equivalences are often defined in term of relations on LTSs or sets of executions. The finest commonly used *extensional* *behavioral equivalence* is *Bisimilarity*. In extensional equivalences, only observable behavior is taken into account, without considering the identity of the processes. This sets bisimilarity apart from stronger graph equivalences like *graph isomorphism*, here the (intensional) identity of processes is relevant. The coarsest commonly used equivalence is *trace equivalence*.

- LT-BT spectrum (between them there is a lattice of equivalences ...) - Wir behandeln bisimilarität hier gesondert wegen dessen bezihung zu HML (HM-Theorem) (s.h. Introduction, doppelung vermeiden). - example bisimilarity

Informally, we call two processes bisimilar if...

Bisimilarity

The notion of strong bisimilarity can be formalised through *strong bisimulation* (SB) relations, introduced originally in (citation Park). A binary relation $\mathcal{R}$ over the set of processes *Proc* is an SB iff for all $(p, q) \in \mathcal{R}$:

$$\forall p' \in \mathit{Proc},\ \alpha \in \mathit{Act}.\ p \xrightarrow{\alpha} p' \longrightarrow \exists q' \in \mathit{Proc}.\ q \xrightarrow{\alpha} q' \wedge (p', q') \in \mathcal{R},\ \text{and}$$
$$\forall q' \in \mathit{Proc},\ \alpha \in \mathit{Act}.\ q \xrightarrow{\alpha} q' \longrightarrow \exists p' \in \mathit{Proc}.\ p \xrightarrow{\alpha} p' \wedge (p', q') \in \mathcal{R}.$$

end

2.3 Hennessy–Milner logic

For the purpose of this thesis, we focus on the modal-logical characterizations of equivalences, using Hennessy–Milner logic (HML). First introduced by Matthew Hennessy and Robin Milner (citation), HML is a modal logic for expressing properties of systems described by LTS. Intuitively, HML describes observations on an LTS and two processes are considered equivalent under HML when there exists no observation that distinguishes between them. (citation) defined the modal-logical language as consisting of (finite) conjunctions, negations and a (modal) possibility operator:

$$\varphi ::= \mathit{tt} \mid \varphi_1 \ \wedge \ \varphi_2 \mid \neg\varphi \mid \langle\alpha\rangle\varphi$$

(where α ranges over the set of actions. The paper also proves that this characterization of strong bisimilarity corresponds to a relational definition that is effectively the same as in (...). Their result can be expressed as follows: for image-finite LTSs, two processes are strongly bisimilar iff they satisfy the same set of HML formulas. We call this the modal characterisation of strong bisimilarity. By allowing for conjunction of arbitrary width (infinitary HML), the modal characterization of strong bisimilarity can be proved for arbitrary LTS. This is done in (...)

Mention: HML to capture equivalences (Spectrum, HM theorem)

Hennessy–Milner logic

The syntax of Hennessy–Milner logic over a set Σ of actions, (HML) - richtige font!!!!![Σ], is defined by the grammar:

$$\varphi ::= \langle a \rangle \varphi \qquad \text{with } a \in \Sigma$$
$$\mid \bigwedge_{i \in I} \psi_i$$
$$\psi ::= \neg\varphi \mid \varphi.$$

The data type `('a, 'i)hml` formalizes the definition of HML formulas above. It is parameterized by the type of actions `'a` for Σ and an index type `'i`. We use an index sets of arbitrary type `I :: 'i set` and a mapping `F :: 'i ⇒ ('a, 'i) hml` to formalize conjunctions so that each element of `I` is mapped to a formula[2]

```
datatype ('a, 'i)hml =
TT |
hml_pos ‹'a› ‹('a, 'i)hml› |
hml_conj 'i set 'i set 'i ⇒ ('a, 'i) hml
```

Note that in canonical definitions of HML `TT` is not usually part of the syntax, but is instead synonymous to $\bigwedge$`{}`. We include `TT` in the definition to enable Isabelle to infer that the type `hml` is not empty.. This formalization allows for conjunctions of arbitrary - even of infinite - width and has been taken from [?] (Appendix B).

```
context lts begin

primrec hml_semantics :: ‹'s ⇒ ('a, 's)hml ⇒ bool›
(‹_ ⊨ _› [50, 50] 50)
where
hml_sem_tt: ‹(_ ⊨ TT) = True› |
hml_sem_pos: ‹(p ⊨ (hml_pos α φ)) = (∃ q. (p ↦α q) ∧ q ⊨ φ)› |
hml_sem_conj: ‹(p ⊨ (hml_conj I J ψs)) = ((∀i ∈ I. p ⊨ (ψs i)) ∧ (∀j
∈ J. ¬(p ⊨ (ψs j))))›

definition HML_true where
HML_true φ ≡ ∀s. s ⊨ φ

lemma
  fixes s::'s
  assumes HML_true (hml_conj I J Φ)
  shows ∀φ ∈ Φ ` I. HML_true φ
  ⟨proof⟩
```

[2] Note that the formalization via an arbitrary set (...) does not yield a valid type, since `set` is not a bounded natural functor.

Two states are HML equivalent if they satisfy the same formula.

```
definition HML_equivalent :: ‹'s ⇒ 's ⇒ bool› where
  ‹HML_equivalent p q ≡ (∀ φ::('a, 's) hml. (p ⊨ φ) ⟷ (q ⊨ φ))›
```

An HML formula φl implies another (φr) if the fact that some process p satisfies φl implies that p must also satisfy φr, no matter the process p.

```
definition hml_impl :: ('a, 's) hml ⇒ ('a, 's) hml ⇒ bool (infix ⇛
60)  where
  φl ⇛ φr ≡ (∀p. (p ⊨ φl) ⟶ (p ⊨ φr))

lemma hml_impl_iffI: φl ⇛ φr = (∀p. (p ⊨ φl) ⟶ (p ⊨ φr))
  ⟨proof⟩
```

Equivalence

A HML formula φl is said to be equivalent to some other HML formula φr (written φl ⇚⇛ φr) iff process p satisfies φl iff it also satisfies φr, no matter the process p.
We have chosen to define this equivalence by appealing to HML formula implication (c.f. pre-order).

```
definition hml_formula_eq :: ('a, 's) hml ⇒ ('a, 's) hml ⇒ bool (infix
⇚⇛ 60)  where
  φl ⇚⇛ φr ≡ φl ⇛ φr ∧ φr ⇛ φl
```

⇚⇛ is truly an equivalence relation.

```
lemma hml_eq_equiv: equivp (⇚⇛)
  ⟨proof⟩

lemma equiv_der:
  assumes HML_equivalent p q ∃p'. p ↦α p'
  shows ∃p' q'. (HML_equivalent p' q') ∧ q ↦α q'
  ⟨proof⟩


lemma equiv_trans: transp HML_equivalent
  ⟨proof⟩
```

A formula distinguishes one state from another if its true for the first and false for the second.

```
abbreviation distinguishes ::  ‹('a, 's) hml ⇒ 's ⇒ 's ⇒ bool› where
   ‹distinguishes φ p q ≡ p ⊨ φ ∧ ¬ q ⊨ φ›

lemma hml_equiv_sym:
  shows ‹symp HML_equivalent›
⟨proof⟩
```

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

```
lemma hml_distinctions:
  fixes state::'s
  assumes ‹¬ HML_equivalent p q›
  shows ‹∃φ. distinguishes φ p q›
⟨proof⟩

end
end

context lts
begin
```

Introduce these definitions later?

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

abbreviation all_traces :: 'a list set where
all_traces ≡{tr. ∃p p'. p ↦$ tr p'}

inductive paths:: ‹'s ⇒ 's list ⇒ 's ⇒ bool› where
‹paths p [] p› |
‹paths p (a#as) p''› if ∃α. p ↦ α a ∧ (paths a as p'')

lemma path_implies_seq:
  assumes A1: ∃xs. paths p xs p'
  shows ∃ys. p ↦$ ys p'
⟨proof⟩

lemma seq_implies_path:
  assumes A1: ∃ys. p ↦$ ys p'
  shows ∃xs. paths p xs p'
⟨proof⟩
```

Trace preorder as inclusion of trace sets

```
definition trace_preordered (infix ‹≲T› 60)where
‹trace_preordered p q ≡ traces p ⊆ traces q›
```

Trace equivalence as mutual preorder

```
abbreviation trace_equivalent (infix ‹≃T› 60) where
‹p ≃T q ≡ p ≲T q ∧ q ≲T p›
```

Trace preorder is transitive

```
lemma trace_preorder_transitive:
  shows ‹transp (≲T)›
  ⟨proof⟩
```

```
lemma empty_trace_trivial:
  fixes p
  shows ‹[] ∈ traces p›
  ⟨proof⟩

lemma ‹equivp (≃T)›
⟨proof⟩
```

Failure Pairs

```
abbreviation failure_pairs :: ‹'s ⇒ ('a list × 'a set) set›
  where
‹failure_pairs p ≡ {(xs, F)|xs F. ∃p'. p ↦$ xs p' ∧ (initial_actions
p' ∩ F = {})}›
```

Failure preorder and -equivalence

```
definition failure_preordered (infix ‹≲F› 60) where
‹p ≲F q ≡ failure_pairs p ⊆ failure_pairs q›

abbreviation failure_equivalent (infix ‹≃F› 60) where
‹ p ≃F q ≡ p ≲F q ∧ q ≲F p›
```

Possible future sets

```
abbreviation possible_future_pairs :: ‹'s ⇒ ('a list × 'a list set) set›
  where
‹possible_future_pairs p ≡ {(xs, X)|xs X. ∃p'. p ↦$ xs p' ∧ traces p'
= X}›

definition possible_futures_preordered (infix ‹≲PF› 60) where
‹p ≲PF q ≡ (possible_future_pairs p ⊆ possible_future_pairs q)›

definition possible_futures_equivalent (infix ‹≃PF› 60) where
‹p ≃PF q ≡ (possible_future_pairs p = possible_future_pairs q)›

lemma PF_trans: transp (≃PF)
  ⟨proof⟩

lemma pf_implies_trace_preord:
  assumes ‹p ≲PF q›
  shows ‹p ≲T q›
  ⟨proof⟩
```

isomorphism

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

definition is_isomorphic :: ‹'s ⇒ 's ⇒ bool› (infix ‹≃ISO› 60) where
‹p ≃ISO q ≡ ∃f. isomorphism f ∧ (f p) = q›
```

Two states are simulation preordered if they can be related by a simulation relation. (Implied by isometry.)

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

definition simulated_by (infix ‹≲S› 60)
  where ‹p ≲S q ≡ ∃R. R p q ∧ simulation R›
```

Simulation preorder implies trace preorder

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

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

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

Bisimilarity is a simulation.

```
lemma bisim_sim:
  shows ‹simulation (≃B)›
  ⟨proof⟩


end
end

inductive TT_like :: ('a, 'i) hml ⇒ bool
  where
TT_like TT |
TT_like (hml_conj I J Φ) if (Φ `I) = {} (Φ ` J) = {}

inductive nested_empty_pos_conj :: ('a, 'i) hml ⇒ bool
  where
nested_empty_pos_conj TT |
nested_empty_pos_conj (hml_conj I J Φ)
if ∀x ∈ (Φ `I). nested_empty_pos_conj x (Φ ` J) = {}

inductive nested_empty_conj :: ('a, 'i) hml ⇒ bool
  where
nested_empty_conj TT |
nested_empty_conj (hml_conj I J Φ)
if ∀x ∈ (Φ `I). nested_empty_conj x ∀x ∈ (Φ `J). nested_empty_pos_conj
x

inductive stacked_pos_conj_pos :: ('a, 'i) hml ⇒ bool
  where
```

```
stacked_pos_conj_pos TT |
stacked_pos_conj_pos (hml_pos _ ψ) if nested_empty_pos_conj ψ |
stacked_pos_conj_pos (hml_conj I J Φ)
if ((∃φ ∈ (Φ ` I). ((stacked_pos_conj_pos φ) ∧
                     (∀ψ ∈ (Φ ` I). ψ ≠ φ ⟶ nested_empty_pos_conj
ψ))) ∨
   (∀ψ ∈ (Φ ` I). nested_empty_pos_conj ψ))
(Φ ` J) = {}

inductive stacked_pos_conj :: ('a, 'i) hml ⇒ bool
  where
stacked_pos_conj TT |
stacked_pos_conj (hml_pos _ ψ) if nested_empty_pos_conj ψ |
stacked_pos_conj (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). ((stacked_pos_conj φ) ∨ nested_empty_conj φ)
(∀ψ ∈ (Φ ` J). nested_empty_conj ψ)

inductive stacked_pos_conj_J_empty :: ('a, 'i) hml ⇒ bool
  where
stacked_pos_conj_J_empty TT |
stacked_pos_conj_J_empty (hml_pos _ ψ) if stacked_pos_conj_J_empty ψ
|
stacked_pos_conj_J_empty (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). (stacked_pos_conj_J_empty φ) Φ ` J = {}

inductive single_pos_pos :: ('a, 'i) hml ⇒ bool
  where
single_pos_pos TT |
single_pos_pos (hml_pos _ ψ) if nested_empty_pos_conj ψ |
single_pos_pos (hml_conj I J Φ) if
(∀φ ∈ (Φ `I). (single_pos_pos φ))
(Φ ` J) = {}

inductive single_pos :: ('a, 'i) hml ⇒ bool
  where
single_pos TT |
single_pos (hml_pos _ ψ) if nested_empty_conj ψ |
single_pos (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). (single_pos φ)
∀φ ∈ (Φ ` J). single_pos_pos φ

context lts begin

lemma index_sets_conj_disjunct:
  assumes I ∩ J ≠ {}
  shows ∀s. ¬ (s ⊨ (hml_conj I J Φ))
⟨proof⟩

lemma HML_true_TT_like:
```

```
  assumes TT_like φ
  shows HML_true φ
  ⟨proof⟩

lemma HML_true_nested_empty_pos_conj:
  assumes nested_empty_pos_conj φ
  shows HML_true φ
  ⟨proof⟩

end
```

```
inductive HML_trace :: ('a, 's)hml ⇒ bool
  where
trace_tt : HML_trace TT |
trace_conj: HML_trace (hml_conj {} {} ψs)|
trace_pos: HML_trace (hml_pos α φ) if HML_trace φ

definition HML_trace_formulas where
HML_trace_formulas ≡ {φ. HML_trace φ}
```

translation of a trace to a formula

```
fun trace_to_formula :: 'a list ⇒ ('a, 's)hml
  where
trace_to_formula [] = TT |
trace_to_formula (a#xs) = hml_pos a (trace_to_formula xs)

inductive HML_failure :: ('a, 's)hml ⇒ bool
  where
failure_tt: HML_failure TT |
failure_pos: HML_failure (hml_pos α φ) if HML_failure φ |
failure_conj: HML_failure (hml_conj I J ψs)
if (∀i ∈ I. TT_like (ψs i)) ∧ (∀j ∈ J. (TT_like (ψs j)) ∨ (∃α χ. ((ψs
j) = hml_pos α χ ∧ (TT_like χ))))

inductive HML_simulation :: ('a, 's)hml ⇒ bool
  where
sim_tt: HML_simulation TT |
sim_pos: HML_simulation (hml_pos α φ) if HML_simulation φ|
sim_conj: HML_simulation (hml_conj I J ψs)
if (∀x ∈ (ψs ` I). HML_simulation x) ∧ (ψs ` J = {})

definition HML_simulation_formulas where
HML_simulation_formulas ≡ {φ. HML_simulation φ}

inductive HML_readiness :: ('a, 's)hml ⇒ bool
  where
read_tt: HML_readiness TT |
```

```
read_pos: HML_readiness (hml_pos α φ) if HML_readiness φ|
read_conj: HML_readiness (hml_conj I J Φ)
if (∀x ∈ (Φ ` (I ∪ J)). TT_like x ∨ (∃α χ. x = hml_pos α χ ∧ TT_like
χ))

inductive HML_impossible_futures ::  ('a, 's)hml ⇒ bool
  where
  if_tt: HML_impossible_futures TT |
  if_pos: HML_impossible_futures (hml_pos α φ) if HML_impossible_futures
φ |
if_conj: HML_impossible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` I). TT_like x ∀x ∈ (Φ ` J). (HML_trace x)

inductive HML_possible_futures :: ('a, 's)hml ⇒ bool
  where
pf_tt: HML_possible_futures TT |
pf_pos: HML_possible_futures (hml_pos α φ) if HML_possible_futures φ
|
pf_conj: HML_possible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I∪ J)). (HML_trace x)

definition HML_possible_futures_formulas where
HML_possible_futures_formulas ≡ {φ. HML_possible_futures φ}

inductive HML_failure_trace :: ('a, 's)hml ⇒ bool
  where
f_trace_tt: HML_failure_trace TT |
f_trace_pos: HML_failure_trace (hml_pos α φ) if HML_failure_trace φ|
f_trace_conj: HML_failure_trace (hml_conj I J Φ)
if ((∃ψ ∈ (Φ ` I). (HML_failure_trace ψ) ∧ (∀y ∈ (Φ ` I). ψ ≠ y ⟶
nested_empty_conj y)) ∨
(∀y ∈ (Φ ` I). nested_empty_conj y)) ∧
(∀y ∈ (Φ ` J). stacked_pos_conj_pos y)


inductive HML_ready_trace :: ('a, 's)hml ⇒ bool
  where
r_trace_tt: HML_ready_trace TT |
r_trace_pos: HML_ready_trace (hml_pos α φ) if HML_ready_trace φ|
r_trace_conj: HML_ready_trace (hml_conj I J Φ)
if (∃x ∈ (Φ ` I). HML_ready_trace x ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ single_pos
y))
∨ (∀y ∈ (Φ ` I).single_pos y)
(∀y ∈ (Φ ` J). single_pos_pos y)

inductive HML_ready_sim :: ('a, 's) hml ⇒ bool
  where
HML_ready_sim TT |
HML_ready_sim (hml_pos α φ) if HML_ready_sim φ |
```

```
HML_ready_sim (hml_conj I J Φ) if
(∀x ∈ (Φ ` I). HML_ready_sim x) ∧ (∀y ∈ (Φ ` J). single_pos_pos y)

inductive HML_2_nested_sim :: ('a, 's) hml ⇒ bool
  where
HML_2_nested_sim TT |
HML_2_nested_sim (hml_pos α φ) if HML_2_nested_sim φ |
HML_2_nested_sim (hml_conj I J Φ)
if (∀x ∈ (Φ ` I). HML_2_nested_sim x) ∧ (∀y ∈ (Φ ` J). HML_simulation
y)

inductive HML_revivals :: ('a, 's) hml ⇒ bool
  where
revivals_tt: HML_revivals TT |
revivals_pos: HML_revivals (hml_pos α φ) if HML_revivals φ |
revivals_conj: HML_revivals (hml_conj I J Φ) if (∀x ∈ (Φ ` I). ∃α χ.
(x = hml_pos α χ) ∧ TT_like χ)
(∀x ∈ (Φ ` J). ∃α χ. (x = hml_pos α χ) ∧ TT_like χ)

end
theory HML_definitions
imports HML_list
begin

inductive hml_trace :: ('a, 's)hml ⇒ bool where
hml_trace TT |
hml_trace (hml_pos α φ) if hml_trace φ

inductive hml_failure :: ('a, 's)hml ⇒ bool
  where
failure_tt: hml_failure TT |
failure_pos: hml_failure (hml_pos α φ) if hml_failure φ |
failure_conj: hml_failure (hml_conj I J ψs)
if I = {} (∀j ∈ J. (∃α. ((ψs j) = hml_pos α TT)) ∨ ψs j = TT)

inductive hml_readiness :: ('a, 's)hml ⇒ bool
  where
read_tt: hml_readiness TT |
read_pos: hml_readiness (hml_pos α φ) if hml_readiness φ|
read_conj: hml_readiness (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I ∪ J)). (∃α. x = (hml_pos α TT::('a, 's)hml)) ∨ x = TT

inductive hml_impossible_futures ::  ('a, 's)hml ⇒ bool
  where
  if_tt: hml_impossible_futures TT |
  if_pos: hml_impossible_futures (hml_pos α φ) if hml_impossible_futures
φ |
if_conj: hml_impossible_futures (hml_conj I J Φ)
if I = {} ∀x ∈ (Φ ` J). (hml_trace x)
```

```
inductive hml_possible_futures :: ('a, 's)hml ⇒ bool
  where
pf_tt: hml_possible_futures TT |
pf_pos: hml_possible_futures (hml_pos α φ) if hml_possible_futures φ
|
pf_conj: hml_possible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I∪ J)). (hml_trace x)

definition hml_possible_futures_formulas where
hml_possible_futures_formulas ≡ {φ. hml_possible_futures φ}

inductive hml_failure_trace :: ('a, 's)hml ⇒ bool where
hml_failure_trace TT |
hml_failure_trace (hml_pos α φ) if hml_failure_trace φ |
hml_failure_trace (hml_conj I J Φ)
  if (Φ ` I) = {} ∨ (∃i ∈ Φ ` I. Φ ` I = {i} ∧ hml_failure_trace i)
     ∀j ∈ Φ ` J. ∃α. j = (hml_pos α TT) ∨ j = TT

inductive hml_ready_trace :: ('a, 's)hml ⇒ bool
  where
r_trace_tt: hml_ready_trace TT |
r_trace_pos: hml_ready_trace (hml_pos α φ) if hml_ready_trace φ|
r_trace_conj: hml_ready_trace (hml_conj I J Φ)
if (∃x ∈ (Φ ` I). hml_ready_trace x ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ (∃α.
y = (hml_pos α TT))))
∨ (∀y ∈ (Φ ` I).(∃α. y = (hml_pos α TT)))
(∀y ∈ (Φ ` J). (∃α. y = (hml_pos α TT)))

inductive hml_ready_sim :: ('a, 's) hml ⇒ bool
  where
hml_ready_sim TT |
hml_ready_sim (hml_pos α φ) if hml_ready_sim φ |
hml_ready_sim (hml_conj I J Φ) if
(∀x ∈ (Φ ` I). hml_ready_sim x) ∧ (∀y ∈ (Φ ` J). (∃α. y = (hml_pos
α TT)))

inductive hml_2_nested_sim :: ('a, 's) hml ⇒ bool
  where
hml_2_nested_sim TT |
hml_2_nested_sim (hml_pos α φ) if hml_2_nested_sim φ |
hml_2_nested_sim (hml_conj I J Φ)
if (∀x ∈ (Φ ` I). hml_2_nested_sim x) ∧ (∀y ∈ (Φ ` J). HML_simulation
y)

context lts begin

lemma alt_trace_def_implies_trace_def:
  fixes φ :: ('a, 's) hml
```

```
  assumes hml_trace φ
  shows ∃ψ. HML_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma trace_def_implies_alt_trace_def:
  fixes φ :: ('a, 's) hml
  assumes HML_trace φ
  shows ∃ψ. hml_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma trace_definitions_equivalent:
  ∀φ. (HML_trace φ ⟶ (∃ψ. hml_trace ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_trace φ ⟶ (∃ψ. HML_trace ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩

lemma alt_failure_def_implies_failure_def:
  fixes φ :: ('a, 's) hml
  assumes hml_failure φ
  shows ∃ψ. HML_failure ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma failure_def_implies_alt_failure_def:
  fixes φ :: ('a, 's) hml
  assumes HML_failure φ
  shows ∃ψ. hml_failure ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma failure_definitions_equivalent:
  ∀φ. (HML_failure φ ⟶ (∃ψ. hml_failure ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_failure φ ⟶ (∃ψ. HML_failure ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩

lemma alt_readiness_def_implies_readiness_def:
  fixes φ :: ('a, 's) hml
  assumes hml_readiness φ
  shows ∃ψ. HML_readiness ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma readiness_def_implies_alt_readiness_def:
  fixes φ :: ('a, 's) hml
  assumes HML_readiness φ
  shows ∃ψ. hml_readiness ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma readiness_definitions_equivalent:
  ∀φ. (HML_readiness φ ⟶ (∃ψ. hml_readiness ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_readiness φ ⟶ (∃ψ. HML_readiness ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩
```

```
lemma alt_impossible_futures_def_implies_impossible_futures_def:
  fixes φ :: ('a, 's) hml
  assumes hml_impossible_futures φ
  shows ∃ψ. HML_impossible_futures ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma impossible_futures_def_implies_alt_impossible_futures_def:
  fixes φ :: ('a, 's) hml
  assumes HML_impossible_futures φ
  shows ∃ψ. hml_impossible_futures ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma alt_failure_trace_def_implies_failure_trace_def:
  fixes φ :: ('a, 's) hml
  assumes hml_failure_trace φ
  shows ∃ψ. HML_failure_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma stacked_pos_rewriting:
  assumes stacked_pos_conj_pos φ ¬HML_true φ
  shows ∃α. (∀s. (s ⊨ φ) ⟷ (s ⊨ (hml_pos α TT)))
  ⟨proof⟩

lemma nested_empty_conj_TT_or_FF:
  assumes nested_empty_conj φ
  shows (∀s. (s ⊨ φ)) ∨ (∀s. ¬(s ⊨ φ))
  ⟨proof⟩

lemma failure_trace_def_implies_alt_failure_trace_def:
  fixes φ :: ('a, 's) hml
  assumes HML_failure_trace φ
  shows ∃ψ. hml_failure_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

end
end
theory HML_equivalences
imports Main
HML_list HML_definitions
begin

context lts begin

definition HML_trace_equivalent where
HML_trace_equivalent p q ≡ (∀ φ. φ ∈ HML_trace_formulas ⟶ (p ⊨ φ)
⟷ (q ⊨ φ))

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

```
(∀φ. φ ∈ HML_simulation_formulas ⟶ (p ⊨ φ ⟷ q ⊨ φ))

definition HML_possible_futures_equivalent where
HML_possible_futures_equivalent p q ≡ (∀ φ. φ ∈ HML_possible_futures_formulas
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))

definition hml_possible_futures_equivalent where
hml_possible_futures_equivalent p q ≡ (∀ φ. φ ∈ hml_possible_futures_formulas
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))

end
end
```

2.4 Price Spectra of Behavioral Equivalences

The linear-time–branching-time spectrum can be represented in terms of HML-expressiveness (s.h. section HML). (Deciding all at once)(energy games) show how one can think of the amount of HML-expressiveness used by a formula by its *price*. The equivalences of the spectrum (or their modal-logical characterizations) can then be defined in terms of *price coordinates*, that is equivalence X is characterized by the HML formulas with prices less then or equal to a *X-price bound* e_X. We use the six dimensions from (energy games) to characterize the notions of equivalence we are interested in (In figure xx oder so umschreiben). Intuitively, the dimensions can be described as follows:
1. Observations
... ...

Formula Prices

The *expressiveness price* $\mathrm{expr} : \mathrm{HML}[\Sigma] \rightarrow (\mathbb{N} \cup \{\infty\})^6$ of a formula is defined recursively, similar to energy games: TODO: expr function

Remark: Infinity is included in our definition, due to infinite branching conjunctions. Supremum over infinite set wird zu unendlich.

To better argue about the function we define each dimension as a seperate function.

Vlt als erstes: modaltiefe als beispiel für observation expressiveness von formel, mit isabelle definition, dann pos_r definition, direct_expr definition, einzelne dimensionen, lemma direct_expr = expr...

```
primrec expr_1 :: ('a, 's)hml ⇒ enat
  where
expr_1_tt: ‹expr_1 TT = 0› |
expr_1_conj: ‹expr_1 (hml_conj I J Φ) = Sup ((expr_1 ∘ Φ) ` I ∪ (expr_1
∘ Φ) ` J)› |
expr_1_pos: ‹expr_1 (hml_pos α φ) =
  1 + (expr_1 φ)›

fun pos_r :: ('a, 's)hml set ⇒ ('a, 's)hml set
  where
pos_r xs = (
let max_val = (Sup (expr_1 ` xs));
  max_elem = (SOME ψ. ψ ∈ xs ∧ expr_1 ψ = max_val);
  xs_new = xs - {max_elem}
in xs_new)

lemma pos_r_subs: pos_r (Φ ` I) ⊆ (Φ ` I)
  ⟨proof⟩

function direct_expr :: ('a, 's)hml ⇒ enat × enat × enat × enat × enat
× enat where
  direct_expr TT = (0, 1, 0, 0, 0, 0) |
  direct_expr (hml_pos α φ) = (1 + fst (direct_expr φ),
                              fst (snd (direct_expr φ)),
                              fst (snd (snd (direct_expr φ))),
                              fst (snd (snd (snd (direct_expr φ)))),

                              fst (snd (snd (snd (snd (direct_expr φ))))),

                              snd (snd (snd (snd (snd (direct_expr φ))))))
|
  direct_expr (hml_conj I J Φ) = (Sup ((fst ∘ direct_expr ∘ Φ) ` I ∪
(fst ∘ direct_expr ∘ Φ) ` J),
                                  1 + Sup ((fst ∘ snd ∘ direct_expr
∘ Φ) ` I ∪ (fst ∘ snd ∘ direct_expr ∘ Φ) ` J),
(Sup ((fst ∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd ∘ snd ∘ direct_expr ∘ Φ)
` I ∪ (fst ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J)),
(Sup (((fst ∘ direct_expr) ` (pos_r (Φ ` I))) ∪ (fst ∘ snd ∘ snd ∘ snd
∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) `
J)),
(Sup ((fst ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd
∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J ∪ (fst ∘ direct_expr ∘ Φ) `
J)),
(Sup ((snd ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` I ∪ ((eSuc ∘ snd
∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J))))
  ⟨proof⟩

inductive_set HML_wf_rel :: (('a, 's)hml) rel where
φ = Φ i ∧ i ∈ (I ∪ J) ⟹ (φ, (hml_conj I J Φ)) ∈ HML_wf_rel |
```

```
(φ, (hml_pos α φ)) ∈ HML_wf_rel

lemma HML_wf_rel_is_wf: ‹wf HML_wf_rel›
  ⟨proof⟩

termination
  ⟨proof⟩

primrec expr_2 :: ('a, 's)hml ⇒ enat
  where
expr_2_tt: ‹expr_2 TT = 1› |
expr_2_conj: ‹expr_2 (hml_conj I J Φ) = 1 + Sup ((expr_2 ∘ Φ) ` I ∪ (expr_2
∘ Φ) ` J)› |
expr_2_pos: ‹expr_2 (hml_pos α φ) = expr_2 φ›


primrec expr_3 :: ('a, 's) hml ⇒ enat
  where
expr_3_tt: ‹expr_3 TT = 0› |
 expr_3_pos: ‹expr_3 (hml_pos α φ) = expr_3 φ› |
expr_3_conj: ‹expr_3 (hml_conj I J Φ) = (Sup ((expr_1 ∘ Φ) ` I ∪ (expr_3
∘ Φ) ` I ∪ (expr_3 ∘ Φ) ` J))›

primrec expr_4 :: ('a, 's)hml ⇒ enat
  where
expr_4_tt: expr_4 TT = 0 |
expr_4_pos: expr_4 (hml_pos a φ) = expr_4 φ |
expr_4_conj: expr_4 (hml_conj I J Φ) = Sup ((expr_1 ` (pos_r (Φ ` I)))
∪ (expr_4 ∘ Φ) ` I ∪ (expr_4 ∘ Φ) ` J)

primrec expr_5 :: ('a, 's)hml ⇒ enat
  where
expr_5_tt: ‹expr_5 TT = 0› |
expr_5_pos:‹expr_5 (hml_pos α φ) = expr_5 φ›|
expr_5_conj: ‹expr_5 (hml_conj I J Φ) =
(Sup ((expr_5 ∘ Φ) ` I ∪ (expr_5 ∘ Φ) ` J ∪ (expr_1 ∘ Φ) ` J))›

primrec expr_6 :: ('a, 's)hml ⇒ enat
  where
expr_6_tt: ‹expr_6 TT = 0› |
expr_6_pos: ‹expr_6 (hml_pos α φ) = expr_6 φ›|
expr_6_conj: ‹expr_6 (hml_conj I J Φ) =
(Sup ((expr_6 ∘ Φ) ` I ∪ ((eSuc ∘ expr_6 ∘ Φ) ` J)))›


fun expr :: ('a, 's)hml ⇒ enat × enat × enat ×  enat × enat × enat

  where
‹expr φ = (expr_1 φ, expr_2 φ, expr_3 φ, expr_4 φ, expr_5 φ, expr_6 φ)›
```

```
lemma apply_comp:
  assumes ∀x ∈ Φ ` I. F x = G1 (G2 (G3 (G4 (G5 ((H x))))))
  shows Sup ((F ∘ Φ) ` I) = Sup((G1 ∘ G2 ∘ G3 ∘ G4 ∘ G5 ∘ H ∘ Φ) ` I)
  ⟨proof⟩

lemma fa_set_eq:
  assumes ∀x ∈ Φ ` I. F x = G (H x)
  shows ((F ∘ Φ) ` I) = (G ∘ H ∘ Φ) ` I
  ⟨proof⟩

lemma expr_6_direct_expr_eq:
  assumes ⋀x. x ∈ Φ ` I ⟹ expr_6 x = snd (snd (snd (snd (snd (direct_expr
x)))))
  shows (expr_6 ∘ Φ) ` I = (snd ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr
∘ Φ) ` I
⟨proof⟩

lemma expr_6_eSuc_eq:
  assumes ⋀x. x ∈ Φ ` J ⟹ eSuc (expr_6 x) = eSuc (snd (snd (snd (snd
(snd (direct_expr x))))))
  shows ((eSuc ∘ expr_6 ∘ Φ) ` J) = ((eSuc ∘ snd ∘ snd ∘ snd ∘ snd ∘ snd
∘ direct_expr ∘ Φ) ` J)
⟨proof⟩

lemma expr_5_dir_eq:
  assumes ∀x ∈ Φ ` (I ∪ J). expr_5 x = fst (snd (snd (snd (snd (direct_expr
x)))))
  shows ((expr_5 ∘ Φ) ` I ∪ (expr_5 ∘ Φ) ` J) =
          ((fst ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` I ∪ (fst
∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J)
⟨proof⟩

thm comp_apply

lemma
  shows expr φ = direct_expr φ
⟨proof⟩
```