

# Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

9th February 2024



# Contents

<b>Contents</b>	<b>3</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Foundations</b>	<b>7</b>
2.1 Labelled Transition Systems . . . . .	7



# 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 continuously interact with their environment. Unlike sequential systems, the behavior of 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 whether or not two systems behave similarly. In the literature on concurrent systems many different notions 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 hierarchy of equivalences. -> New Paper characterizes them differently... (HML beschreibung als erstes?!!)

- Reactive Systems
- modelling (via LTS etc)
- Semantics of resysts
- Verification
- different notions of equivalence (because of nondeterminism?) -> 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 of 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  $\mathcal{S} = (Proc, Act, \rightarrow)$  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.

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  $\alpha$ -derivatives of a process are the processes that can be reached with one  $\alpha$ -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  $\Rightarrow$  'a list set> where
<traces p  $\equiv$  {tr.  $\exists p'$ . p  $\mapsto$  $ tr p'}>
```

```
abbreviation all_traces :: 'a list set where
all_traces  $\equiv$  {tr.  $\exists p p'$ . p  $\mapsto$  $ tr p'}
```

```
inductive paths :: <'s  $\Rightarrow$  's list  $\Rightarrow$  's  $\Rightarrow$  bool> where
<paths p [] p> |
<paths p (a#as) p'> if  $\exists \alpha$ . p  $\mapsto$   $\alpha$  a  $\wedge$  (paths a as p')
```

```
lemma path_implies_seq:
  assumes A1:  $\exists xs$ . paths p xs p'
  shows  $\exists ys$ . p  $\mapsto$  $ ys p'
<proof>
```

```
lemma seq_implies_path:
  assumes A1:  $\exists ys$ . p  $\mapsto$  $ ys p'
  shows  $\exists xs$ . paths p xs p'
<proof>
```

Trace preorder as inclusion of trace sets

```
definition trace_preordered (infix <math>\lesssim^T> 60) where
<trace_preordered p q  $\equiv$  traces p  $\subseteq$  traces q>
```

Trace equivalence as mutual preorder

```
abbreviation trace_equivalent (infix <math>\simeq^T> 60) where
<p  $\simeq^T$  q  $\equiv$  p  $\lesssim^T$  q  $\wedge$  q  $\lesssim^T$  p>
```

Trace preorder is transitive

```
lemma T_trans:
  shows <transp ( $\lesssim^T$ )>
<proof>
```

Failure Pairs

```
abbreviation failure_pairs :: <'s  $\Rightarrow$  ('a list  $\times$  'a set) set>
  where
<failure_pairs p  $\equiv$  {(xs, F) | xs F.  $\exists p'$ . p  $\mapsto$  $ xs p'  $\wedge$  (initial_actions
p'  $\cap$  F = {})}>
```

Failure preorder and -equivalence

```
definition failure_preordered (infix <math>\lesssim^F> 60) where
<p  $\lesssim^F$  q  $\equiv$  failure_pairs p  $\subseteq$  failure_pairs q>
```

**abbreviation** failure\_equivalent (**infix**  $\simeq_F$  60) **where**  
 $\langle p \simeq_F q \equiv p \lesssim_F q \wedge q \lesssim_F p \rangle$

Possible future sets

**abbreviation** possible\_future\_pairs ::  $\langle 's \Rightarrow ('a \text{ list} \times 'a \text{ list list}) \text{ set} \rangle$   
**where**  
 $\langle \text{possible\_future\_pairs } p \equiv \{(xs, X) \mid xs \text{ X. } \exists p'. p \mapsto \$ xs \ p' \wedge \text{traces } p' = (\text{set } X)\} \rangle$

**definition** possible\_futures\_equivalent (**infix**  $\simeq_{PF}$  60) **where**  
 $\langle p \simeq_{PF} q \equiv (\text{possible\_future\_pairs } p = \text{possible\_future\_pairs } q) \rangle$

**lemma** PF\_trans: transp ( $\simeq_{PF}$ )  
 $\langle \text{proof} \rangle$

isomorphism

**definition** isomorphism ::  $\langle ('s \Rightarrow 's) \Rightarrow \text{bool} \rangle$  **where**  
 $\langle \text{isomorphism } f \equiv \text{bij } f \wedge (\forall p \ a \ p'. p \mapsto a \ p' \longleftrightarrow f \ p \mapsto a \ (f \ p')) \rangle$

**definition** is\_isomorphic ::  $\langle 's \Rightarrow 's \Rightarrow \text{bool} \rangle$  (**infix**  $\simeq_{ISO}$  60) **where**  
 $\langle p \simeq_{ISO} q \equiv \exists f. \text{isomorphism } f \wedge (f \ p) = q \rangle$

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

**definition** simulation  
**where**  $\langle \text{simulation } R \equiv$   
 $\forall p \ q \ a \ p'. p \mapsto a \ p' \wedge R \ p \ q \longrightarrow (\exists q'. q \mapsto a \ q' \wedge R \ p' \ q') \rangle$

**definition** simulated\_by (**infix**  $\lesssim_S$  60)  
**where**  $\langle p \lesssim_S q \equiv \exists R. R \ p \ q \wedge \text{simulation } R \rangle$

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

**definition** bisimilar (**infix**  $\simeq_B$  80) **where**  
 $\langle p \simeq_B q \equiv \exists R. \text{simulation } R \wedge \text{symp } R \wedge R \ p \ q \rangle$

Bisimilarity is a simulation.

**lemma** bisim\_sim:  
**shows**  $\langle \text{simulation } (\simeq_B) \rangle$   
 $\langle \text{proof} \rangle$

**end**  
**end**  
**theory** HML\_list  
**imports**  
Main

```

Transition_Systems

begin
datatype ('a, 'i)hml =
TT |
hml_pos <'a> <('a, 'i)hml> |
hml_conj 'i set 'i set 'i  $\Rightarrow$  ('a, 'i) hml

inductive TT_like :: ('a, 'i) hml  $\Rightarrow$  bool
  where
TT_like TT |
TT_like (hml_conj I J  $\Phi$ ) if ( $\Phi \setminus I$ ) = {} ( $\Phi \setminus J$ ) = {}

inductive nested_empty_pos_conj :: ('a, 'i) hml  $\Rightarrow$  bool
  where
nested_empty_pos_conj TT |
nested_empty_pos_conj (hml_conj I J  $\Phi$ )
if  $\forall x \in (\Phi \setminus I). \text{nested\_empty\_pos\_conj } x$  ( $\Phi \setminus J$ ) = {}

inductive nested_empty_conj :: ('a, 'i) hml  $\Rightarrow$  bool
  where
nested_empty_conj TT |
nested_empty_conj (hml_conj I J  $\Phi$ )
if  $\forall x \in (\Phi \setminus I). \text{nested\_empty\_conj } x \forall x \in (\Phi \setminus J). \text{nested\_empty\_pos\_conj } x$ 

inductive stacked_pos_conj_pos :: ('a, 'i) hml  $\Rightarrow$  bool
  where
stacked_pos_conj_pos TT |
stacked_pos_conj_pos (hml_pos _  $\psi$ ) if nested_empty_pos_conj  $\psi$  |
stacked_pos_conj_pos (hml_conj I J  $\Phi$ )
if ( $\exists \varphi \in (\Phi \setminus I). ((\text{stacked\_pos\_conj\_pos } \varphi) \wedge$ 
 $(\forall \psi \in (\Phi \setminus I). \psi \neq \varphi \longrightarrow \text{nested\_empty\_pos\_conj } \psi))) \vee$ 
 $(\forall \psi \in (\Phi \setminus I). \text{nested\_empty\_pos\_conj } \psi))$ 
( $\Phi \setminus J$ ) = {}

inductive stacked_pos_conj :: ('a, 'i) hml  $\Rightarrow$  bool
  where
stacked_pos_conj TT |
stacked_pos_conj (hml_pos _  $\psi$ ) if nested_empty_pos_conj  $\psi$  |
stacked_pos_conj (hml_conj I J  $\Phi$ )
if  $\forall \varphi \in (\Phi \setminus I). ((\text{stacked\_pos\_conj } \varphi) \vee \text{nested\_empty\_conj } \varphi)$ 
 $(\forall \psi \in (\Phi \setminus J). \text{nested\_empty\_conj } \psi)$ 

```

```

inductive stacked_pos_conj_J_empty :: ('a, 'i) hml  $\Rightarrow$  bool
  where
    stacked_pos_conj_J_empty TT |
    stacked_pos_conj_J_empty (hml_pos _  $\psi$ ) if stacked_pos_conj_J_empty  $\psi$  |
    stacked_pos_conj_J_empty (hml_conj I J  $\Phi$ )
if  $\forall \varphi \in (\Phi \setminus I). (\text{stacked\_pos\_conj\_J\_empty } \varphi) \ \Phi \setminus J = \{\}$ 

```

```

inductive single_pos_pos :: ('a, 'i) hml  $\Rightarrow$  bool
  where
    single_pos_pos TT |
    single_pos_pos (hml_pos _  $\psi$ ) if nested_empty_pos_conj  $\psi$  |
    single_pos_pos (hml_conj I J  $\Phi$ ) if
      ( $\forall \varphi \in (\Phi \setminus I). (\text{single\_pos\_pos } \varphi)$ )
      ( $\Phi \setminus J = \{\}$ )

```

```

inductive single_pos :: ('a, 'i) hml  $\Rightarrow$  bool
  where
    single_pos TT |
    single_pos (hml_pos _  $\psi$ ) if nested_empty_conj  $\psi$  |
    single_pos (hml_conj I J  $\Phi$ )
if  $\forall \varphi \in (\Phi \setminus I). (\text{single\_pos } \varphi)$ 
 $\forall \varphi \in (\Phi \setminus J). \text{single\_pos\_pos } \varphi$ 

```

```

primrec flatten :: ('a, 'i) hml  $\Rightarrow$  ('a, 'i) hml where
  flatten TT = TT |
  flatten (hml_pos  $\alpha$   $\psi$ ) = (hml_pos  $\alpha$  (flatten  $\psi$ )) |
  flatten (hml_conj I J  $\Phi$ ) = (hml_conj I J  $\Phi$ )

```

```

inductive flattened :: ('a, 'i) hml  $\Rightarrow$  bool
  where
    flattened TT |
    flattened (hml_pos _  $\psi$ ) if flattened  $\psi$  |
    flattened (hml_conj I J  $\Phi$ )
if  $\forall x \in (\Phi \setminus I). (\exists \alpha \psi. x = (\text{hml\_pos } \alpha \psi) \wedge \text{flattened } \psi)$ 
 $\forall y \in (\Phi \setminus J). (\exists \alpha \psi. y = (\text{hml\_pos } \alpha \psi) \wedge \text{flattened } \psi) \vee$ 
 $(\exists I' J' \Phi'. y = (\text{hml\_conj } I' J' \Phi') \wedge (\forall x \in \Phi' \setminus I'. (\exists \alpha \psi. x = (\text{hml\_pos } \alpha \psi) \wedge \text{flattened } \psi)))$ 
 $\wedge (\Phi' \setminus J' = \{\})$ 

```

```

context lts begin

primrec hml_semantics :: '<'s  $\Rightarrow$  ('a, 's)hml  $\Rightarrow$  bool>
  (<_  $\models$  _> [50, 50] 50)
where
  hml_sem_tt: <(_  $\models$  TT) = True> |
  hml_sem_pos: <(p  $\models$  (hml_pos  $\alpha$   $\varphi$ )) = ( $\exists$  q. (p  $\mapsto \alpha$  q)  $\wedge$  q  $\models$   $\varphi$ )> |
  hml_sem_conj: <(p  $\models$  (hml_conj I J  $\psi$ s)) = (( $\forall$  i  $\in$  I. p  $\models$  ( $\psi$ s i))  $\wedge$  ( $\forall$  j
 $\in$  J.  $\neg$ (p  $\models$  ( $\psi$ s j))))>

lemma index_sets_conj_disjunct:
  assumes I  $\cap$  J  $\neq$  {}
  shows  $\forall$ s.  $\neg$  (s  $\models$  (hml_conj I J  $\Phi$ ))
<proof>

definition HML_true where
HML_true  $\varphi \equiv \forall$ s. s  $\models$   $\varphi$ 

lemma
  fixes s::'s
  assumes HML_true (hml_conj I J  $\Phi$ )
  shows  $\forall \varphi \in \Phi \setminus I$ . HML_true  $\varphi$ 
<proof>

lemma HML_true_TT_like:
  assumes TT_like  $\varphi$ 
  shows HML_true  $\varphi$ 
<proof>

lemma HML_true_nested_empty_pos_conj:
  assumes nested_empty_pos_conj  $\varphi$ 
  shows HML_true  $\varphi$ 
<proof>

```

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

```

definition HML_equivalent :: '<'s  $\Rightarrow$  's  $\Rightarrow$  bool> where
  <HML_equivalent p q  $\equiv$  ( $\forall$   $\varphi$ ::('a, 's) hml. (p  $\models$   $\varphi$ )  $\longleftrightarrow$  (q  $\models$   $\varphi$ ))>

```

An HML formula  $\varphi_1$  implies another ( $\varphi_r$ ) if the fact that some process  $p$  satisfies  $\varphi_1$  implies that  $p$  must also satisfy  $\varphi_r$ , no matter the process  $p$ .

```

definition hml_impl :: ('a, 's) hml  $\Rightarrow$  ('a, 's) hml  $\Rightarrow$  bool (infix  $\Rightarrow$ 
60) where
   $\varphi_1 \Rightarrow \varphi_r \equiv (\forall p. (p \models \varphi_1) \longrightarrow (p \models \varphi_r))$ 

```

```

lemma hml_impl_iffI:  $\varphi_1 \Rightarrow \varphi_r = (\forall p. (p \models \varphi_1) \longrightarrow (p \models \varphi_r))$ 
<proof>

```

A HML formula  $\varphi_l$  is said to be equivalent to some other HML formula  $\varphi_r$  (written  $\varphi_l \Leftrightarrow \varphi_r$ ) iff process  $p$  satisfies  $\varphi_l$  iff it also satisfies  $\varphi_r$ , 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  $\Rightarrow$  ('a, 's) hml  $\Rightarrow$  bool (infix
 $\Leftrightarrow$  60) where
   $\varphi_l \Leftrightarrow \varphi_r \equiv \varphi_l \Rightarrow \varphi_r \wedge \varphi_r \Rightarrow \varphi_l$ 
```

$\Leftrightarrow$  is truly an equivalence relation.

```
lemma hml_eq_equiv: equivp ( $\Leftrightarrow$ )
  <proof>
```

```
lemma equiv_der:
  assumes HML_equivalent p q  $\exists p'. p \mapsto_\alpha p'$ 
  shows  $\exists p' q'. (\text{HML\_equivalent } p' q') \wedge q \mapsto_\alpha q'$ 
  <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  $\Rightarrow$  's  $\Rightarrow$  's  $\Rightarrow$  bool> where
  <distinguishes  $\varphi$  p q  $\equiv p \models \varphi \wedge \neg q \models \varphi$ >
```

```
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 < $\neg$  HML_equivalent p q>
  shows < $\exists \varphi. \text{distinguishes } \varphi$  p q>
  <proof>

end
```

```
inductive HML_trace :: ('a, 's) hml  $\Rightarrow$  bool
  where
```

```

trace_tt : HML_trace TT |
trace_conj: HML_trace (hml_conj {} {}  $\psi$ s) |
trace_pos: HML_trace (hml_pos  $\alpha$   $\varphi$ ) if HML_trace  $\varphi$ 

```

**definition** HML\_trace\_formulas **where**  
HML\_trace\_formulas  $\equiv \{\varphi. \text{HML\_trace } \varphi\}$

translation of a trace to a formula

```

fun trace_to_formula :: 'a list  $\Rightarrow$  ('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  $\Rightarrow$  bool
  where
failure_tt: HML_failure TT |
failure_pos: HML_failure (hml_pos  $\alpha$   $\varphi$ ) if HML_failure  $\varphi$  |
failure_conj: HML_failure (hml_conj I J  $\psi$ s)
if ( $\forall i \in I. \text{TT\_like } (\psi \text{ s } i) \wedge (\forall j \in J. (\text{TT\_like } (\psi \text{ s } j)) \vee (\exists \alpha \chi. ((\psi \text{ s } j) = \text{hml\_pos } \alpha \chi \wedge (\text{TT\_like } \chi))))$ )

```

```

inductive HML_simulation :: ('a, 's)hml  $\Rightarrow$  bool
  where
sim_tt: HML_simulation TT |
sim_pos: HML_simulation (hml_pos  $\alpha$   $\varphi$ ) if HML_simulation  $\varphi$  |
sim_conj: HML_simulation (hml_conj I J  $\psi$ s)
if ( $\forall x \in (\psi \text{ s } \setminus I). \text{HML\_simulation } x \wedge (\psi \text{ s } \setminus J = \{\})$ )

```

**definition** HML\_simulation\_formulas **where**  
HML\_simulation\_formulas  $\equiv \{\varphi. \text{HML\_simulation } \varphi\}$

```

inductive HML_readiness :: ('a, 's)hml  $\Rightarrow$  bool
  where
read_tt: HML_readiness TT |
read_pos: HML_readiness (hml_pos  $\alpha$   $\varphi$ ) if HML_readiness  $\varphi$  |
read_conj: HML_readiness (hml_conj I J  $\Phi$ )
if ( $\forall x \in (\Phi \setminus (I \cup J)). \text{TT\_like } x \vee (\exists \alpha \chi. x = \text{hml\_pos } \alpha \chi \wedge \text{TT\_like } \chi)$ )

```

```

inductive HML_impossible_futures :: ('a, 's)hml  $\Rightarrow$  bool
  where
  if_tt: HML_impossible_futures TT |
  if_pos: HML_impossible_futures (hml_pos  $\alpha$   $\varphi$ ) if HML_impossible_futures  $\varphi$  |
  if_conj: HML_impossible_futures (hml_conj I J  $\Phi$ )
if  $\forall x \in (\Phi \setminus I). \text{TT\_like } x \vee \forall x \in (\Phi \setminus J). (\text{HML\_trace } x)$ 

```

```

inductive HML_possible_futures :: ('a, 's)hml  $\Rightarrow$  bool
  where

```

```

pf_tt: HML_possible_futures TT |
pf_pos: HML_possible_futures (hml_pos  $\alpha$   $\varphi$ ) if HML_possible_futures  $\varphi$ 
|
pf_conj: HML_possible_futures (hml_conj I J  $\Phi$ )
if  $\forall x \in (\Phi \setminus (I \cup J)). (HML\_trace\ x)$ 

```

**definition** HML\_possible\_futures\_formulas **where**

HML\_possible\_futures\_formulas  $\equiv \{\varphi. HML\_possible\_futures\ \varphi\}$

**inductive** HML\_failure\_trace :: ('a, 's)hml  $\Rightarrow$  bool

**where**

```

f_trace_tt: HML_failure_trace TT |
f_trace_pos: HML_failure_trace (hml_pos  $\alpha$   $\varphi$ ) if HML_failure_trace  $\varphi$  |
f_trace_conj: HML_failure_trace (hml_conj I J  $\Phi$ )
if  $((\exists \psi \in (\Phi \setminus I). (HML\_failure\_trace\ \psi) \wedge (\forall y \in (\Phi \setminus I). \psi \neq y \longrightarrow$ 
nested_empty_conj y))  $\vee$ 
 $(\forall y \in (\Phi \setminus I). nested\_empty\_conj\ y)) \wedge$ 
 $(\forall y \in (\Phi \setminus J). stacked\_pos\_conj\_pos\ y)$ 

```

**inductive** HML\_ready\_trace :: ('a, 's)hml  $\Rightarrow$  bool

**where**

```

r_trace_tt: HML_ready_trace TT |
r_trace_pos: HML_ready_trace (hml_pos  $\alpha$   $\varphi$ ) if HML_ready_trace  $\varphi$  |
r_trace_conj: HML_ready_trace (hml_conj I J  $\Phi$ )
if  $(\exists x \in (\Phi \setminus I). HML\_ready\_trace\ x \wedge (\forall y \in (\Phi \setminus I). x \neq y \longrightarrow single\_pos$ 
y))
 $\vee (\forall y \in (\Phi \setminus I). single\_pos\ y)$ 
 $(\forall y \in (\Phi \setminus J). single\_pos\_pos\ y)$ 

```

**inductive** HML\_ready\_sim :: ('a, 's) hml  $\Rightarrow$  bool

**where**

```

HML_ready_sim TT |
HML_ready_sim (hml_pos  $\alpha$   $\varphi$ ) if HML_ready_sim  $\varphi$  |
HML_ready_sim (hml_conj I J  $\Phi$ ) if
 $(\forall x \in (\Phi \setminus I). HML\_ready\_sim\ x) \wedge (\forall y \in (\Phi \setminus J). single\_pos\_pos\ y)$ 

```

**inductive** HML\_2\_nested\_sim :: ('a, 's) hml  $\Rightarrow$  bool

**where**

```

HML_2_nested_sim TT |
HML_2_nested_sim (hml_pos  $\alpha$   $\varphi$ ) if HML_2_nested_sim  $\varphi$  |
HML_2_nested_sim (hml_conj I J  $\Phi$ )
if  $(\forall x \in (\Phi \setminus I). HML\_2\_nested\_sim\ x) \wedge (\forall y \in (\Phi \setminus J). HML\_simulation$ 
y)

```

**inductive** HML\_revivals :: ('a, 's) hml  $\Rightarrow$  bool

**where**

```

revivals_tt: HML_revivals TT |
revivals_pos: HML_revivals (hml_pos  $\alpha$   $\varphi$ ) if HML_revivals  $\varphi$  |

```



```

revivals_conj: HML_revivals (hml_conj I J  $\Phi$ ) if ( $\forall x \in (\Phi \setminus I). \exists \alpha \chi. (x = \text{hml\_pos } \alpha \chi) \wedge \text{TT\_like } \chi$ )
( $\forall x \in (\Phi \setminus J). \exists \alpha \chi. (x = \text{hml\_pos } \alpha \chi) \wedge \text{TT\_like } \chi$ )

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

primrec expr_1 :: ('a, 's)hml  $\Rightarrow$  enat
  where
    expr_1_tt:  $\langle \text{expr}_1 \text{ TT} = 0 \rangle$  |
    expr_1_conj:  $\langle \text{expr}_1 (\text{hml\_conj } I \ J \ \Phi) = \text{Sup } ((\text{expr}_1 \circ \Phi) \setminus I \cup (\text{expr}_1 \circ \Phi) \setminus J) \rangle$  |
    expr_1_pos:  $\langle \text{expr}_1 (\text{hml\_pos } \alpha \ \varphi) = 1 + (\text{expr}_1 \ \varphi) \rangle$ 

primrec expr_2 :: ('a, 's)hml  $\Rightarrow$  enat
  where
    expr_2_tt:  $\langle \text{expr}_2 \text{ TT} = 1 \rangle$  |
    expr_2_conj:  $\langle \text{expr}_2 (\text{hml\_conj } I \ J \ \Phi) = 1 + \text{Sup } ((\text{expr}_2 \circ \Phi) \setminus I \cup (\text{expr}_2 \circ \Phi) \setminus J) \rangle$  |
    expr_2_pos:  $\langle \text{expr}_2 (\text{hml\_pos } \alpha \ \varphi) = \text{expr}_2 \ \varphi \rangle$ 

primrec expr_3 :: ('a, 's) hml  $\Rightarrow$  enat
  where
    expr_3_tt:  $\langle \text{expr}_3 \text{ TT} = 0 \rangle$  |
    expr_3_pos:  $\langle \text{expr}_3 (\text{hml\_pos } \alpha \ \varphi) = \text{expr}_3 \ \varphi \rangle$  |
    expr_3_conj:  $\langle \text{expr}_3 (\text{hml\_conj } I \ J \ \Phi) = (\text{Sup } ((\text{expr}_1 \circ \Phi) \setminus I \cup (\text{expr}_3 \circ \Phi) \setminus I \cup (\text{expr}_3 \circ \Phi) \setminus J)) \rangle$ 

fun pos_r :: ('a, 's)hml set  $\Rightarrow$  ('a, 's)hml set
  where
    pos_r xs = (
      let max_val = (Sup (expr_1 ` xs));
      max_elem = (SOME  $\psi. \psi \in \text{xs} \wedge \text{expr}_1 \ \psi = \text{max\_val}$ );
      xs_new = xs - {max_elem}
      in xs_new)

primrec expr_4 :: ('a, 's)hml  $\Rightarrow$  enat
  where
    expr_4_tt:  $\text{expr}_4 \text{ TT} = 0$  |
    expr_4_pos:  $\text{expr}_4 (\text{hml\_pos } a \ \varphi) = \text{expr}_4 \ \varphi$  |
    expr_4_conj:  $\text{expr}_4 (\text{hml\_conj } I \ J \ \Phi) = \text{Sup } ((\text{expr}_1 \setminus (\text{pos\_r } (\Phi \setminus I)))$ 

```

$$\cup (\text{expr}_4 \circ \Phi) \setminus I \cup (\text{expr}_4 \circ \Phi) \setminus J)$$

```
primrec expr_5 :: ('a, 's)hml  $\Rightarrow$  enat
```

```
  where
```

```
  expr_5_tt: <expr_5 TT = 0> |
```

```
  expr_5_pos: <expr_5 (hml_pos  $\alpha$   $\varphi$ ) = expr_5  $\varphi$ > |
```

```
  expr_5_conj: <expr_5 (hml_conj I J  $\Phi$ ) =
```

```
  (Sup ((expr_5  $\circ$   $\Phi$ )  $\setminus$  I  $\cup$  (expr_5  $\circ$   $\Phi$ )  $\setminus$  J  $\cup$  (expr_1  $\circ$   $\Phi$ )  $\setminus$  J)))>
```

```
primrec expr_6 :: ('a, 's)hml  $\Rightarrow$  enat
```

```
  where
```

```
  expr_6_tt: <expr_6 TT = 0> |
```

```
  expr_6_pos: <expr_6 (hml_pos  $\alpha$   $\varphi$ ) = expr_6  $\varphi$ > |
```

```
  expr_6_conj: <expr_6 (hml_conj I J  $\Phi$ ) =
```

```
  (Sup ((expr_6  $\circ$   $\Phi$ )  $\setminus$  I  $\cup$  ((eSuc  $\circ$  expr_6  $\circ$   $\Phi$ )  $\setminus$  J)))>
```

```
fun expr :: ('a, 's)hml  $\Rightarrow$  enat  $\times$  enat  $\times$  enat  $\times$  enat  $\times$  enat  $\times$  enat
```

```
  where
```

```
  <expr  $\varphi$  = (expr_1  $\varphi$ , expr_2  $\varphi$ , expr_3  $\varphi$ , expr_4  $\varphi$ , expr_5  $\varphi$ , expr_6  $\varphi$ )>
```

```
end
```