Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

15th 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 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).

The term *reactive system* (citation) describes computing systems that continously 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.

The semantics of reactive systems can be modeled as equivalences, that determine wether 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 Labelled Transition Systems

Definition 1 (Labeled transition Systems)

A Labelled Transition System (LTS) is a tuple $\mathcal{S} = (\mathit{Proc}, \mathit{Act}, \rightarrow)$ where Proc is the set of processes, Act is the set of actions and $\rightarrow \subseteq \mathit{Proc} \times \mathit{Act} \times \mathit{Proc}$ is a transition relation.

In concurrency theory, it is customary that the semantics of Reactive Systems are given in terms of labelled transition systems. The processes represent the states a reactive system can be in. A transition of one state into another, caused by performing an action, can be understood as moving along the corresponding edge in the transition relation.

- Example?

- examples (to reuse later?)???

Some more Definitions

The α-derivatives of a process are the processes that can be reached with one α-transition:

- image finite? nötig?
- image countable? - initial actions - deadlock - relevant actions? - step sequence! -

Isabelle

Zustände: `'s` und Aktionen `'a`, Transitionsrelation ist locale trans. Ein LTS wird dann durch seine Transitionsrelation definiert.

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

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

Transition System is image-finite

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

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

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

nötig?

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

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 T_trans:
  shows ‹transp (≲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 list)
set›
  where
‹possible_future_pairs p ≡ {(xs, X)|xs X. ∃p'. p ↦$ xs p' ∧ traces p'
= (set X)}›

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⟩
```

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›
```

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
```

Hennessy–Milner logic

Hennessy–Milner logic, first introduced by Matthew Hennessy and Robin Milner (citation), 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:

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

(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 (...)

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

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

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[1]

```
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).

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

```
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

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))))›

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

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⟩

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⟩
```

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



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
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))

end
end
theory formula_prices_list
  imports
    Main
    HML_list
    HOL-Library.Extended_Nat
begin

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)›


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))›

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)

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)›

end
```