Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

25th March 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 Bisping in [Bis23]. I formalize the concepts and proofs discussed in this thesis in the interactive proof assistant Isabelle.

Reactive systems are computing systems that continuously interact with their environment, reacting to external stimuli and producing outputs accordingly [HP85]. At a high level of abstraction, these systems can be seen as collections of interacting processes, where each process represents a state or configuration of the system. Labeled Transition Systems (LTS) [Kel76] provide a formal framework for modeling and analyzing the behavior of reactive systems. Roughly, an LTS is a labeled directed graph, whose nodes denote the processes and whose edges correspond to transitions between these processes (or states).

Verification of these systems involves proving statements regarding the behavior of such a system model. Often, verification tasks aim to show that a system's observed behavior aligns with its intended behavior. That requires criteria of what constitutes similar behavior on LTS, commonly referred to as the *semantics of equality* of processes. Depending on the requirements of a particular user, many different such criterions have been defined. For a subset of processes, namely the class of concrete sequential processes, [vG01] classified many such semantics. *Sequential* means that the processes can only perform one action at a time. *Concrete* processes are processes in which no internal actions occur, meaning that it exclusively captures the system's interactions with its environment. In such LTS, every transition represents an observable event or action between the system and its environment. The classification in [vG01] involved partially ordering many of these semantics by the relation 'makes strictly more identifications on processes than'. The resulting lattice is known as the (infinitary) linear-time–branching-time spectrum[1][2]. One way to characterize the behavior of LTS is through the use of modal logics. Formulas of a logic can be seen as describing certain properties of states within an LTS. A commonly used modal logic is Hennessy—Milner logic (HML) [HM85]. Equivalence in terms of HML is determined by whether processes satisfy the same set of formulas. The linear-time–branching-time spectrum can be recharted in terms of the subset relation between these modal-logical characterizations. In the context of this spectrum, demonstrating that a system model's observed behavior aligns with the behavior of a model of the specification can be done by 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 [BJN22][Bis23]. These approaches rechart the linear-time–branching-time spectrum using an expressiveness function that assigns a *formula price* to every formula. This price is supposed to capture the expressive capabilities of a particular formula. However, to be sure that these characterizations really capture the desired equivalences, one has to perform the proofs.

Contributions

This thesis provides a machine-checkable proof that the price bounds of the expressiveness function expr of [Bis23] correspond to the modal-logical characterizations of named equivalences. More precisely, we consider a formula φ to be in an observation language $\mathcal{O}_X$ iff its price is within the given price bound. For every expressiveness price bound e_X, we derive the sublanguage of Hennessy–Miler logic $\mathcal{O}_X$ and show that a formula φ is in $\mathcal{O}_X$ precisely if its price $\texttt{expr}(\varphi)$ is less than or equal to e_X. Then we show that $\mathcal{O}_X$ has exactly the same distinguishing power as the modal-logical characterization of that equivalence. In (ref Foundations (chapter 2)) we discuss and introduce formal definitions of LTSs, Hennessy-Milner logic and the expressiveness function expr. In (ref The Correspondances?! name!) we provide modal-logical definitions and perform the proofs for the standard notions of equivalence, i.e. the equivalences of (ref Figure 1). Namely for trace-, failures-, failure-trace-, readiness-, ready-trace-, revivals-, possible-futures-, impossible-futures-, simulation-, ready-simulation-, 2-nested-simulation- and bisimulation semantics. All the main concepts and proofs have been formalized and conducted using the interactive proof assistant Isabelle. More information on Isabelle can be found in (appendix?). We tried to present Isabelle implementations directly after their corresponding mathematical

[1]On Infinity?

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

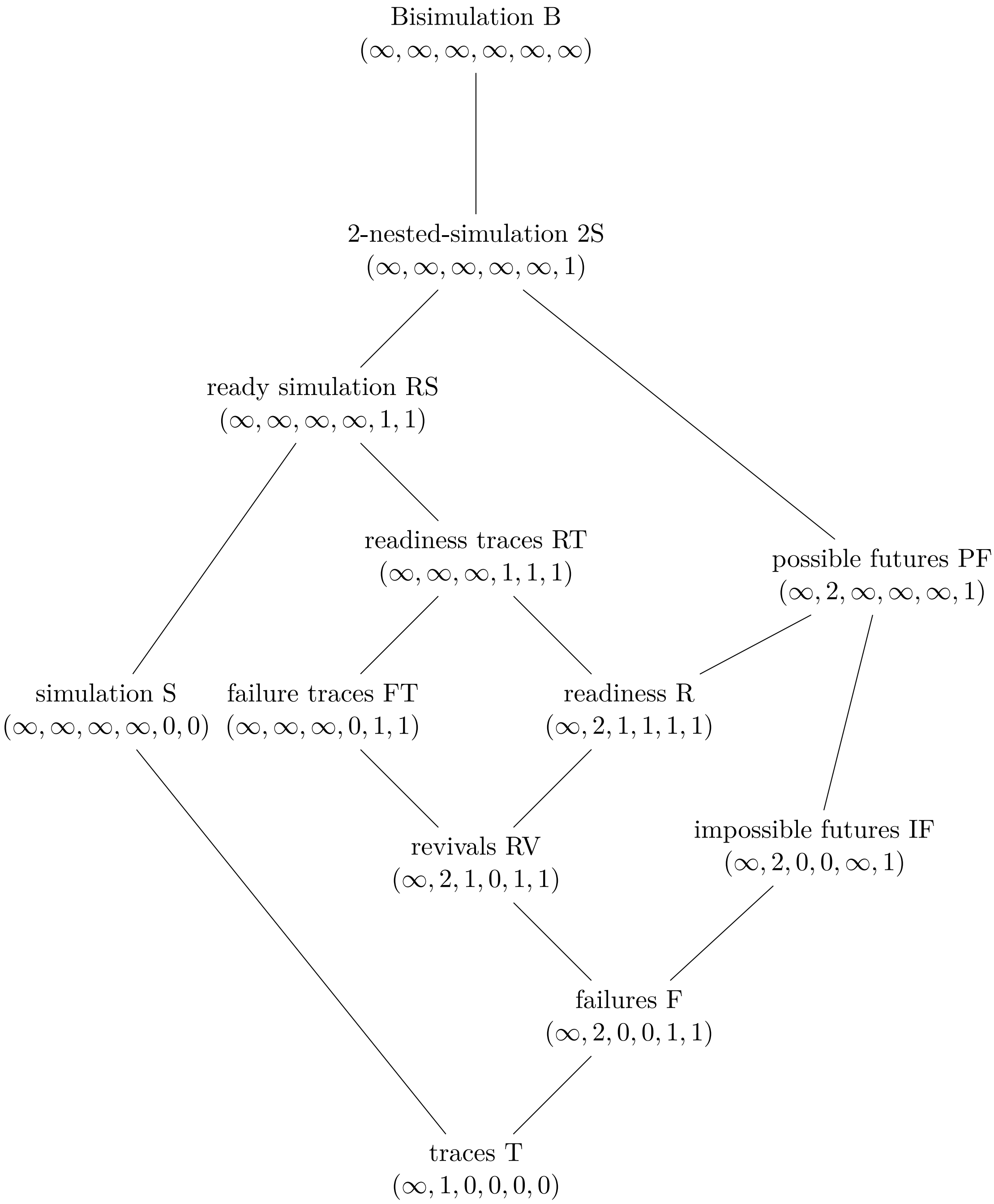

Figure 1.1: linear-time–branching-time spectrum

definitions. The mathematical definitions are marked as 'definitions' and presented in standard text format. Their corresponding Isabelle implementations are presented right after, distinguished by their `monospaced font` and colored syntax highlighting. However, for readability purposes, a majority of the Isabelle proofs are hidden and replaced by ⟨*proof*⟩ and some lemmas excluded. The whole Isabelle code and a web version of this thesis can be found on Github[3].

[3] Link!!!

Chapter 2

Foundations

In this chapter, relevant concepts will be introduced as well as formalized in Isabelle. The formalizations of (sections 2.1 and 2.2) are based on those done by Benjamin Bisping (cite) and Max Pohlmann (Cite).

2.1 Labeled Transition Systems

As described in chapter 1, 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 by the same or different actions. This apparent 'choice' of transition signifies that the system can select from these options non-deterministicaly[1]. The semantic equivalences we investigate are defined entirely in terms of action relations. Many modeling methods use a special τ-action to represent internal behavior. These internal transitions are not observable from the outside, which yields new notions of equivalence. However, in our definition of LTS, τ-transitions are not explicitly treated different from other transitions.

[1]In the context of reactive systems, this 'choice' is a representation of the system's possible behaviors rather than actual non-determinism. In reality, transitions represent synchronizations with the system's environment. The next state of the system is then uniquely determined by its current state and the external action.

Definition 2.1.1 (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* $\cdot \xrightarrow{\cdot} \cdot \subseteq Proc \times Act \times Proc$ *is a transition relation. We write* $p \xrightarrow{\alpha} p'$ *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`. In Isabelle we associate `tran` 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
```

The graph 2.1 depicts a simple LTS. Depending on how ‘close’ we look, we might consider the observable behaviors of p_1 and q_1 equivalent or not.

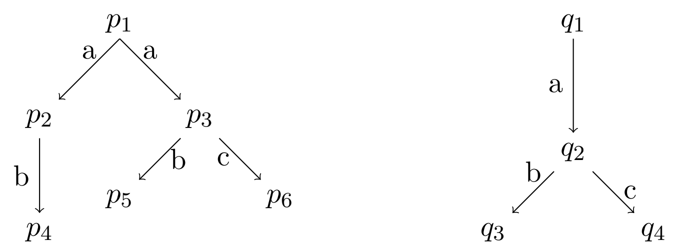

Figure 2.1: Counterexample 3 glaabbeck

If we compare the states p_1 and q_1 of (ref example 1), we can observe many similarities but also differences in their behavior. They can perform the same set of action sequences; however, the state p_1 can transition to p_2 via an a-transition, whereas only a b-transition is possible from q_1 to q_2, where both b and c actions are possible.

Abstracting away details of the inner workings of a system leads us to a notion of equivalence that focuses solely on its externally observable behavior, called *trace equivalence*. We can imagine an observer who simply records the events of a process as they occur. This observer views two processes as equivalent if and only if they allow the same sequences of actions. As discussed, p_1 and q_1 are trace-equivalent since they allow for the same action sequences. In contrast, *strong bisimilarity*[2] considers two states equivalent

[2]Behavioral equivalences are commonly denoted as strong, as opposed to weak, if they do not take internal behavior into account. Since we are only concerned with concrete processes, we omit such qualifiers.

if, for every possible action of one state, there exists a corresponding action of the other, and vice versa. Additionally, the resulting states after taking these actions must also be bisimilar. The states p_1 and q_1 are not bisimilar, since for an a-transition from q_1 to q_2, p_1 can perform an a-transition to p_2, but q_2 and p_2 do not have the same possible actions. Bisimilarity is the finest[3] commonly used *extensional behavioral equivalence*. 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*, where the (intensional) identity of processes is relevant.

Figure 1.1 charts the *linear-time–branching-time-spectrum*. This spectrum orders behavioral equivalences between trace- and bisimulation semantics by how refined one equivalence is. Finer equivalences make more distinctions between processes, while coarser ones make fewer distinctions. If processes are equated by one notion of equivalence, they are also equated by every notion below. Note that, like [Bis23], we omit the examination of completed trace, completed simulation and possible worlds observations (evtl discussion?).

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

Definition 2.1.2

- *The* α-derivatives *of a state refer to the set of states that can be reached with an* α*-transition:* $Der(p, \alpha) = \{p' \mid p \xrightarrow{\alpha} p'\}$.
- *A process is in a* deadlock *if no observation is possible. That is:* $deadlock(p) = (\forall \alpha. Der(p, \alpha) = \varnothing)$
- *The set of* initial actions *of a process* p *is defined by:* $I(p) = \{\alpha \in Act \mid \exists p'.p \xrightarrow{\alpha} p'\}$
- *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:*

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

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

- *We call a sequence of states* $s_0, s_1, s_2, ..., s_n$ *a* path *if there exists a step sequence between* s_0 *and* s_n.

3

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

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

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

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

If there exists a path from p to p'' there exists a corresponding step sequence and vice versa.

```
lemma
  assumes ‹paths (p # ps @ [p''])›
  shows ‹∃tr. p ↦$ tr p''›
  ⟨proof⟩

lemma
  assumes ‹p ↦$ tr p''›
  shows ‹∃ps. paths (p # ps @ [p''])›
  ⟨proof⟩
```

LTSs can be classified by imposing limitations on the number of possible transitions from each state.

Definition 2.1.3

A process p is image-finite *if, for each $\alpha \in Act$, the set $Der(p, \alpha)$ is finite. A LTS is image-finite if each $p \in Proc$ is image-finite: $\forall p \in Proc, \alpha \in Act.Der(p, \alpha)$ is finite.*

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

Our definition of LTS allows for an unrestricted number of states, all of which can be arbitrarily branching. This means that they can have unlimited ways to proceed. Given the possibility of infinity in sequential and

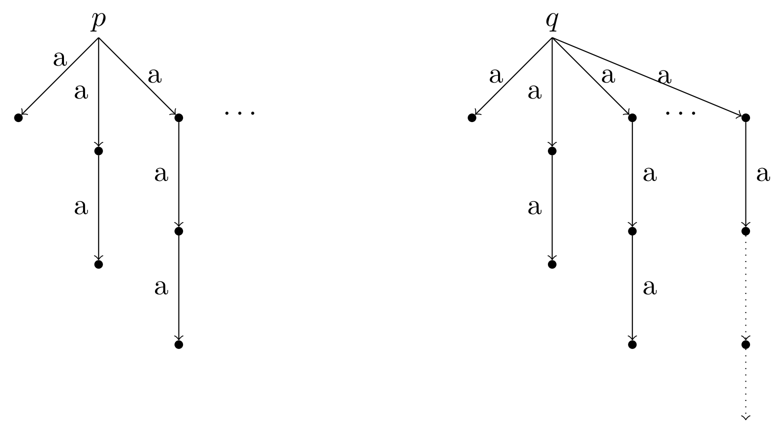

Figure 2.2: counterexample glaabeeck (cite)

branching behavior, we must consider how we identify processes that only differ in their infinite behavior. Take the states p and q of 2.2, they have the same (finite) step sequences, however, only q has an infinite trace. Do we consider them trace equivalent? This distinction criterion leads to a number of new equivalences. (Van glaabeeck) distinguishes between finite and infinite versions for all equivalences. They also investigate an intermediate version for simulation-like semantics, that assumes that an observer can investigate arbitrary many properties of a process in parallel, but only in a finite amount of time, and a version of the finite versions of semantics with refusal sets, where these sets are finite. This thesis focuses on the default versions of these semantics, allowing for infinite copies of a process to be tested but only for a finite duration. That corresponds to the finitary version for trace-like semantics. Processes whose behavior differ only in infinite execution, such as p and q, are considered equivalent regarding trace-like semantics. For simulation-like semantics, this corresponds to the infinitary version. An observer can observe arbitrary many copies of a processes, and can therefore also observe infinite sequential behavior (see van glaabeeck prop 8.3, theorem 4). This means that simulation-like semantics can distinguish between p and q (see simulation chapter).

2.2 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 [HM85], 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 if there exists no observation that distinguishes between them. In their seminal paper, Matthew Hennessy and Robin Milner defined the modal-logical language as consisting of (finite) conjunctions, negations and a (modal) possibility operator:

$$\varphi ::= tt \mid \varphi_1 \;\wedge\; \varphi_2 \mid \neg\varphi \mid \langle\alpha\rangle\varphi \quad \text{with } \alpha \in \Sigma$$

The paper also proves that this language characterizes a relation that is effectively the same as bisimilarity. This theorem is called the Hennessy–Milner Theorem and can be expressed as follows: for image-finite LTSs, two processes are bisimilar iff they satisfy the same set of HML formulas. We call this the modal characterization of bisimilarity. (Infinitary) Hennessy–Milner logic extends the original definition by allowing for conjunctions of arbitrary width. This yields the modal characterization of bisimilarity for arbitrary LTS and is proven in (Appendix). In the following sections we always mean the infinitary version when talking about HML.

Definition 2.2.1 (Hennessy–Milner logic)

Syntax *The* syntax of Hennessy–Milner logic *over a set* Σ *of actions HML[*Σ*] is defined by the grammar:*

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

Where I *denotes an index set. The empty conjunction* $\top := \bigwedge \varnothing$ *is usually omitted in writing.*

Semantics *The* semantics of HML *parametrized by* Σ *(on LTS processes) are given by the relation* $\models$ *:* $(\mathit{Proc}, \mathit{HML}[\Sigma])$*:*

$$\begin{aligned} p &\models \langle\alpha\rangle\varphi && \textit{if there exists } q \textit{ such that } q \in \mathit{Der}(p,\alpha) \textit{ and } q \models \varphi \\ p &\models \bigwedge_{i\in I} \psi_i && \textit{if } p \models \psi_i \textit{ for all } i \in I \\ p &\models \neg\varphi && \textit{if } p \not\models \varphi \end{aligned}$$

$\langle a\rangle$ captures the observation of an a-transition by the system. Similar to propositional logic, conjunctions are used to describe multiple properties of a state that must hold simultaneously. Each conjunct represents a possible branching or execution path of the system. $\neg\varphi$ indicates the absence of behavior represented by the subformula φ.

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 include the constructor `TT` for the formula T as part of the Isabelle syntax. This is to enable Isabelle to infer that the type `('a, 'i)hml` is not empty. The constructor `hml_pos` corresponds directly to the possibility operator. Conjunctions are formalized using the constructor `hml_conj`. The constructor has two index sets of arbitrary type `'i set` and a mapping `F :: 'i ⇒ ('a, 'i) hml` as type variables. The first variable is used to denote the positive conjuncts and the second denotes the negative conjuncts. The term `(hml_conj I J Φ)` corresponds to $\bigwedge \left(\bigcup_{i \in I}\{(\Phi\ i)\} \cup \bigcup_{i \in J}\{\neg(\Phi\ i)\}\right)$. We decided to formalize HML without the explicit ψ to avoid using mutual recursion, since it is harder to handle especially in proofs using induction over the data type. Note that the formalization via an arbitrary set, i.e. `hml_conj ‹('a)hml set›` does not yield a valid type, since `set` is not a bounded natural functor. Corresponding to the mathematical definition, this formalization allows for conjunctions of arbitrary—even of infinite—width.

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

The semantic models-relation is formalized in Isabelle in the context of LTS. This means that the index type `'i` is replaced by the type of processes `'s`. Since this modal-logically characterizes bisimilarity, we can conclude that it suffices for the cardinality of the indexsets to be equal to the cardinality of the set of processes.

```
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))))›
```

A formula that is true for all processes in a LTS can be considered a property that holds universally for the system, akin to a tautology in classical logic.

```
definition HML_true where
HML_true φ ≡ ∀s. s ⊨ φ
```
⟨*proof*⟩

Definition 2.2.2

- *As discussed, equivalences in LTS can be defined in terms of* HML *subsets. Two processes are* X-equivalent *regarding a subset of Hennessy–Milner logic,* $\mathcal{O}_X \subseteq \text{HML}[\Sigma]$, *if they satisfy the same formulas of that subset.*

- *A subset provides a* modal-logical characterization *of an equivalence X if, according to that subset, the same processes are considered equivalent as they are under the colloquial definition of that equivalence.*

- *A formula* $\varphi \in \mathrm{HML}[\Sigma]$ distinguishes *one state from another if it is true for the former and false for the latter.*

We do not introduce the modal-logical characterizations of all equivalences here, but one by one in chapter (ref).

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

definition HML_equivalent :: 's ⇒ 's ⇒ bool where
  HML_equivalent p q ≡ HML_subset_equivalent {φ. True} p q

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

For the purposes of this thesis, we consider the modal-logical characterizations, similar to those presented in (van Glaabbeck), as synonymous with the characterization of the equivalences. X-equivalence of two processes p and q is denoted by $p \sim_X q$. If they are equivalent for every formula in $\mathrm{HML}[\Sigma]$, they are bisimilar, denoted as $p \sim_B q$.

Next we show some properties to better talk about these definitions. We show that $\cdot \sim_X \cdot$ is an equivalence relation. Also, the equivalence is preserved under transitions, meaning that the X-equivalence is maintained when processes transition to new states. Finally, we show that if two states are not HML equivalent, there must be a distinguishing formula.

```
lemma ‹equivp (HML_subset_equivalent X)›
  ⟨proof⟩

lemma equiv_der:
  assumes HML_equivalent p q ∃p'. p ↦α p'
  shows ∃p' q'. (HML_equivalent p' q') ∧ q ↦α q'
  using assms hml_semantics.simps
  unfolding HML_equivalent_def HML_subset_equivalent_def
  by (metis UNIV_def iso_tuple_UNIV_I)

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

Example 1. We can now use HML to capture differences between p_1 and q_1 of Figure 2.1. The formula $\varphi_1 := \langle a \rangle \bigwedge\{\neg\langle c \rangle\}$ distinguishes p_1 from q_1 and $\varphi_2 := \bigwedge\{\neg\langle a \rangle \bigwedge\{\neg\langle c \rangle\}\}$ distinguishes q_1 from p_1. The Hennessy–Milner Theorem implies that if a distinguishing formula exists, then p_1 and q_1 cannot be bisimilar.

end — of context lts

2.3 Price Spectra of Behavioral Equivalences

[Bis23, BJN22] use a pricing system to measure the amount of HML-expressiveness used by a formula. By assigning an expressiveness price to each formula, the authors create a price lattice that allows for comparing distinguishing power of different formulas, where lower prices indicate less distinguishing power. This allows for a new way of defining HML-subsets. Instead of bounding the subsets by the structure of the included formulas, they are defined as sets of formulas whose prices are less than or equal to a given expressiveness price bound, or *price coordinates*. The value of each dimension of these price coordinates constrains different syntactic features of the formulas. The authors derive the linear-time–branching-time spectrum by assigning a price coordinate to every equivalence in the spectrum, see fig. 1.1. In this section, we introduce the definition of the expressiveness price function of ([Bis23], definition 5) and how that function is used to chart the spectrum.

Unlike [Bis23], we define the price for every dimension i as a seperate function, $\mathsf{expr}_i : \mathrm{HML}[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ and combine them to the expressiveness function, $\mathsf{expr} : \mathrm{HML}[\Sigma] \to (\mathbb{N} \cup \{\infty\})^6$. Each function inductively traverses the syntax tree of a formula and increases its value when encountering the respective syntax feature.

Definition 2.3.1 (Formula Prices)

(1) The modal depth expr_1 *measures the nesting depth of observations within a formula: HML*$[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\textit{if } \varphi = \langle a \rangle \psi \quad \textit{with } a \in \Sigma \\
&\quad \textit{then } \mathsf{expr}_1(\varphi) = 1 + \mathsf{expr}_1(\psi) \\
&\textit{if } \varphi = \bigwedge_{i \in I}\{\psi_1, \psi_2, \ldots\} \\
&\quad \textit{then } \mathsf{expr}_1(\varphi) = \sup(\mathsf{expr}_1(\psi_i)) \\
&\textit{if } \psi = \neg\varphi \\
&\quad \textit{then } \mathsf{expr}_1(\psi) = \mathsf{expr}_1(\varphi)
\end{aligned}$$

(2) The nesting depth of conjunctions expr_2 *measures the maximal number of conjunctions that are nested inside one another in a formula:* $HML[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\textit{if } \varphi = \langle a \rangle \psi \quad \textit{with } a \in \Sigma \\
&\quad \textit{then } \mathsf{expr}_2(\varphi) = \mathsf{expr}_2(\psi) \\
&\textit{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \textit{then } \mathsf{expr}_2(\varphi) = 1 + \sup(\mathsf{expr}_2(\psi_i)) \\
&\textit{if } \psi = \neg\varphi \\
&\quad \textit{then } \mathsf{expr}_2(\psi) = \mathsf{expr}_2(\varphi)
\end{aligned}$$

(3) The maximal modal depth of deepest positive clauses in conjunctions expr_3 *measures the deepest modal depth of the positive conjuncts of all conjunctions of a formula:* $HML[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\textit{if } \varphi = \langle a \rangle \psi \quad \textit{with } a \in \Sigma \\
&\quad \textit{then } \mathsf{md}(\varphi) = \mathsf{md}(\psi) \\
&\textit{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \textit{then } \mathsf{md}(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in Pos\} \cup \{\mathsf{expr}_3(\psi_i) | i \in I\}) \\
&\textit{if } \psi = \neg\varphi \\
&\quad \textit{then } \mathsf{expr}_3(\psi) = \mathsf{expr}_3(\varphi)
\end{aligned}$$

(4) The maximal modal depth of other positive clauses in conjunctions expr_4 *measures the deepest positive modal depth aside from the deepest positive clause:* $HML[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\textit{if } \varphi = \langle a \rangle \psi \quad \textit{with } a \in \Sigma \\
&\quad \textit{then } \mathsf{expr}_4(\varphi) = \mathsf{expr}_4(\psi) \\
&\textit{if } \varphi = \bigwedge_{i \in I} \{\ \psi_i\} \\
&\quad \textit{then } md(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in Pos \backslash \mathcal{R}\} \cup \{\mathsf{expr}_4(\psi_i) | i \in I\}) \\
&\textit{if } \psi = \neg\varphi \\
&\quad \textit{then } \mathsf{expr}_4(\psi) = \mathsf{expr}_4(\varphi)
\end{aligned}$$

(5) The maximal modal depth of negative clauses in conjunctions expr_5 *measures the deepest modal depth of the negative conjuncts of all conjunctions of a formula:* $HML[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{expr}_5(\varphi) = \mathsf{expr}_5(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \text{then } \mathsf{expr}_5(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in Neg\} \cup \{\mathsf{expr}_5(\psi_i) | i \in I\}) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_5(\psi) = \mathsf{expr}_5(\varphi)
\end{aligned}$$

(6) The nesting depth of negations expr_6 *measures the maximal number of negations when traversing the syntax tree of a formula:* $HML[\Sigma] \to (\mathbb{N} \cup \{\infty\})$ *of a formula* φ *is defined recursively by:*

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{expr}_6(\varphi) = \mathsf{expr}_6(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \text{then } \mathsf{expr}_6(\varphi) = \sup(\{\mathsf{expr}_6(\psi_i) | i \in I\}) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_6(\psi) = 1 + \mathsf{expr}_6(\varphi)
\end{aligned}$$

where:

$Neg := \{i \in I \mid \exists \varphi'_i . \psi_i = \neg \varphi'_i\}$

$Pos := I \setminus \text{Neg}$

$$\mathcal{R} := \left\{ \begin{matrix} \varnothing \text{ if } Pos = \varnothing, \\ \{r\} \text{ for some } r \in Pos \text{ where } expr_1(\psi_r) \text{ maximal for } Pos \end{matrix} \right\}.$$

We combine this to the expressiveness price $\mathsf{expr} : HML[\Sigma] \to (\mathbb{N} \cup \infty)^6$ *of a formula* φ*:*

$$expr(\varphi) := \begin{pmatrix} expr_1(\varphi) \\ expr_2(\varphi) \\ expr_3(\varphi) \\ expr_4(\varphi) \\ expr_5(\varphi) \\ expr_6(\varphi) \end{pmatrix}$$

We show that expr defines the same function as ([Bis23], definition 5) in (appendix).

The formalization closely follows the structure outlined in the definition. Neg and Pos can easily be derived using our formalization of Conjunctions. The function pos_r formalizes the set $Pos - \mathcal{R}$. It invokes the axiom of choice by selecting and removing a formula with maximal modal depth from Pos using the Hilbert choice operator SOME.

```
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)›
```

Prices are compared component wise, i.e., $(e_1, \ldots e_6) \leq (f_1 \ldots f_6)$ iff $e_i \leq f_i$ for each i.

```
fun less_eq_t :: (enat × enat × enat × enat × enat × enat) ⇒ (enat
× enat × enat × enat × enat × enat) ⇒ bool
  where
less_eq_t (n1, n2, n3, n4, n5, n6) (i1, i2, i3, i4, i5, i6) =
    (n1 ≤ i1 ∧ n2 ≤ i2 ∧ n3 ≤ i3 ∧ n4 ≤ i4 ∧ n5 ≤ i5 ∧ n6 ≤ i6)

definition less where
less x y ≡ less_eq_t x y ∧ ¬ (less_eq_t y x)


lemma example_2_3:
  fixes s and t and a and b and c
  assumes s ≠ t
  defines φ: (φ::('a, 's)hml) ≡
  (hml_pos a
    (hml_conj {s, t} {}
      (λi. (if i = s
            then (hml_pos b TT)
            else
              (if i = t
               then (hml_pos a
                      (hml_conj {} {s, t}
                        (λj. (if j = s
                              then (hml_pos a
                                     (hml_pos c TT))
                              else
                                (if j = t
                                 then (hml_pos b TT)
                                 else undefined)))))
               else undefined)))))

shows
  expr_1 φ = 4
  expr_2 φ = 3
```

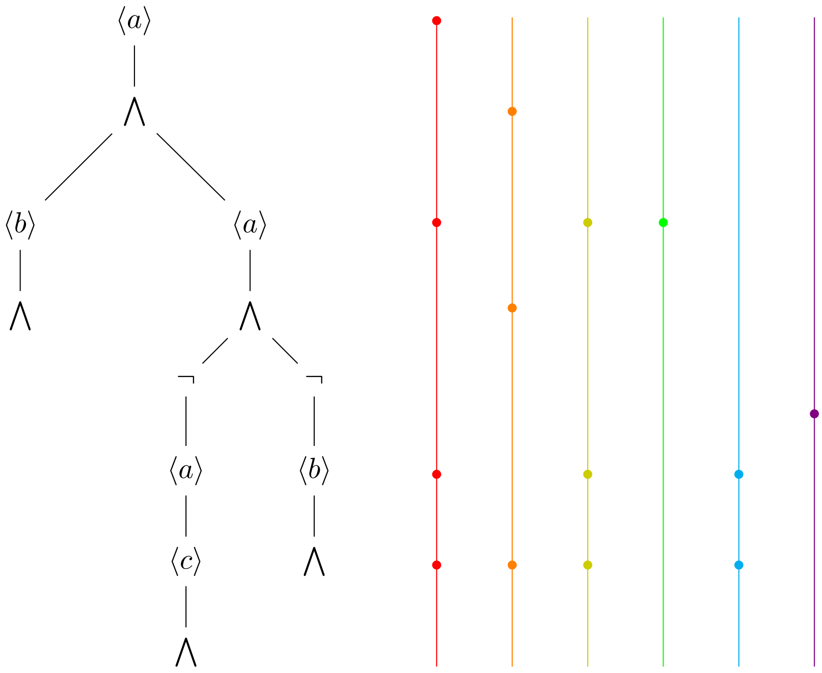

Figure 2.3: Pricing of formula $\langle a\rangle \bigwedge\{\langle b\rangle, \langle a\rangle \bigwedge\{\neg\langle a\rangle\langle c\rangle, \neg\langle b\rangle\}\}$

```
expr_3 φ = 3
expr_4 φ = 1
expr_5 φ = 2
expr_6 φ = 1
```
$\langle$*proof*$\rangle$

Example 2: This example illustrates how the prices of a formula are calculated. In Figure 2.3, you can see the pricing process for the formula $\langle a\rangle \bigwedge\{\langle b\rangle, \langle a\rangle \bigwedge\{\neg\langle a\rangle\langle c\rangle, \neg\langle b\rangle\}\}$. Each line to the right of the syntax tree represents the price of a specific dimension. The circles of each line represent an increase in the price of that dimension. The colors of these lines correspond to those defined in (ref definition 2.3.1) and indicate the dimension they represent. Note the finishing empty conjunction, which increases the conjunction depth by one. We can use the function to calculate the prices of the formulas in Example 1. The price of $\varphi_1 := \langle a\rangle \bigwedge\{\neg\langle c\rangle\}$ is $\mathit{expr}(\varphi_1) = (2, 2, 0, 0, 1, 1)$. For $\varphi_2 := \bigwedge\{\neg\langle a\rangle \bigwedge\{\neg\langle c\rangle\}\}$, $\mathit{expr}(\varphi_2) = (2, 3, 0, 0, 2, 2)$.

Proposition The expressiveness function is monotonic. Specifically, for any formula $\langle\alpha\rangle\varphi$, is the expressiveness of the subformula φ less than or equal to the expressiveness of $\langle\alpha\rangle\varphi$. Similarly, for any conjunctive formula $\bigwedge_{i\in I}\psi_i$, the expressiveness of every conjunct ψ_i is less than or equal to the expressiveness of $\bigwedge_{i\in I} \psi_i$.

```
lemma mon_pos:
  fixes n1 and n2 and n3 and n4::enat and n5 and n6 and α
  assumes A1: less_eq_t (expr (hml_pos α φ)) (n1, n2, n3, n4, n5, n6)
  shows less_eq_t (expr φ) (n1, n2, n3, n4, n5, n6)
⟨proof⟩

lemma mon_conj:
  fixes n1 and n2 and n3 and n4 and n5 and n6 and xs and ys
  assumes less_eq_t (expr (hml_conj I J Φ)) (n1, n2, n3, n4, n5, n6)
  shows (∀x ∈ (Φ ` I). less_eq_t (expr x) (n1, n2, n3, n4, n5, n6))
(∀y ∈ (Φ ` J). less_eq_t (expr y) (n1, n2, n3, n4, n5, n6))
⟨proof⟩
```

Chapter 3

Characterizing Equivalences

In this chapter, we introduce the modal-logical characterizations $\mathcal{O}_X$ of the various equivalences and link them to the HML sublanguages $\mathcal{O}_{e_X}$ determined certain by price bounds. The proofs follow the same structure: We first derive the modal characterization of $\mathcal{O}_{e_X}$ and then show that this characterization is equivalent to the corresponding $\mathcal{O}_X$. We derive these modal-logical characterizations from (Glaabbeeck). In the appendix we prove for trace equivalence $\mathcal{O}_T$ and bisimilarity $\mathcal{O}_B$ that the modal-logical characterization really captures the colloquial definitions via trace sets/the relational definition of bisimilarity.

3.1 Trace semantics

As discussed, trace semantics identifies two processes as equivalent if they allow for the same set of observations, or sequences of actions.

Definition 3.1.1

The modal-characterization of trace semantics *is given by the set $\mathcal{O}_T$ of trace formulas over Act, recursively defined by:*

$$\langle a\rangle\varphi \in \mathcal{O}_T \text{ if } \varphi \in \mathcal{O}_T \text{ and } a \in Act$$
$$\bigwedge \varnothing \in \mathcal{O}_T$$

```
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}
```

This definition allows for the construction of traces such as $\langle a_1\rangle\langle a_2\rangle \dots \langle a_n\rangle\mathsf{T}$, which represents action sequences or traces. Two processes p and q are considered trace-equivalent if they satisfy the same formulas in $\mathcal{O}_T$, namely

$$p \sim_T q \longleftrightarrow \forall \varphi \in \mathcal{O}_T . p \models \varphi \longleftrightarrow q \models \varphi$$

```
context lts
begin

definition hml_trace_equivalent where
hml_trace_equivalent p q ≡ HML_subset_equivalent hml_trace_formulas p
q
```

The subset $\mathcal{O}_X$ only allows for finite sequences of actions, without the use of conjunctions or negations. Therefore, the complexity of a trace formula is limited by its modal depth (and one conjunction for T). As a result, the language derivied from the price coordinate $(\infty, 1, 0, 0, 0, 0)$ encompasses all trace formulas. We refer to this HML-sublanguage as $\mathcal{O}e_T$.

```
definition expr_traces
  where
expr_traces = {φ. (less_eq_t (expr φ) (∞, 1, 0, 0, 0, 0))}

definition expr_trace_equivalent
  where
expr_trace_equivalent p q ≡ HML_subset_equivalent expr_traces p q
end
```

Proposition 3.1.2

The language of formulas with prices below $(\infty, 1, 0, 0, 0, 0)$ characterizes trace equivalence. That is, for two processes p and q, $p \sim_T q \longleftrightarrow p \sim_{e_T} q$. Explicitly:

$$\forall \varphi \in \mathcal{O}_T . p \models \varphi \longleftrightarrow q \models \varphi \longleftrightarrow \forall \varphi \in \mathcal{O}_{e_T} . p \models \varphi \longleftrightarrow q \models \varphi$$

Proof. We show that $\mathcal{O}_T$ and $\mathcal{O}_{e_T}$ capture the same set of formulas. We do this for both sides by induction over the structure of $\mathsf{HML}[\Sigma]$.
First, we show that if $\varphi \in \mathcal{O}_\mathrm{T}$, then $\mathsf{expr}(\varphi) \leq (\infty, 1, 0, 0, 0, 0)$:

(Base) Case $\bigwedge \varnothing$: We can easily derive that $\bigwedge \varnothing = (0, 1, 0, 0, 0, 0)$ and thus $\bigwedge \varnothing \leq (\infty, 1, 0, 0, 0, 0)$.

Case $\langle a\rangle\varphi$: Since $\langle a\rangle$ only adds to expr_1, we can easily show that if $\mathsf{expr}(\varphi) \leq (\infty, 1, 0, 0, 0, 0)$, then $\langle a\rangle\varphi \leq (\infty, 1, 0, 0, 0, 0)$.

Next, we show that if $\mathsf{expr}(\varphi) \leq (\infty, 1, 0, 0, 0, 0)$, then $\varphi \in \mathcal{O}_{\mathrm{X}}$:

Case $\bigwedge_{i \in I}(\psi_i)$: Since every formula ends with T, and expr_2 denotes the depth of a conjunction, $\mathsf{expr}_2(\bigwedge_{i \in I}(\psi_i)) \geq 2$ if $I \neq \varnothing$. Therefore, I must be empty.

Case $\langle a \rangle \varphi$: From the induction hypothesis and the monotonicity attribute, we have that $\varphi \in \mathcal{O}_T$. With the definition of $\mathcal{O}_T$, we have that $\langle a \rangle \varphi \in \mathcal{O}_T$.

```
lemma trace_right:
  assumes hml_trace φ
  shows (less_eq_t (expr φ) (∞, 1, 0, 0, 0, 0))
  ⟨proof⟩

lemma HML_trace_conj_empty:
  assumes A1: less_eq_t (expr (hml_conj I J Φ)) (∞, 1, 0, 0, 0, 0)
  shows I = {} ∧ J = {}
⟨proof⟩

lemma trace_left:
  assumes (less_eq_t (expr φ) (∞, 1, 0, 0, 0, 0))
  shows (hml_trace φ)
  ⟨proof⟩

context lts begin

lemma hml_trace_equivalent p q ⟷ expr_trace_equivalent p q
  ⟨proof⟩

end

⟨proof⟩⟨proof⟩⟨proof⟩
end
end
```

3.2 Failures semantics

We can imagine the observer not only observing all traces of a system but also identifying scenarios where specific behavior is not possible. For Failures in particular, the observer can distinguish between step-sequences based on what actions are possible in the resulting state. Another way to think about Failures is that the process autonomously chooses an execution path, but only using a set of free allowed actions. We want the failure formulas to represent either a trace (of the form $\langle a_1 \rangle \langle a_2 \rangle \ldots \langle a_n \rangle \mathsf{T}$) or a failure pair, where some set of actions is not possible (of the form $\langle a_1 \rangle \langle a_2 \rangle \ldots \langle a_n \rangle \bigwedge \langle a_i \rangle \mathsf{T}$).

Definition 3.2.1

The modal characterization of failures semantics $\mathcal{O}_F$ *is defined recursively:*

$$\langle a \rangle \varphi \text{ if } \varphi \in \mathcal{O}_F$$
$$\bigwedge_{i \in I} \neg \langle a \rangle \mathsf{T}$$

```
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)

definition hml_failure_formulas
  where
hml_failure_formulas ≡ {φ. hml_failure φ}
```

The processes p_1 and q_1 of Figure 2.1 are an example of two processes that are trace equivalent but not failures equivalent. The formula $\langle a \rangle \bigwedge \neg \langle b \rangle$ distinguishes p_1 from q_1 and is in $\mathcal{O}_F$.

The syntactic features of failures formulas or those of trace formulas, extended by a possible conjunction over negated actions at the end of the sequence of observations. This increases the bound for nesting depth of conjunctions, the depth of negations and the modal depth of negative clauses by one. As a result, the price coordinate is $(\infty, 2, 0, 0, 1, 1)$.

We define the sublanguage $\mathcal{O}_{e_F}$ as the set of formulas φ with prices less than or equal to $(\infty, 2, 0, 0, 1, 1)$.

```
definition expr_failure
  where
expr_failure = {φ. (less_eq_t (expr φ) (∞, 2, 0, 0, 1, 1))}

context lts
begin
```

We define the equivalences accordingly. Two processes p q are considered Failures equivalent $\sim_F$ iff there is no formula in $\mathcal{O}_F$ that distinguishes them.

```
definition hml_failure_equivalent
  where
hml_failure_equivalent p q ≡ HML_subset_equivalent hml_failure_formulas
p q
```

p and q are to be considered equivalent iff there is no formula in $\mathcal{L}_F$ that distinguishes them.

```
definition expr_failure_equivalent
  where
expr_failure_equivalent p q ≡ HML_subset_equivalent expr_failure p q
end
```

Proposition 3.2.2

$p \sim_F q \longleftrightarrow p \sim_{e_F} q$.
The language of formulas with prices below $(\infty, 2, 0, 0, 1, 1)$ characterizes trace equivalence.

Proof. We derivive the modal-logical definition of $\mathcal{O}_{e_F}$. Due to the characteristics of the expr function, this definition differs from $\mathcal{O}_F$. Then we show the actual equivalence by... .

According to the definition of expr, we have: $\mathsf{expr}(\bigwedge_{i \in I} \psi_i) \in \mathcal{O}_{e_F} \leq (\infty, 2, 0, 0, 1, 1)$. This holds true if

1. For all ψ_i where $i \in \text{Pos}$:

- $\mathsf{expr}_1(\psi_i) \leq 0$
- $\mathsf{expr}_2(\psi_i) \leq 1$

This implies that the modal depth is 0 and the conjunction depth is also 0. Consequently, every ψ_i has the form T.

2. For all ψ_i where $i \in \text{Neg}$:

- $\mathsf{expr}_1(\psi_i) \leq 1$
- $\mathsf{expr}_2(\psi_i) \leq 1$

This implies that the maximal modal depth is 1 and the conjunction depth is also 1. Consequently, every ψ_i has the form T or $\langle a \rangle \mathsf{T}$.

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

lemma expr_TT:
  assumes TT_like χ
  shows expr χ = (0, 1, 0, 0, 0, 0)
⟨proof⟩

lemma assumes TT_like χ
shows e1_tt: expr_1 (hml_pos α χ) = 1
and e2_tt: expr_2 (hml_pos α χ) = 1
and e3_tt: expr_3 (hml_pos α χ) = 0
```

```
and e4_tt: expr_4 (hml_pos α χ) = 0
and e5_tt: expr_5 (hml_pos α χ) = 0
and e6_tt: expr_6 (hml_pos α χ) = 0
  ⟨proof⟩

context lts begin
lemma HML_true_TT_like:
  assumes TT_like φ
  shows HML_true φ
  ⟨proof⟩
end

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))))

lemma mon_expr_1_pos_r:
  Sup (expr_1 ` (pos_r xs)) ≤ Sup (expr_1 ` xs)
  ⟨proof⟩

lemma failure_right:
  assumes HML_failure φ
  shows (less_eq_t (expr φ) (∞, 2, 0, 0, 1, 1))
  ⟨proof⟩

lemma failure_pos_tt_like:
  assumes less_eq_t (expr (hml_conj I J Φ)) (∞, 2, 0, 0, 1, 1)
shows (∀i ∈ I. TT_like (Φ i))
⟨proof⟩

lemma expr_2_le_1:
  assumes expr_2 (hml_conj I J Φ) ≤ 1
  shows Φ ` I = {} Φ ` J = {}
⟨proof⟩

lemma expr_2_expr_5_restrict_negations:
  assumes expr_2 (hml_conj I J Φ) ≤ 2 expr_5 (hml_conj I J Φ) ≤ 1
  shows (∀j ∈ J. (TT_like (Φ j)) ∨ (∃α χ. ((Φ j) = hml_pos α χ ∧ (TT_like
χ))))
⟨proof⟩

lemma failure_left:
  fixes φ
  assumes (less_eq_t (expr φ) (∞, 2, 0, 0, 1, 1))
  shows HML_failure φ
```

```
  ⟨proof⟩

lemma failure_lemma:
  shows (HML_failure φ) = (less_eq_t (expr φ) (∞, 2, 0, 0, 1, 1))
  ⟨proof⟩

context lts begin
lemma hml_failure_equivalent p q ⟷ expr_failure_equivalent p q  ⟨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›
end
end
theory Failure_traces
  imports Failures Transition_Systems HML formula_prices_list Expr_helper
begin
```

3.3 Failure trace semantics

In failure trace semantics, the observer not only identifies processes based on which actions are blocked in the final state of an execution but also analyzes the sets of actions that were not possible throughout the entire execution of the system. This allows the observer to not only distinguish processes based on blocked behavior at the end of an execution but also to impose limitations on the behavior of each process over time. Example:...

Definition 3.3.1

The modal characterization of failure trace semantics $\mathcal{O}_FT$ *is defined recursively:*

$$\langle a \rangle \varphi \text{ if } \varphi \in \mathcal{O}_F$$

$$\bigwedge_{i \in I} \neg \langle a \rangle \top$$

```
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

definition hml_failure_trace_formulas
  where
hml_failure_trace_formulas ≡ {φ. hml_failure_trace φ}
```

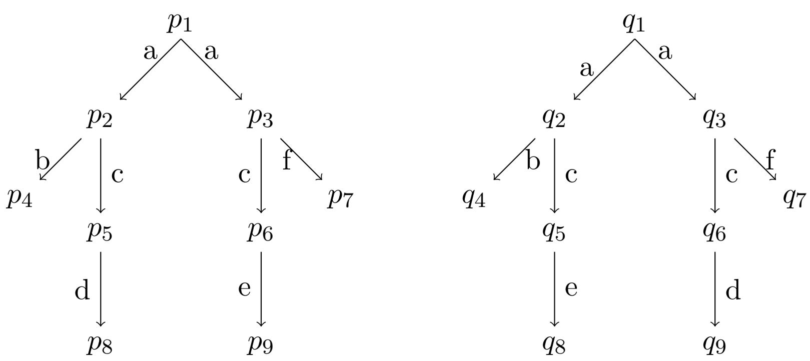

Figure 3.1: Graphs p and q

```
definition expr_failure_trace
  where
expr_failure_trace = {φ. (less_eq_t (expr φ) (∞, ∞, ∞, 0, 1, 1))}

context lts
begin

definition expr_failure_trace_equivalent
  where
expr_failure_trace_equivalent p q ≡ (∀ φ. φ ∈ expr_failure_trace ⟶
(p ⊨ φ) ⟷ (q ⊨ φ))
end
```

Proposition. $p \sim_{\mathrm{FT}} q \longleftrightarrow p \sim_{e_{\mathrm{FT}}}$

Proof.
We first establish the modal characterization of $\mathcal{O}_{e_{\mathrm{FT}}}$.
Since $\mathsf{expr}(\langle a\rangle) = (1,0,0,0,0,0) + \mathsf{expr}(\varphi)$, $\langle a\rangle\varphi$ belongs to $\mathcal{O}_{e_{\mathrm{FT}}}$ if φ is in $\mathcal{O}_{e_{\mathrm{FT}}}$.
For $\bigwedge_{i\in I}\psi_i$, we investigate the syntactic constraints the price bound imposes onto each ψ_i.
Since expr_1 is unbounded, expr_3 and expr_4 together uniquely determine the modal depth of the positive conjuncts. One positive conjunct may contain

arbitrary observations in its syntax tree. The modal depth of the others is bounded by 0, indicating that they consist solely of nested conjunctions. The other dimensions of the positive conjunctions are limited by the same bounds $\bigwedge_{i\in I}\psi_i$ is limited by.

The nesting depth of negations expr_6 of $\bigwedge_{i\in I}\psi_i$ is bounded by one. Since the negative conjuncts ψ_i take the form $\neg\varphi$, the corresponding φ's must not have any negations. Consequently, no conjunction within φ can include any negative conjuncts. expr_5 bounds the modal depth of the negative conjuncts by 1.

In summary, we have one positive conjunct r with $\mathsf{expr}(r) \leq (\infty,\infty,\infty,0,1,1)$, while all other positive conjuncts ψ_i are bounded by $\mathsf{expr}(\psi_i) \leq (0,\infty,0,0,0,1)$. The negative conjuncts ψ_j are bounded by $\mathsf{expr}(\psi_i) \leq (1,\infty,1,0,0,0)$.

These bounds give rise to subsets themselves, and we derive their modal characterization in a similar manner.

The recursive definition of the modal characterization is given by:

$\mathcal{O}_{FT_{x_1}}$:

$$\bigwedge_{i\in I}\varphi_i \text{ with } \varphi_i \in \mathcal{O}_{FT_x}$$

$\mathcal{O}_{FT_{x_2}}$:

$$\bigwedge_{i\in I}\psi_i$$
$$\psi_i \coloneqq \varphi \text{ with } \varphi \in \mathcal{O}_{FT_{x_2}} \mid \neg\varphi \text{ with } \varphi \in \mathcal{O}_{FT_{x_1}}$$

For any $\alpha, \varphi :$ if $\mathcal{O}_{e_{\mathrm{FT}}}(\varphi)$ then $\mathcal{O}_{e_{\mathrm{FT}}}(\text{hml\_pos } \alpha\varphi)$

For any I, J, Φ :

$$\begin{aligned}&\text{if } (\exists\psi \in (\Phi \circ I).\,(\mathcal{O}_{e_{\mathrm{FT}}}(\psi) \land \forall y \in (\Phi \circ I).\psi \neq y \Rightarrow \text{ nested\_empty\_conj } y)\\ &\qquad\qquad \lor (\forall y \in (\Phi \circ I).\text{ nested\_empty\_conj } y))\\ &\quad \land (\forall y \in (\Phi \circ J).\text{ stacked\_pos\_conj\_pos } y)\\ &\text{then } \mathcal{O}_{e_{\mathrm{FT}}}(\text{hml\_conj } IJ\Phi)\end{aligned}$$

```
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 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)

lemma expr_nested_empty_pos_conj:
  assumes nested_empty_pos_conj φ
  shows less_eq_t (expr φ) (0, ∞, 0, 0, 0, 0)
  ⟨proof⟩

context lts begin
lemma HML_true_nested_empty_pos_conj:
  assumes nested_empty_pos_conj φ
  shows HML_true φ
  ⟨proof⟩
end

lemma expr_nested_empty_conj:
  assumes nested_empty_conj φ
  shows less_eq_t (expr φ) (0, ∞, 0, 0, 0, 1)
  ⟨proof⟩

lemma expr_stacked_pos_conj_pos:
  assumes stacked_pos_conj_pos φ
  shows less_eq_t (expr φ) (1, ∞, 1, 0, 0, 0)
  ⟨proof⟩
```

```
lemma failure_trace_right:
  assumes (HML_failure_trace φ)
  shows (less_eq_t (expr φ) (∞, ∞, ∞, 0, 1, 1))
  ⟨proof⟩

lemma expr_6_conj:
  assumes (Φ ` J) ≠ {}
  shows expr_6 (hml_conj I J Φ) ≥ 1
⟨proof⟩

lemma expr_1_expr_6_le_0_is_nested_empty_pos_conj:
  assumes expr_1 φ ≤ 0 expr_6 φ ≤ 0
  shows nested_empty_pos_conj φ
  ⟨proof⟩

lemma expr_5_restrict_negations:
  assumes expr_5 (hml_conj I J Φ) ≤ 1 expr_6 (hml_conj I J Φ) ≤ 1
expr_4 (hml_conj I J Φ) ≤ 0
  shows (∀y ∈ (Φ ` J). stacked_pos_conj_pos y)
⟨proof⟩

lemma expr_1_0_expr_6_1_nested_empty_conj:
assumes expr_1 φ ≤ 0 expr_6 φ ≤ 1
shows nested_empty_conj φ
  ⟨proof⟩

lemma expr_4_expr_6_ft_to_recursive_ft:
  assumes expr_4 (hml_conj I J Φ) ≤ 0 expr_5 (hml_conj I J Φ) ≤ 1
expr_6 (hml_conj I J Φ) ≤ 1 ∀φ ∈ (Φ ` I). HML_failure_trace φ
  shows (∃ψ ∈ (Φ ` I). (HML_failure_trace ψ) ∧ (∀y ∈ (Φ ` I). ψ ≠
y ⟶ nested_empty_conj y)) ∨
(∀y ∈ (Φ ` I). nested_empty_conj y)
⟨proof⟩

lemma failure_trace_left:
  assumes (less_eq_t (expr φ) (∞, ∞, ∞, 0, 1, 1))
  shows (HML_failure_trace φ)
  ⟨proof⟩

lemma ft_lemma:
  shows (HML_failure_trace φ) = (less_eq_t (expr φ) (∞, ∞, ∞, 0, 1,
1))
  ⟨proof⟩
```

BlaBla... Dann Induktion über die Formeln und für jede formel äquivalente formel erstellen.

```
context lts begin
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 Readiness
imports Transition_Systems HML formula_prices_list Failures
begin
```

Readiness semantics

Readiness semantics provides a finer distinguishing power than failures by not only considering the actions that a system refuses after a given sequence of actions, but explicitly modeling the actions the system can engage in. (Figure ...) highlights the difference, between p_1 and q_1, no matter which actions the observer refuses at whatever point in the execution, the other process has an execution path that is indistinguishable. The processes are failures and failure trace equivalent. However, unlike p_1, q_1 can transition to a state q_3 where it is ready to perform actions a and b.

Definition The language $\mathcal{O}_R$ of readiness-formulas is defined recursively:

$$\langle a\rangle\varphi \text{ if } \varphi \in \mathcal{O}_R \mid \bigwedge_{i\in I} \neg\langle a\rangle\mathsf{T}$$

```
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 ψs)
if ∀i ∈ I. (∃α. ((ψs i) = hml_pos α TT)) (∀j ∈ J. (∃α. ((ψs j) = hml_pos
α TT)) ∨ ψs j = TT)
```

```
definition hml_readiness_formulas
  where
hml_readiness_formulas ≡ {φ. hml_readiness φ}
```

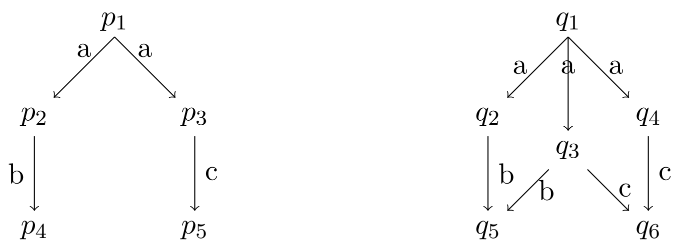

Figure 3.2: TEEEEEEEEEEEEEEEEEEEST

```
definition expr_ready_trace
  where
expr_ready_trace = {φ. (less_eq_t (expr φ) (∞, ∞, ∞, 1, 1, 1))}

context lts
begin

definition expr_ready_trace_equivalent
  where
expr_ready_trace_equivalent p q ≡ (∀ φ. φ ∈ expr_ready_trace ⟶ (p
⊨ φ) ⟷ (q ⊨ φ))
```

Proposition

```
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))


lemma readiness_right:
  assumes A1: HML_readiness φ
  shows (less_eq_t (expr φ) (∞, 2, 1, 1, 1, 1))
  ⟨proof⟩

lemma expr_2_expr_3_restrict_positives:
  assumes (expr_2 (hml_conj I J Φ)) ≤ 2 (expr_3 (hml_conj I J Φ)) ≤
1
  shows (∀x ∈ (Φ ` I). TT_like x ∨ (∃α χ. x = hml_pos α χ ∧ TT_like
χ))
⟨proof⟩
```

```
lemma readiness_left:
  assumes (less_eq_t (expr φ) (∞, 2, 1, 1, 1, 1))
  shows HML_readiness φ
  ⟨proof⟩

lemma readiness_lemma:
  shows (HML_readiness φ) = (less_eq_t (expr φ) (∞, 2, 1, 1, 1, 1))
  ⟨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⟩


end
end
theory Ready_traces
imports Transition_Systems HML formula_prices_list Failure_traces Readiness
begin
```

Failures semantics

```
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 ((∀i ∈ Φ ` I. ∃α. i = hml_pos α TT) ∨ (∃i ∈ Φ ` I. hml_ready_trace
i ∧ (∀j ∈ Φ ` I. i ≠ j ⟶ (∃α. j = hml_pos α TT))))
     ∀j ∈ Φ ` J. ∃α. j = (hml_pos α TT) ∨ j = TT

definition hml_ready_trace_formulas
  where
hml_ready_trace_formulas ≡ {φ. hml_ready_trace φ}
```

```
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


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)


definition expr_readiness
  where
expr_readiness = {φ. (less_eq_t (expr φ) (∞, 2, 1, 1, 1, 1))}

context lts
begin

definition expr_readiness_equivalent
  where
expr_readiness_equivalent p q ≡ (∀ φ. φ ∈ expr_readiness ⟶ (p ⊨ φ)
⟷ (q ⊨ φ))

end

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 = {}


lemma expr_stacked_pos_conj:
  assumes stacked_pos_conj φ
  shows less_eq_t (expr φ) (1, ∞, 1, 1, 1, 2)
  ⟨proof⟩

lemma expr_single_pos_pos:
  assumes single_pos_pos φ
  shows less_eq_t (expr φ) (1, ∞, 1, 1, 0, 0)
  ⟨proof⟩

lemma expr_single_pos:
  assumes single_pos φ
  shows less_eq_t (expr φ) (1, ∞ , 1, 1, 1, 1)
  ⟨proof⟩

lemma single_pos_pos_expr:
  assumes expr_1 φ ≤ 1 expr_6 φ ≤ 0
  shows single_pos_pos φ
  ⟨proof⟩

lemma single_pos_expr:
assumes expr_5 φ ≤ 1 expr_6 φ ≤ 1
expr_1 φ ≤ 1
shows single_pos φ
  ⟨proof⟩

lemma stacked_pos_conj_right:
  assumes expr_5 (hml_conj I J Φ) ≤ 1 expr_6 (hml_conj I J Φ) ≤ 1
expr_4 (hml_conj I J Φ) ≤ 1 ∀φ ∈ (Φ ` I). HML_ready_trace φ
shows (∃x ∈ (Φ ` I). HML_ready_trace x ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ single_pos
y))
∨ (∀y ∈ (Φ ` I).single_pos y)
⟨proof⟩

lemma stacked_pos_conj_left:
  assumes expr_5 (hml_conj I J Φ) ≤ 1 expr_6 (hml_conj I J Φ) ≤ 1
expr_4 (hml_conj I J Φ) ≤ 1
shows (∀y ∈ (Φ ` J). single_pos_pos y)
⟨proof⟩
```

```
lemma ready_trace_right:
  assumes HML_ready_trace φ
  shows less_eq_t (expr φ) (∞, ∞, ∞, 1, 1, 1)
  ⟨proof⟩

lemma ready_trace_left:
  assumes less_eq_t (expr φ) (∞, ∞, ∞, 1, 1, 1)
  shows HML_ready_trace φ
  ⟨proof⟩
end
theory Revivals
imports Transition_Systems HML formula_prices_list Failures Expr_helper
begin
```

Readiness semantics

```
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 ψs)
if ∀i ∈ I. (∃α. ((ψs i) = hml_pos α TT)) (∀j ∈ J. (∃α. ((ψs j) = hml_pos
α TT)) ∨ ψs j = TT)

definition hml_readiness_formulas
  where
hml_readiness_formulas ≡ {φ. hml_readiness φ}
```

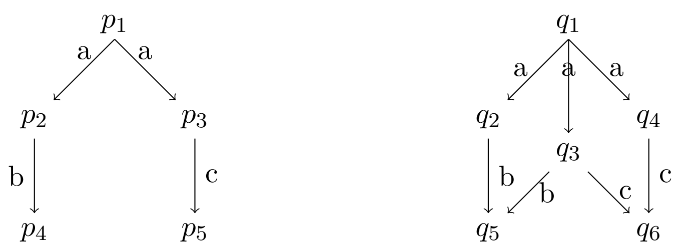

Figure 3.3: TEEEEEEEEEEEEEEEEEEEST

```
definition expr_ready_trace
  where
expr_ready_trace = {φ. (less_eq_t (expr φ) (∞, ∞, ∞, 1, 1, 1))}

context lts
begin
```

```
definition expr_ready_trace_equivalent
  where
expr_ready_trace_equivalent p q ≡ (∀ φ. φ ∈ expr_ready_trace ⟶ (p
⊨ φ) ⟷ (q ⊨ φ))
```

Proposition

```
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) ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ TT_like y))
∨ (∀y ∈ (Φ ` I).TT_like y)
(∀x ∈ (Φ ` J). TT_like x ∨ (∃α χ. (x = hml_pos α χ) ∧ TT_like χ))
lemma revivals_right:
  assumes HML_revivals φ
  shows less_eq_t (expr φ) (∞, 2, 1, 0, 1, 1)
  ⟨proof⟩

lemma pos_r_apply:
  assumes ∀x ∈ (pos_r (Φ ` I)). expr_1 x ≤ n ∀x ∈ Φ ` I. expr_1 x ≤
m
  shows ∀x ∈ (Φ ` I). expr_1 x ≤ n ∨ (∃x ∈ Φ ` I. expr_1 x ≤ m ∧ (∀y
∈ Φ ` I. y ≠ x ⟶ expr_1 y ≤ n))
⟨proof⟩

lemma e1_le_0_e2_le_1:
  assumes expr_1 φ ≤ 0 expr_2 φ ≤ 1
  shows TT_like φ
  ⟨proof⟩

lemma e1_le_1_e2_le_1:
  assumes expr_1 φ ≤ 1 expr_2 φ ≤ 1
  shows TT_like φ ∨ (∃α ψ. φ = (hml_pos α ψ) ∧ TT_like ψ)
  ⟨proof⟩

lemma revivals_pos:
  assumes less_eq_t (expr (hml_conj I J Φ)) (∞, 2, 1, 0, 1, 1)
  shows (∃x ∈ (Φ ` I). (∃α χ. (x = hml_pos α χ) ∧ TT_like χ) ∧ (∀y
∈ (Φ ` I). x ≠ y ⟶ TT_like y))
∨ (∀y ∈ (Φ ` I).TT_like y)
⟨proof⟩

lemma revivals_left:
  assumes less_eq_t (expr φ) (∞, 2, 1, 0, 1, 1)
  shows HML_revivals φ
⟨proof⟩

end
```

```
end
theory Impossible_futures
imports Transition_Systems HML formula_prices_list Failure_traces
begin
```

Failures semantics

```
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)

definition hml_impossible_futures_formulas
  where
hml_impossible_futures_formulas ≡ {φ. hml_impossible_futures φ}

definition expr_impossible_futures
  where
expr_impossible_futures = {φ. (less_eq_t (expr φ) (∞, 2, 0, 0, ∞, 1))}

context lts
begin

definition expr_impossible_futures_equivalent
  where
expr_impossible_futures_equivalent p q ≡ (∀ φ. φ ∈ expr_impossible_futures
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))
end
```

Proposition

```
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)

lemma impossible_futures_right:
  assumes A1: HML_impossible_futures φ
  shows less_eq_t (expr φ) (∞, 2, 0, 0, ∞, 1)
  ⟨proof⟩

lemma e6_e5_le_0:
  assumes expr_6 φ ≤ 0
```

```
  shows expr_5 φ ≤ 0
  ⟨proof⟩


lemma e5_e6_ge_1:
  fixes φ
  assumes expr_5 φ ≥ 1
  shows expr_6 φ ≥ 1
  ⟨proof⟩

lemma expr_2_le_2_is_trace:
  assumes expr_2 (hml_conj I J Φ) ≤ 2
  shows ∀x ∈ (Φ ` I ∪ Φ ` J). (hml_trace x)
⟨proof⟩

lemma impossible_futures_left:
  assumes less_eq_t (expr φ) (∞, 2, 0, 0, ∞, 1)
  shows HML_impossible_futures φ
  ⟨proof⟩

lemma impossible_futures_lemma:
  shows HML_impossible_futures φ = less_eq_t (expr φ) (∞, 2, 0, 0, ∞,
1)
  ⟨proof⟩

context lts begin
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⟩

end
end
theory Possible_futures
imports Transition_Systems HML formula_prices_list Impossible_futures
begin
```

Failures semantics

```
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}

definition expr_possible_futures
  where
expr_possible_futures = {φ. (less_eq_t (expr φ) (∞, 2, ∞, ∞, ∞, 1))}

context lts
begin

definition expr_possible_futures_equivalent
  where
expr_possible_futures_equivalent p q ≡ (∀ φ. φ ∈ expr_possible_futures
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))
end
lemma possible_futures_right:
  assumes hml_possible_futures φ
  shows less_eq_t (expr φ) (∞, 2, ∞, ∞, ∞, 1)
  ⟨proof⟩

lemma possible_futures_left:
  assumes less_eq_t (expr φ) (∞, 2, ∞, ∞, ∞, 1)
  shows hml_possible_futures φ
  ⟨proof⟩

lemma possible_futures_lemma:
  shows hml_possible_futures φ = less_eq_t (expr φ) (∞, 2, ∞, ∞, ∞,
1)
  ⟨proof⟩

end
theory Simulation
imports Transition_Systems HML formula_prices_list Traces
begin
```

Failures semantics

```
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}

definition expr_simulation
  where
expr_simulation = {φ. (less_eq_t (expr φ) (∞, ∞, ∞, ∞, 0, 0))}

context lts
begin

definition expr_simulation_equivalent
  where
expr_simulation_equivalent p q ≡ (∀ φ. φ ∈ expr_simulation ⟶ (p ⊨
φ) ⟷ (q ⊨ φ))
end

lemma simulation_right:
  assumes hml_simulation φ
  shows (less_eq_t (expr φ) (∞, ∞, ∞, ∞, 0, 0))
  ⟨proof⟩


lemma Max_eq_expr_6:
  fixes x1 x2
  defines DA: A ≡ {0} ∪ {expr_6 xa |xa. xa ∈ set x1} ∪ {1 + expr_6 ya
|ya. ya ∈ set x2}
  defines DB: B ≡ {0} ∪ {expr_6 xa |xa. xa ∈ set x1} ∪ {Max ({0} ∪ {1
+ expr_6 ya |ya. ya ∈ set x2})}
  shows Max A = Max B
⟨proof⟩

lemma x2_empty:
  assumes (less_eq_t (expr (hml_conj I J Φ)) (∞, ∞, ∞, ∞, 0, 0))
  shows (Φ ` J) = {}
⟨proof⟩

lemma simulation_left:
  assumes (less_eq_t (expr φ) (∞, ∞, ∞, ∞, 0, 0))
  shows (hml_simulation φ)
  ⟨proof⟩

end
theory Two_nested_sim
imports Transition_Systems HML formula_prices_list Simulation
begin
```

Failures semantics

```
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)

definition hml_2_nested_sim_formulas where
hml_2_nested_sim_formulas ≡ {φ. hml_2_nested_sim φ}

definition expr_2_nested_sim
  where
expr_2_nested_sim = {φ. (less_eq_t (expr φ) (∞, ∞, ∞, ∞, ∞, 1))}

context lts
begin

definition expr_2_nested_sim_equivalent
  where
expr_2_nested_sim_equivalent p q ≡ (∀ φ. φ ∈ expr_2_nested_sim ⟶ (p
⊨ φ) ⟷ (q ⊨ φ))
end

lemma nested_sim_right:
  assumes hml_2_nested_sim φ
  shows less_eq_t (expr φ) (∞, ∞, ∞, ∞, ∞, 1)
⟨proof⟩

lemma e5_e6_ge_1:
  fixes φ
  assumes expr_5 φ ≥ 1
  shows expr_6 φ ≥ 1
  ⟨proof⟩

lemma nested_sim_left:
  assumes less_eq_t (expr φ) (∞, ∞, ∞, ∞, ∞, 1)
  shows hml_2_nested_sim φ
⟨proof⟩

end
```

Bibliography

[Bis23] Benjamin Bisping. Process equivalence problems as energy games, 2023. arXiv:2303.08904.

[BJN22] Benjamin Bisping, David N. Jansen, and Uwe Nestmann. Deciding all behavioral equivalences at once: A game for linear-time–branching-time spectroscopy, 2022. arXiv:2109.15295.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137––161, 1985. doi:10.1145/2455.2460.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, volume 13, pages 477–498. Springer Berlin Heidelberg, 1985. doi:10.1007/978-3-642-82453-1_17.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. URL: https://doi.org/10.1145/360248.360251, doi:10.1145/360248.360251.

[vG01] R.J. van Glabbeek. Chapter 1 - the linear time - branching time spectrum i. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier Science, Amsterdam, 2001. doi:https://doi.org/10.1016/B978-044482830-9/50019-9.

Appendix A

Alternative price function

Den Rest in den Appendix:

The expressiveness price $\mathsf{expr} : HML[\Sigma] \to (\mathbb{N} \cup \infty)^6$ *of a formula interpreted as* 6×1*-dimensional vectors is defined recursively by:*

$$\mathrm{expr}(\langle a \rangle \varphi) := \begin{pmatrix} 1 + \mathrm{expr}_1(\varphi) \\ \mathrm{expr}_2(\varphi) \\ \mathrm{expr}_3(\varphi) \\ \mathrm{expr}_4(\varphi) \\ \mathrm{expr}_5(\varphi) \\ \mathrm{expr}_6(\varphi) \end{pmatrix} \qquad \mathrm{expr}(\neg\varphi) := \begin{pmatrix} \mathrm{expr}_1(\varphi) \\ \mathrm{expr}_2(\varphi) \\ \mathrm{expr}_3(\varphi) \\ \mathrm{expr}_4(\varphi) \\ \mathrm{expr}_5(\varphi) \\ 1 + \mathrm{expr}_6(\varphi) \end{pmatrix}$$

$$\mathrm{expr}\left(\bigwedge_{i \in I} \psi_i\right) := \sup(\{\begin{pmatrix} 0 \\ 1 + \sup_{i \in I} \mathrm{expr}_2(\psi_i) \\ \sup_{i \in \mathrm{Pos}} \mathrm{expr}_1(\psi_i) \\ \sup_{i \in \mathrm{Pos} \setminus \mathcal{R}} \mathrm{expr}_1(\psi_i) \\ \sup_{i \in \mathrm{Neg}} \mathrm{expr}_1(\psi_i) \\ 0 \end{pmatrix}\} \cup \{\mathsf{expr}(\psi_i) | i \in I\}$$

Remark: The only deviation from [Bis23] (Definition 5) is, that we include infinity in the range of the function since we allow for infinite branching conjunctions.

It turned out that it was easier to also define the price for every dimension i as a seperate function $\mathsf{expr} : \mathrm{HML}[\Sigma] \to (\mathbb{N} \cup \infty)$.

Our Isabelle definition of HML makes it very easy to derive the sets Pos and Neg, by `Φ ` I` and `Φ ` J` respectively.

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...

Now we can directly define the expressiveness function as `direct_expr`.

```
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⟩
```

In order to demonstrate termination of the function, it is necessary to establish that each sequence of recursive function calls reaches a base case. This is accomplished by proving that the relation between process-formula pairs, as defined recursively by the function, is contained within a well-founded relation. A relation $R \subset X \times X$ is considered well-founded if every non-empty subset $X' \subset X$ contains a minimal element m such that $(x, m) \notin R$ for all $x \in X'$. A key property of well-founded relations is that all descending chains $(x_0, x_1, x_2, \ldots)$ (where $(x_i, x_{i+1}) \in R$) originating from any element $x_0 \in X$ are finite. Consequently, this ensures that each sequence of recursive invocations terminates after a finite number of steps.

These proofs were inspired by the Isabelle formalizations presented in [WEP+16].

```
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⟩
```

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

termination
  ⟨proof⟩
```

We show that `direct_expr` and `expr` are the same:

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