Questions and Answers for PSC

Innocenzo Fulginiti, Gabriele Pappalardo

February-June 2022

Answered Questions

1) What is the termination property?
The termination property ensures that for each expression $\mathrm{E} \in \mathrm{Exp}$ exists a value $n \in \mathbb{N}$ such that E is evaluated as n.

2) What is the formula to express termination in operational semantics?
For the Small-step semantics we have:

$$\forall\, \mathrm{E}.\, \exists\, n.\, \mathrm{E} \to^* n$$

For the Big-step semantics we have:

$$\forall\, \mathrm{E}.\, \exists\, n.\, \mathrm{E} \longrightarrow n$$

3) What is the formula to express termination in denotational semantics?
Given a semantic domain $\mathbb{N}$ and an interpretation function $\mathcal{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$, the formula is

$$\forall\, \mathrm{E}.\, \exists\, n.\, \mathcal{E}[\![E]\!] = n$$

4) What is the determinacy property?
If we can evaluate an expression with n and m, then n and m must be equal.

5) What is the formula to express determinacy in operational semantics?
For the Small-step semantics we have:

$$\forall\, \mathrm{E},\, n,\, m.\, (\mathrm{E} \to^* n \wedge \mathrm{E} \to^* m) \implies n = m$$

For the Big-step semantics we have:

$$\forall\, \mathrm{E},\, n,\, m.\, (\mathrm{E} \longrightarrow n \wedge \mathrm{E} \longrightarrow m) \implies n = m$$

6) What is the formula to express determinacy in denotational semantics?
Given a semantic domain $\mathbb{N}$ and an interpretation function $\mathcal{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$, the formula is

$$\forall\, \mathrm{E},\, n,\, m.\, (\mathcal{E}[\![E]\!] = n \wedge \mathcal{E}[\![E]\!] = m) \implies n = m$$

7) What does it mean equivalence between programs?
Two programs E_0, E_1 are equivalents if they are evaluated with the same semantic value. We indicate the equivalence between two programs as $\mathrm{E}_0 \equiv \mathrm{E}_1$.

8) What is the formula to express equivalence in operational semantics?
The formula to express equivalence in small-step operational semantics, $\mathrm{E}_0 \equiv_s \mathrm{E}_1$, is

$$\forall n.(\mathrm{E}_0 \to^* n \iff \mathrm{E}_1 \to^* n)$$

The formula to express equivalence in big-step operational semantics, $\mathrm{E}_0 \equiv_b \mathrm{E}_1$, is

$$\forall n.(\mathrm{E}_0 \longrightarrow n \iff \mathrm{E}_1 \longrightarrow n)$$

9) What is the formula to express equivalence in denotational semantics?
The formula to express equivalence in denotational semantics, $\mathrm{E}_0 \equiv_d \mathrm{E}_1$, is

$$\mathcal{E}[\![E_0]\!] = \mathcal{E}[\![E_1]\!]$$

10) What is the difference between small-step and big-step semantics?
In big-step semantics, we have that a complete computation is done as a single, large derivation. Often, it can correspond to an efficient implementation for an interpreter of a language. Small-step semantics formally describes how individual steps of a computation take place on an abstract device, but ignore details. This kind of semantics give more control of the evaluation details and the order of the evaluation.

11) How to state the consistency between different semantics?
We express the consistency between different semantics as

$$\forall \mathrm{E}, n.(\mathrm{E} \to^* n \iff \mathrm{E} \longrightarrow n \iff \mathcal{E}[\![E]\!] = n)$$

12) What is a congruence?
An equivalence can also be a congruence. We have a **congruence** when we can replace an equivalent term within a bigger expression, without changing the meaning of the expression.

Definition 1 (Congruence) *An equivalence $\equiv$ is said to be a **congruence** (with respect to a class of contexts $C[\cdot]$) if:*

$$\forall C[\cdot].\ p \equiv q \implies C[p] \equiv C[q]$$

13) How to verify that an equivalence is a congruence?
For small-step operational semantics, must be proved that

$$\forall E_0, E_1, C[\cdot].(E_0 \equiv_s E_1 \implies C[E_0] \equiv_s C[E_1])$$

For big-step operational semantics, must be proved that

$$\forall E_0, E_1, C[\cdot].(E_0 \equiv_b E_1 \implies C[E_0] \equiv_b C[E_1])$$

For denotational semantics, it's obvious that

$$\forall E_0, E_1, C[\cdot].(E_0 \equiv_d E_1 \implies C[E_0] \equiv_d C[E_1])$$

14) What is the compositionality principle for denotational semantics?
The meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings.

15) What is a grammar?
A **grammar** describes how to form strings from a language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form. A formal grammar is defined as a set of production rules for such strings in a formal language.

16) What is a signature?
A signature is an arity-indexed family of sets of operators:

$$\Sigma = \{\Sigma_n\}_{n \in \mathbb{N}}$$

17) What are the terms generated by a signature?
One-sorted / Unsorted, given:

$$\Sigma = \{\Sigma_n\}_{n \in \mathbb{N}} \text{ (a signature)}$$

$$X = \{x, y, z, ...\} \text{ (a set of variables)}$$

we denote with $T_{\Sigma,X}$ the set of all terms over Σ, X, defined as the least set such that:

- if $x \in X$, then $x \in T_{\Sigma,X}$
- if $c \in \Sigma_0$, then $c \in T_{\Sigma,X}$
- if $f \in \Sigma_n$ and $t_1, ..., t_n \in T_{\Sigma,X}$, then $f(t_1, ..., t_n) \in T_{\Sigma,X}$

Many-sorted, given:

$$S = \{s_1, s_2, ...\} \text{ (a set of sorts)}$$

$$\Sigma = \{\Sigma_{s_1,...,s_n,s \in S}\}_{s_1,...,s_n,s \in S} \text{ (a many-sorted signature)}$$

$T_{\Sigma,s}$ denotes the set of all terms of sort s and it is defined as least set such that:

- if $c \in \Sigma_{\epsilon,s}$, then $c \in T_{\Sigma,s}$
- if $f \in \Sigma_{s_1,...,s_n,s \in S}$ and $\forall i.t_i \in T_{\Sigma,s_i}$, then $f(t_1, ..., t_n) \in T_{\Sigma,s}$

$T_{\Sigma,s_{s \in S}}$ denotes the set of all well-sorted terms.

18) What is the difference between an unsorted signature and a many sorted one?
In una signature many-sorted, oltre a specificare un'arietà, si specifica un "tipo" per i vari termini. Una signature one-sorted è equivalente ad una signature unsorted. Nelle signature unsorted abbiamo la seguente relazione di sottotermine:

$$t_i \prec f(t_1 ... t_n)$$

Nelle signature many-sorted, delle volte, si può aggiungere un ulteriore vincolo per la relazione di sottotermine, cioè si può richiedere che:

$$f \in \Sigma s_0, ..., s_n, s \wedge s_i = s$$

19) What is a type system?
A **type system** is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a program, such as variables, expressions, functions or modules.

(*From: "Models of Computation" by Roberto Bruni and Ugo Montanari*) Types are often presented as sets of logic rules, called **type systems**, that are used to assign a type unambiguously to each program and computed value.

20) What is an inference rule?
Una **regola di inferenza** è uno schema formale che si applica nell'eseguire un'inferenza. In altre parole, è una regola che permette di passare da un numero finito di proposizioni assunte come premesse (anche nessuna nel caso degli assiomi) a una proposizione che funge da conclusione.

(*From: "Models of Computation" by Roberto Bruni and Ugo Montanari*).

Definition 2 (Inference Rule) *Let $x_1, \ldots, x_n, y$ be (well-formed, i.e. not ambiguous) formulas. An **inference rule** is written, using the notation:*

$$r = \frac{x_1, x_2, \ldots, x_n}{y}$$

*Where $\{x_1, x_2, \ldots, x_n\}$ is the set of **premises** and y is the **conclusion**.*

The meaning of a such rule r is that if we can prove all formulas $x_1, \ldots, x_n$ in our logical system, then by exploiting the inference rule r we can also derive the validity of y.

21) What is a logical system?
A **logical system** is just a set of axioms and inference rules. If an inference rule contains some variables, we assume all its instances are in R.

22) What is a derivation?
A **derivation** is a *proof tree* (whose leaves are axioms). Given a logical system R, a derivation in R, is written:

$$d \models_R y$$

where d can be:

$$d = \frac{}{y} \in R \text{ (an axiom of R)}$$

$$d = \frac{d_1, ..., d_n}{y} \text{ for some derivations } d_1 \models_R y_1, ..., d_n \models_R y_n$$
$$\text{such that } \frac{x_1, ..., x_n}{y} \in R \text{ is an inference rule of R.}$$

23) What is a theorem in a logical system?
A **theorem**, in a logical system R, is a well-formed formula y for which there exists a proof, and we write $\models_R y$. In other words, y is a theorem in R if $\exists d.d \models_R y$ (there exists a derivation in our logical system R that can be used to conclude y).

The set of all theorems of R is denoted by:

$$I_R = \{y \mid \models_R y\}$$

24) What is the unification problem?
The **unification problem** in its simplest form (syntactic, first-order) can be expressed as follows: given a set of potential equalities

$$\mathcal{G} = [l_1 \stackrel{?}{=} r_1, ..., l_n \stackrel{?}{=} r_n]$$

where $l_1, ..., l_n, r_1, ..., r_n \in T_{\Sigma,X}$. Can we find a substitution function $\rho : T_{\Sigma,X} \to T_{\Sigma,X}$ such that:

$$\forall i \in [1...n].\rho(l_i) = \rho(r_i) \ ?$$

We call such solution ρ of $\mathcal{G}$. The set of all the solutions of $\mathcal{G}$ is the following one:

$$\text{sols}(\mathcal{G}) = \{\rho \mid \forall i \in [1...n].\rho(l_i) = \rho(r_i)\}$$

Note that one solution exists is not necessarily unique. Moreover, it may happen that a solution does not exist at all.

25) What are the rules to solve the unification problem?
To solve the unification problem, we apply the *unification algorithm*. The algorithm takes as input a set of potential equalities $\mathcal{G}$ and transform it until:

1. either it terminates (no transformation can be applied any more) after having transformed $\mathcal{G}$ into an equivalent set of equalities;
2. or it fails, meaning that the potential equalities cannot be unified.

Thus, given a set of potential equalities:

$$\mathcal{G} = \{l_1 \stackrel{?}{=} r_1, ..., l_n \stackrel{?}{=} r_n\}$$

we have the following rules:

1. **delete**: $\mathcal{G} \cup \{t \stackrel{?}{=} t\}$ becomes $\mathcal{G}$;
2. **swap**: $\mathcal{G} \cup \{f(t_1, ..., t_n) \stackrel{?}{=} x\}$ becomes $\mathcal{G} \cup \{x \stackrel{?}{=} f(t_1, ..., t_n)\}$;
3. **eliminate**: $\mathcal{G} \cup \{x \stackrel{?}{=} t\}$ with $x \in vars(\mathcal{G}) \setminus vars(t)$ becomes $\mathcal{G}[t = x] \cup \{x \stackrel{?}{=} t\}$;
4. **decompose**: $\mathcal{G} \cup \{f(t_1, ..., t_n) \stackrel{?}{=} f(u_1, ..., u_n)\}$ becomes $\mathcal{G} \cup \{t_1 \stackrel{?}{=} u_1, ..., t_n \stackrel{?}{=} u_n\}$;
5. **occur-check**: $\mathcal{G} \cup \{x \stackrel{?}{=} f(t_1, ..., t_n)\}$ fails if $x \in vars(f(t_1, ..., t_n))$;

6. **conflict**: $\mathcal{G} \cup \{f(t_1, ..., t_n) \stackrel{?}{=} g(u_1, ..., u_m)\}$ fails if $f \neq g \vee n \neq m$.

26) What is an infinite descending chain?
Given the set of elements $A = \{a_0, ..., a_n, a_{n+1}, ...\}$, an **infinite descending chain** is a sequence $\{a_n\}_{n \in \mathbb{N}}$ such that:

$$\forall i \in \mathbb{N}. a_i \succ a_{i+1}$$

27) What is a well-founded relation?
A relation is called **well-founded** if it has no infinite descending chain.

28) What is a minimal element?
Given a binary relation $\prec \subseteq A \times A$, let $Q \subseteq A$ and $m \in Q$ we say that m is **minimal** in Q if none of the elements in Q precedes m, i.e.

$$\forall x \in Q. x \not\prec m$$

29) What is the well-founded induction principle?
Given a w.f. relation $\prec \subseteq A \times A$, the w.f. **induction principle** can be defined with the following inference rule:

$$\frac{\forall a \in A.((\forall b \prec a.\ P(b)) \implies P(a))}{\forall a \in A.\ P(a)}$$

30) How do other induction principles are derived from well-founded induction?
Changing the underlying elements set A and the well-founded relation $\prec$.

31) What is mathematical induction?
Weak mathematical induction can be defined using well-founded induction, taking:

$$A = \mathbb{N}$$

$$\prec = \{(n, n+1) \mid n \in \mathbb{N}\} \text{ (immediate precedence relation)}$$

we obtain the following inference rule:

$$\frac{P(0) \quad \forall n \in \mathbb{N}.\ P(n) \implies P(n+1)}{\forall n \in \mathbb{N}.\ P(n)}$$

Strong mathematical induction can be defined using well-founded induction, taking:

$$A = \mathbb{N}$$

$$\prec = <$$

we obtain the following inference rule:

$$\frac{P(0) \quad \forall n \in \mathbb{N}.\ (P(0) \wedge ... \wedge P(n)) \implies P(n+1)}{\forall n \in \mathbb{N}.\ P(n)}$$

32) What is structural induction?
The **structural induction** can be defined using well-founded induction, taking:

$$A = T_\Sigma \text{ (closed terms, where } \Sigma = \{\Sigma\}_{n \in \mathbb{N}})$$

$$\prec = \{(t_i, f(t_1, ..., t_n)) \mid f \in \Sigma_n,\ i \in [1...n]\}$$

we obtain the following inference rule:

$$\frac{\forall f \in \Sigma_{s_1,...,s_n,s}.\forall t_1 \in T_{\Sigma,s_1,...}\forall t_n \in T_{\Sigma,s_n}.(P(t_1) \wedge \cdots \wedge P(t_n)) \implies P(f(t_1, \ldots, t_n))}{\forall t \in T_\Sigma.P(t)}$$

33) What is rule induction?
Give a logical system R, the **rule induction** can be defined using well-founded induction, taking:

$$A = D_R = \{d \mid d \models_R y\}$$

$$\prec = \left\{ \left(d_i, \frac{d_1, ..., d_n}{y} \right) \;\middle|\; d_1 \models_R y_1, ..., d_n \models_R y_n, \frac{x_1, ..., x_n}{y} \in R \right\}$$

we obtain the following inference rule:

$$\frac{\forall \frac{X}{y} \in R.(\forall x \in X.P(x)) \implies P(y)}{\forall x \in I_R.P(x)}$$

34) What is well-founded recursion?

Theorem 1 (Well-founded Recursion) *Given a w.f. relation $\prec$ on a domain A, we define the following family of functions:*

$$F \triangleq \{F_a : (\lfloor a \rfloor \to B) \to B\}_{a \in A}$$

so, we have that:

$$\forall a \in A.\forall h \in \lfloor a \rfloor \to B.F_a(h) \in B$$

A recursive definition of a function f is **well-founded** when the recursive calls to f take as arguments values that are smaller with regard to the ones taken by the defined function.

35) What is a partial order?
A **partial order** $(P, \sqsubseteq_P)$ is a pair made by a set P and a relation $\sqsubseteq_P \subseteq P \times P$ over it, that satisfy the following properties:

1. *Reflexivity*: $\forall p \in P.p \sqsubseteq_P p$
2. *Antisymmetric*: $\forall p, q \in P.p \sqsubseteq_P q \wedge q \sqsubseteq_P p \implies p = q$
3. *Transitivity*: $\forall p, q, r \in P.p \sqsubseteq_P q \wedge q \sqsubseteq_P r \implies p \sqsubseteq_P r$

36) What is a total order?
A **total order** is a partial order where every pair of elements are **comparable**, that is:

$$\forall p, q \in P.p \neq q \wedge p \sqsubseteq_P q \vee q \sqsubseteq_P p$$

37) What is a discrete order?
A **discrete order** is a partial order where **any** element is **comparable** only to itself, that is:

$$\forall p, q \in P.p \sqsubseteq_P q \iff p = q$$

38) What is a flat order?
A **flat order** is a partial order where any element is **comparable** only to itself and with a distinguished (smaller) element $\bot$, that is:

$$\forall p, q \in P. p \sqsubseteq_P q \iff p = q \vee p = \bot$$

39) What is a chain?
A **chain** $\{d_i\}_{i \in \mathbb{N}}$ is a succession of elements such that:

$$\forall i \in \mathbb{N}. d_i \sqsubseteq d_{i+1}$$

A chain can be also seen as a function $d : \mathbb{N} \to P$, where to each natural number there is an element of $p \in P$ associated.

$$d(0) = d_0,\ d(1) = d_1,\ \ldots\ d(n) = d_n,\ \ldots$$

Be aware, the chain function builds up a set of the following form:

$$\{d_0, d_1, d_2, \ldots, d_n, \ldots\} \subseteq P$$

Which is the image set of the function d explained above.

40) What is a finite chain?
A **finite chain** is a chain in a PO if it is constituted of only finitely many *different* elements, that is:

$$\exists k \in \mathbb{N}.\ \forall i \geq k.\ d_i = d_{i+1}\ ,$$

or equivalently:

$$\exists k \in \mathbb{N}.\ \forall i \geq k.\ d_i = d_k\ .$$

A finite chain is still an infinite sequence of elements, but the number of different elements in the chain is finite. If the chain is not finite, we say that it is *infinite*.

41) What is the least element?
Given a PO $(P, \sqsubseteq)$, $Q \subseteq P$, $l \in Q$, l is the *least* element of Q if:

$$\forall q \in Q.\ l \sqsubseteq q$$

42) What is the bottom element?
Given a PO $(P, \sqsubseteq)$ the least element of P (if it exists) is called *bottom* and denoted $\bot$.

43) What is an upper bound?
Let $(P, \sqsubseteq)$ be a PO, consider $Q \subseteq P$ and $u \in P$, we say that u is an *upper bound* of Q if:

$$\forall q \in Q.\ q \sqsubseteq u\ .$$

Definition 3 (Least Upper Bound) *Let $(P, \sqsubseteq)$ be a PO, consider $Q \subseteq P$ and $p \in P$, we say that p is the least upper bound (lub) of Q if:*

1. *p is an upper bound of Q, that is*

$$\forall q \in Q.\, q \sqsubseteq p\ ,$$

2. *p is smaller than any other upper bound of Q, that is*

$$\forall u \in P.\, (\forall q \in Q.\, q \sqsubseteq u) \implies p \sqsubseteq u\ .$$

44) What is a limit?
Let $(P, \sqsubseteq)$ be a PO, let $\{d_i\}_{i\in\mathbb{N}}$ be a chain in $(P, \sqsubseteq)$, we define the *limit* as the least upper bound (if it exists) of the chain $\{d_i\}_{i\in\mathbb{N}}$ and we denote it as $\bigsqcup_{i\in\mathbb{N}} d_i$.

45) What is a complete partial order?
Let $(P, \sqsubseteq)$ be a PO, we say that it is a *complete partial order (CPO)*, if each chain in it has a limit.

46) How to prove that a partial order is complete?
To prove that a PO is a Complete Partial Order we must show that every chain in the PO has a limit (lub).

47) What is a monotone function?
Given two partial orders $(P, \sqsubseteq_P), (E, \sqsubseteq_E)$, a **monotone function** $f : D \to E$ is an *order preserving* function that satisfy the following property:

$$\forall d, d' \in D. d \sqsubseteq_D d' \implies f(d) \sqsubseteq_E f(d')$$

48) How to prove that a function is monotone?
Monotone functions *preserve the order* of the elements in a chain, to prove that a function is monotone we can show that if:

$$\{d_i\}_{i\in\mathbb{N}} \text{ is a chain in } D,\ f : D \to E \text{ monotone} \implies \{f(d_i)\}_{i\in\mathbb{N}} \text{ is a chain in } E\ .$$

49) What is a continuous function?
Given two partial orders $(P, \sqsubseteq_P), (E, \sqsubseteq_E)$ and a monotone function $f : D \to E$, we say that f is **continuous**, or *limit preserving*, if:

$$\forall \{d_i\}_{i\in\mathbb{N}}. f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} f(d_i)$$

50) How to prove that a function is continuous?
We have to show that the function is monotone and:

$$\forall \{d_i\}_{i\in\mathbb{N}}. f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} f(d_i)$$

51a) What is a fixpoint?
Given a PO $(D, \sqsubseteq)$, a monotone function $f : D \to D$ a **fixpoint** is an element $p \in D$ such that:

$$f(p) = p$$

51b) What is a pre-fixpoint?
Given a PO $(D, \sqsubseteq)$, a monotone function $f : D \to D$ a **pre-fixpoint** is an element $p \in D$ such that:

$$f(p) \sqsubseteq p$$

52) What is Kleene's fixpoint theorem?
Let $(D, \sqsubseteq_D)$ be a CPO with bottom, $f : D \to D$ a continuous function, we define:

$$fix(f) \triangleq \bigsqcup_{i \in \mathbb{N}} f^n(\bot)$$

then:

1. $fix(f)$ is a fixpoint of f:
$$f(fix(f)) = fix(f)$$
2. $fix(f)$ is the least pre-fixpoint of f:
$$\forall d \in D. f(d) \sqsubseteq d \implies fix(f) \sqsubseteq d$$
if d is a pre-fixpoint then $fix(f)$ is smaller than d.

53) What is the immediate consequence operator, also known as ICO?
Given a set of formulas F, a logical system R, a set of facts / hypothesis S, we define the **immediate consequence operator** $\hat{R}$ as:

$$\hat{R}(S) \triangleq \left\{ y \;\middle|\; \exists\ \frac{x_1, ..., x_n}{y} \in R.\ \{x_1, ..., x_n\} \subseteq S \right\}$$

- $\hat{R}$ applies the rules to the facts in all possible ways;
- $\hat{R}(S)$: all conclusions we can draw in one step from hypothesis on S;

For example, $\hat{R}(\emptyset)$ allows deriving all the conclusions with empty premises, that are the *axioms* of the logical system R.

54) When is the immediate consequence operator continuous?
The immediate consequence operator is continuous if and only if each rule in R has finitely many premises. Otherwise, we could not be able to prove the continuity of the function.

Proof. To show that the ICO function is continuos, we should prove:

$$\forall \{S_i\}_{i \in \mathbb{N}} \subseteq \mathcal{P}(F). \bigsqcup_{i \in \mathbb{N}} \hat{R}(S_i) = \hat{R}(\bigsqcup_{i \in \mathbb{N}} S_i)$$

To prove the equality between two sets, we should be able to prove the following property. Given two sets A and B, they are equal if they contain the same elements, that is:

$$A = B \iff A \subseteq B \wedge B \subseteq A$$

Therefore, the proof is dived in two parts:

1. $\bigcup_{i\in\mathbb{N}} \hat{R}(S_i) \subseteq \hat{R}\big(\bigcup_{i\in\mathbb{N}} S_i\big)$
2. $\bigcup_{i\in\mathbb{N}} \hat{R}(S_i) \supseteq \hat{R}\big(\bigcup_{i\in\mathbb{N}} S_i\big)$

The first point is trivial, we know that $\hat{R}$ and we are done with that. The second point requires some attentions. To show it, we need to see if each element $y \in \hat{R}\big(\bigsqcup_{i\in\mathbb{N}} S_i\big)$ then, we have $y \in \bigsqcup_{i\in\mathbb{N}} \hat{R}(S_i)$. To do that, let us take a generic element y (a conclusion) and we know that exists a rule such that $\exists \frac{x_1,\ldots,x_n}{y} \in R$ with $\{x_1,\ldots,x_n\} \subseteq \bigcup_{i\in\mathbb{N}} S_i$. Thus,

$$\forall j \in [1,\ldots,n].\exists k_j \in \mathbb{N}.x_j \in S_{k_j}$$

That is, for each premises with index j, there exists an index k_j, such that a premise x_j is contained inside the set S_{kj}. Now, fix k as the maximum of $k = \max\{k_1,\ldots,k_n\}$. The key point of the proof is here. If the set of premises was not finitely many, then we would not be able to fix k! Clearly,

$$\{x_1,\ldots,x_n\} \subseteq S_k$$

Thus,

$$y \in \hat{R}(S_k) \subseteq \bigcup_{i\in\mathbb{N}} \hat{R}(S_i)$$

55) How can the immediate consequence operator used to compute the theorems of a logical theory?
Given the theorem:

$$\forall n \in \mathbb{N}.\ P(n)$$

where:$\emptyset$

$$P(n) \triangleq I_R^n = \hat{R}^n(0)$$

we can compute the theorems of a logical theory using the following theorem (under the hypotheses that each rule must have finitely many premises)

$$fix(\hat{R}) = I_R$$

56) What is the syntax of IMP?

$$a \ ::= \ x \mid n \mid a_0 \ \mathbf{op} \ a_1$$
$$b \ ::= \ v \mid a \ \mathbf{cmp} \ a \mid \neg\, b \mid b \ \mathbf{bop} \ b$$
$$c \ ::= \ \mathbf{skip} \mid x \ := \ a \mid c\,;\, c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

where:

$$x \in Ide,\ n \in \mathbb{Z}, v \in \mathbb{B},\ \mathbf{op} \in \{+,\times,-\},\ \mathbf{bop} \in \{\wedge,\vee\},\ \mathbf{cmp} \in \{<,>,\leq,\geq,=,\neq\}.$$

57) What are the rules of the operational semantics of IMP?
The rules of the operational semantics of IMP can be categorized in three sections, one for each

syntactic element described in the grammar. Given an environment function $\sigma : \text{Ide} \to \mathbb{Z}$, we define the semantics of the arithmetic expressions as:

$$\frac{}{\langle n, \sigma\rangle \to n}\ (\textbf{num}) \quad \frac{}{\langle x, \sigma\rangle \to \sigma(x)}\ (\textbf{ide})$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \cdot_A a_1, \sigma\rangle \to n}[n = n_0 \cdot_A n_1]\ (\textbf{binop})$$

Where the $\cdot_A$ operator can be $+, -, \times$. For the boolean expressions, we have:

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \quad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \cdot_{Ba} a_1, \sigma\rangle \to n_0 \cdot_{Ba} n_1}\ (\textbf{equ, leq, geq})$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\textbf{not})$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to v_0 \wedge v_1}\ (\textbf{and}) \quad \frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to v_0 \vee v_1}\ (\textbf{or})$$

At the end, for the commands we have:

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ (\textbf{skip})$$

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma[m/x]}\ (\textbf{assign})$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}\ (\textbf{seq})$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\textbf{iftt}) \quad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\textbf{ifff})$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma''}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}\ (\textbf{whtt})$$

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}\ (\textbf{whff})$$

59) How is the denotational semantics of IMP defined?
The denotational semantics of arithmetic expressions is defined as follows. Let's define an interpretation function $\mathcal{A} : \text{Aexp} \to (\mathbb{M} \to \mathbb{Z})$:

$$\mathcal{A}[\![n]\!]\sigma \triangleq n$$

$$\mathcal{A}[\![x]\!]\sigma \triangleq \sigma(x)$$
$$\mathcal{A}[\![a_0 \ \mathbf{op} \ a_1]\!]\sigma \triangleq \mathcal{A}[\![a_0]\!]\sigma \ \ op \ \mathcal{A}[\![a_1]\!]\sigma$$

The denotational semantics of boolean expressions is defined as follows. Let's define an interpretation function $\mathcal{B} : \text{Bexp} \to (\mathbb{M} \to \mathbb{B})$:

$$\mathcal{B}[\![v]\!]\sigma \triangleq v$$
$$\mathcal{B}[\![a_0 \ \mathbf{cmp} \ a_1]\!]\sigma \triangleq \mathcal{A}[\![a_0]\!]\sigma \ \ cmp \ \mathcal{A}[\![a_1]\!]\sigma$$
$$\mathcal{B}[\![\neg b]\!]\sigma \triangleq \neg\mathcal{B}[\![b]\!]\sigma$$
$$\mathcal{B}[\![b_0 \ \mathbf{bop} \ b_1]\!]\sigma \triangleq \mathcal{B}[\![b_0]\!]\sigma \ \ bop \ \mathcal{B}[\![b_1]\!]\sigma$$

The denotational semantics of commands is defined as follows. Before defining the interpretation function, let us introduce the lifting function and the *cond* function:

Definition 4 (Lifting)
$$f^*(x) = \begin{cases} \bot & x = \bot \\ f(x) & otherwise \end{cases}$$

Definition 5 (Cond) *The* $cond : (\mathbb{B} \times \mathbb{M} \times \mathbb{M}) \to \mathbb{M}$ *function is defined as follows:*
$$cond(t, \sigma, \sigma') = \begin{cases} \sigma & t = True \\ \sigma' & otherwise \end{cases}$$

The commands will be denoted using memories, let us define the interpretation function $\mathcal{C}$: Com:

$$\mathcal{C}[\![\mathbf{skip}]\!]\sigma \triangleq \sigma$$
$$\mathcal{C}[\![x := a]\!]\sigma \triangleq \sigma[\mathcal{A}[\![a]\!]\sigma/x]$$
$$\mathcal{C}[\![c_0; c_1]\!]\sigma \triangleq \mathcal{C}[\![c_1]\!]^*(\mathcal{C}[\![c_0]\!]\sigma)$$
$$C[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] = \text{cond}(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!], \mathcal{C}[\![C_1]\!])$$

The denotational semantics of the "while" is the most tricky one. Let us define the continuous function $\Gamma_{b,c} : (\mathbb{M} \rightharpoonup \mathbb{M}) \to (\mathbb{M} \rightharpoonup \mathbb{M})$ as:

$$\Gamma_{b,c} \triangleq \lambda\phi.\lambda\sigma.\text{cond}(\mathcal{B}[\![b]\!], \phi^*(\mathcal{C}[\![c]\!]\sigma), \sigma)$$

Therefore, the denotational semantics of the "while" command is:

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \Gamma_{b,c}\ \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \text{fix}\ \Gamma_{b,c} = \bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}(\bot_{\mathbb{M}\to\mathbb{M}_\bot})$$

Using the Immediate Consequence Operator (ICO), we can compute the fixpoint of the command using $\hat{R}_{b,c}$:
$$\hat{R}_{b,c} = \left\{ \frac{}{(\sigma,\sigma)}\ \neg\mathcal{B}[\![b]\!]\sigma,\ \frac{(\sigma'',\sigma')}{(\sigma,\sigma')}\ \mathcal{B}[\![b]\!]\sigma \wedge \mathcal{C}[\![c]\!]\sigma = \sigma'' \right\}$$

60) How to prove termination of arithmetic expressions?
We can prove termination of arithmetic expressions with structural induction, we should prove:

$$\forall a.P(a)$$

where:

$$P(a) \triangleq \forall \sigma \in \mathbb{M}.\, \exists m \in \mathbb{Z}.\, \langle a, \sigma \rangle \to m$$

61) How to prove determinacy of arithmetic expressions?
We can prove determinacy of arithmetic expressions with structural induction, we should prove:

$$\forall a.P(a)$$

where:

$$P(a) \triangleq \forall \sigma \in \mathbb{M}.\, \forall m, m' \in \mathbb{Z}.\, \langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m' \implies m = m'$$

62) How to prove determinacy of commands?
We can prove determinacy of commands expressions with rule induction, we should prove:

$$\forall c, \sigma, \sigma_1.P(\langle c, \sigma \rangle \to \sigma_1)$$

where:

$$P(\langle c, \sigma \rangle \to \sigma_1) \triangleq \forall \sigma_2.\langle c, \sigma \rangle \to \sigma_2 \implies \sigma_1 = \sigma_2$$

64) Which rule to prove divergence of a command?
Consider $w \triangleq$ **while** b **do** c, suppose we find a set of memories $S \subseteq M$ such that:

$$\forall \sigma.\langle b, \sigma \rangle \to \mathbf{tt}$$

$$\forall \sigma.\, \forall \sigma'.\langle c, \sigma \rangle \to \sigma' \implies \sigma' \in S$$

then we can conclude $\forall \sigma.\langle w, \sigma \rangle \not\to$

6x) Correctness and Completeness of commands in IMP?
The central proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in very different formalism: on the one hand we have an inference rule system which allows us to calculate the execution of each command; on the other hand we have a function which associates with each command its functional meaning. Therefore, to show the equivalence between the two semantics, we have to prove the following theorem.

Theorem 2 (Equivalence of Commands)

$$\forall c \in \mathit{Com}.\forall \sigma, \sigma' \in \Sigma. \quad c, \sigma \to \sigma' \iff \mathcal{C}[\![c]\!]\sigma = \sigma'$$

The theorem should be proved into two parts:

- **Correctness**: $\forall c \in \mathrm{Com}.\forall \sigma, \sigma' \in \Sigma$ we prove using, rule induction, the following predicate:

$$P(\langle c, \sigma \rangle \to \sigma') \triangleq \mathcal{C}[\![c]\!]\sigma = \sigma'$$

- **Completeness**: $\forall c \in \mathrm{Com}$ we prove, using structural induction, the following predicate:

$$P(c) \triangleq \forall \sigma, \sigma' \in \Sigma.\mathcal{C}[\![c]\!]\sigma = \sigma' \implies \langle c, \sigma \rangle \to \sigma'$$

Notice that in this way the undefined cases are also handled for the equivalence. For instance, we have a corollary that:

$$\langle c, \sigma \rangle \not\to \iff \mathcal{C}[c]\sigma = \perp_{\Sigma_\perp}$$

64) What is the lambda-notation?
Lambda notation is a calculus invented by *Alonzo Church*. The λ-notation is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

65) What is alpha-conversion?
Alpha conversion (also written α-conversion) is a way of removing name clashes in expressions. A name clash arises when a β-reduction places an expression with a free variable in the scope of a bound variable with the same name as the free variable.

66) What is capture-avoiding substitution?
È una tecnica usata per evitare che nell'applicazione della *beta reduction* vengano "catturate" delle variabili libere. Cioè le variabili libere in e, devono restare libere anche dopo l'applicazione di [e / x] e questo lo si ottiene applicando un'*alpha conversion* prima di sostituire.
Esempio:

$$\begin{aligned}(\lambda y.x^2 + 2y + 5)[y/x] \equiv (\lambda z.(x^2 + 2y + 5)[z/y])[y/x] \equiv \\ \equiv (\lambda z.x^2 + 2z + 5)[y/x] \equiv \lambda z.((x^2 + 2z + 5)[y/x]) \equiv \\ \equiv \lambda z.y^2 + 2z + 5\end{aligned}$$

Notiamo che la variabile y è ancora *free*.

67) What is a pure functional programming language?
A pure functional programming language is a functional language without side effects, like *Haskell* for example. Instead, *OCaml* is not a pure functional programming language because it allows side effects using the *reference* type constructor.

68) What is lazy evaluation?
In programming language theory, lazy evaluation, or call-by-need, is **an evaluation strategy** which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).

The benefits of lazy evaluation include:

- The ability to define control flow (structures) as abstractions instead of primitives.
- The ability to define potentially infinite data structures. This allows for more straightforward implementation of some algorithms.
- The ability to define partially-defined data structures where some elements are errors. This allows for rapid prototyping.

69) What is list comprehension in Haskell?
È un modo di definire una nuova lista utilizzando gli elementi di una lista già esistente, attraverso una notazione simile alla "set comprehension".
Per esempio, supponiamo di avere una lista X, possiamo definirne una nuova

$$[x \mid x \leftarrow X,\ f\,x > 5]$$

70) How is iteration achieved in Haskell?
The iteration in Haskell is achieved through the meaning of recursion.

71) What is the syntax for guards in function definition in Haskell?

```
absolute x
 | x<0 = -x
 | otherwise = x
```

72) What is the syntax for pattern matching in function definition in Haskell?

```
take  0     _           =  []
take  _     []          =  []
take  n     (x:xs)      =  x : take (n-1) xs
```

73) What is partial application?
In computer science, partial application (or partial function application) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity.
For example the following function expects two arguments,

```
sum :: Num a ⇒ a → a → a
sum x y = x + y
```

If we provide only one argument a partial application is performed and the return value is another function that expects only one argument

```
sum5 = sum 5
Prelude> :t sum5
sum5 :: Num a ⇒ a → a
```

74) What is a functional type?
A **functional type** is the signature (the type) of a function, that is, given two types $\tau_0, \tau_1 \in \mathcal{T}$ and a function f, we say:

$$f : \tau_0 \to \tau_1 \in \mathcal{T}$$

is a *functional type*.

75) What is a higher order function in Haskell?
It's a function that as argument expects another function, for example the `map` function shown below.

```
map :: (a → b) → [a] → [b]
```

The `map` function takes two arguments: a function `f`, which it will be applied to each element of the list passed as the second argument.

76) What is the syntax for defining new data types in Haskell?

```
data ThisOrThat = This | That
```

77) What is the syntax for defining new type classes in Haskell?

```
class Typeclass a where
  fn1 :: a → [a] → [a]
```

```
fn2 :: a → a → a
fn3 :: Integer → a → Integer
```

As an analogy from an object-oriented paradigm, *typeclasses* are similar to interfaces in that any type that has an instance of a specific typeclass must implement every function defined by that typeclass.

(From "*Real World Haskell*" by Bryan O'Sullivan et al.). **Typeclasses** define a set of functions that can have different implementations depending on the type of data they are given. Typeclasses allow polymorphic functions in Haskell. By default, in Haskell are defined several typeclasses, such as: `Show`, `Num`, `Read` and so on and so forth.

78) What is the syntax of HOFL?

$$t ::= n \mid x \mid t_0 \ \mathbf{op} \ t_1 \mid \mathbf{if} \ t \ \mathbf{then} \ t_0 \ \mathbf{else} \ t_1$$
$$\mid (t_0, t_1) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t)$$
$$\mid \lambda x.t \mid t_0 \ t_1 \mid \mathbf{rec} \ x.t$$

79) What is the difference between pre-terms and terms in HOFL?
I *pre-terms* sono tutti i termini che si possono generare dalla grammatica di HOFL e comprendono anche i termini che non sono semanticamente corretti, ad esempio

$$\text{fst}(4)$$

$$4 + \lambda x.5$$

mentre i *terms* sono tutti i termini corretti e che sono garantiti essere validi da un sistema di tipi.

Let $\mathcal{T}$ be the set of all types, a pre-term t is *well-formed* if:

$$\exists \tau \in \mathcal{T}.\ t : \tau$$

80) What is a closed term of HOFL?
Terms with no free variables, that is, variables not bound by a binder (like the λ).

81) What are the types of HOFL?
Types in HOFL are generated by the following grammar:

$$\tau ::= \text{int} \mid \tau_0 * \tau_1 \mid \tau_0 \to \tau_1$$

82) What are the rules of the type system of HOFL?

$$\frac{}{n : int} \qquad \frac{}{x : \hat{x}} \qquad \frac{t_0 : int \quad t_1 : int}{t_0 \ \mathbf{op} \ t_1 : int} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1}$$

$$\frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1} \qquad \frac{x : \tau_0 \quad t : \tau}{\lambda x.t : \tau_0 \to \tau_1}$$

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_0 \to \tau_1}{t_1 \ t_0 : \tau_1} \qquad \frac{x : \tau \quad t : \tau}{\mathbf{rec} \ x.t : \tau}$$

83) What is a principal type?
È il tipo più generale di un termine che si riesce ad inferire con le type rules.

84) What is the style of the operational semantics of HOFL?
The style of the operational semantics of HOFL is *lazy big step* semantics.

85) What is a canonical form?
A set of canonical forms $C_\tau \subseteq T_\tau$ for a type τ, is defined as follows:

$$\frac{}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}}$$

86) What are the rules of the operational semantics of HOFL?

$$\frac{}{n \to n} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \to (t_0, t_1)} \qquad \frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \to \lambda x.t}$$

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{op} n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1} \qquad \frac{t[\textbf{rec } x.t/x] \to c}{\textbf{rec } x.t \to c}$$

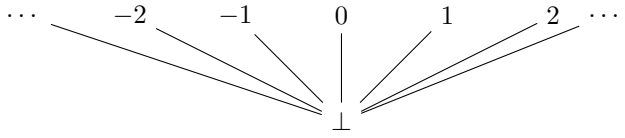
$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[t_0/x] \to c}{(t_1\ t_0) \to c}\ \text{(lazy)} \qquad \frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[c_0/x] \to c}{(t_1\ t_0) \to c}\ \text{(eager)}$$

87) What is the difference between lazy and eager evaluation?

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[t_0/x] \to c}{(t_1\ t_0) \to c}\ \text{(lazy)} \qquad \frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[c_0/x] \to c}{(t_1\ t_0) \to c}\ \text{(eager)}$$

As shown in the rules above, the main differences between the two approaches are that in the eager evaluation we evaluate the given function parameter as soon as possible, and then we substitute each occurrence inside the function's body with the evaluated value. Instead, in the lazy one, we delay the evaluation, and we just substitute the occurrence of the parameter inside the function's body.

88) What is the domain $\mathbb{Z}_\perp$?
The domain $\mathbb{Z}_\perp$ is the flat domain of integer numbers.



89) How is the cartesian product domain defined?
Given $\mathcal{D}(D, \sqsubseteq_D)$, $\mathcal{E}(E, \sqsubseteq_E)$ two $\text{CPO}_\perp$, let's define the cartesian product:

$$\mathcal{D} \times \mathcal{E} \triangleq (D \times E, \sqsubseteq_{D \times E})$$

we can order pairs as:

$$(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1) \iff d_0 \sqsubseteq_D d_1 \wedge e_0 \sqsubseteq_E e_1$$

90) How is the limit in a cartesian product domain defined?
Given the CPO:

$$\mathcal{D} \times \mathcal{E} \triangleq (D \times E, \sqsubseteq_{D\times E})$$

take a chain $\{(d_i, e_i)_{i\in\mathbb{N}}\}$, its limit is:

$$\left(\bigsqcup_{i\in\mathbb{N}} d_i, \bigsqcup_{i\in\mathbb{N}} e_i\right)$$

91) What is the Switch Lemma?
Let $(E, \sqsubseteq_E)$ be a CPO whose elements are of the form $e_{n,m}$ with $n, m \in \mathbb{N}$. If $\sqsubseteq$ is such that

$$e_{n,m} \sqsubseteq e_{n',m'} \text{ if } n \leq n' \wedge m \leq m'$$

then it holds:

$$\bigsqcup_{n,m\in\mathbb{N}} e_{n,m} = \bigsqcup_{n\in\mathbb{N}} \left(\bigsqcup_{m\in\mathbb{N}} e_{n,m}\right) = \bigsqcup_{m\in\mathbb{N}} \left(\bigsqcup_{n\in\mathbb{N}} e_{n,m}\right) = \bigsqcup_{k\in\mathbb{N}} e_{k,k}$$

92) How is the functional domain defined?
Let $\mathcal{D} = (D, \sqsubseteq_D)$, $\mathcal{E} = (E, \sqsubseteq_E)$ be two CPO, we define the set of all functions as

$$D \to E \triangleq \{f \mid f : D \to E\}$$

the set of all continuous functions as:

$$[D \to E] \triangleq \{f \mid f : D \to E,\ f \text{ is continuous}\}$$

we define the $\text{CPO}_\perp$:

$$[\mathcal{D} \to \mathcal{E}] \triangleq ([D \to E], \sqsubseteq_{[D\to E]})$$

where:

$$f \sqsubseteq_{[D\to E]} g \iff \forall d \in D.\, f(d) \sqsubseteq_E g(d)$$

93) How is the limit in a functional domain defined?
Given a chain $\{f_n : D \to E\}_{n\in\mathbb{N}}$, we define its limit as:

$$h \triangleq \lambda d.\bigsqcup_{n\in\mathbb{N}} f_n(d)$$

or similarly:

$$h(n) \triangleq \bigsqcup_{n\in\mathbb{N}} f_n(d)$$

94) How is the lifted domain defined?
Let $(D, \subseteq_D)$ be a CPO. We can build a lifted domain $\mathcal{D} = (D_\perp, \subseteq_{D_\perp})$ in the following way:

$$D_\perp = \{\perp\} \uplus D = \{(0, \perp)\} \cup (\{1\} \times D) = \{(0, \perp)\} \cup \{(1, d) \mid d \in D\}$$

The bottom element of the lifted domain is defined as $\bot_{D_\bot} = (0, \bot)$. A *lifting function* is provided with a lifted domain, it allows to map an element from a domain D to a lifted one:

$$\begin{aligned} \lfloor \cdot \rfloor &: D \to D_\bot \\ \lfloor d \rfloor &= (1, d) \end{aligned}$$

The elements are ordered according to the following criteria:

1. $\forall x \in D_\bot . \bot_{D_\bot} \sqsubseteq_{D_\bot} x$
2. $\forall d_1, d_2 \in D . \lfloor d_1 \rfloor \sqsubseteq_{D_\bot} \lfloor d_2 \rfloor \iff d_1 \sqsubseteq_D d_2$

95) How is the limit in a lifted domain defined?
(From "*The Formal Semantics of Programming Languages*" by Glynn Winskel). [...] Lifting adjoins a bottom element below a copy of the original CPO. We can use the definition of the lifting function to derive the limit of a lifted domain in the following way:

$$\bigsqcup_{i \in \mathbb{N}} \lfloor d_i \rfloor = \bigsqcup_{i \in \mathbb{N}} (1, d_i)$$

96) How to prove continuity of a function $f : D \to E_1 \times E_2$?
Theorem. Given $(D, \sqsubseteq_D)$, $(E_i, \sqsubseteq_{E_i})$, $f : D \to E_1 \times E_2$, $g \triangleq \pi_i \circ f$, then

$$f \text{ is continuous} \iff g_1, g_2 \text{ are continuous}$$

Proof.

$$(\implies)$$

f is continuous, π_i is continuous, then g_i is continuous.

$$(\impliedby)$$

Note that $f(d) = (g_1(d), g_2(d))$, assume g_1, g_2 are continuous, we show that f is also continuous. Take a chain $\{d_i\}_{i \in \mathbb{N}}$, we need to prove that

$$f\left(\bigsqcup_{i \in \mathbb{N}} d_i\right) = \bigsqcup_{i \in \mathbb{N}} f(d_i)$$

so,

$$\begin{aligned} f\left(\bigsqcup_{i \in \mathbb{N}} d_i\right) &= \left(g_1\left(\bigsqcup_{i \in \mathbb{N}} d_i\right), g_2\left(\bigsqcup_{i \in \mathbb{N}} d_i\right)\right) \\ &= \left(\bigsqcup_{i \in \mathbb{N}} g_1(d_i), \bigsqcup_{i \in \mathbb{N}} g_2(d_i)\right) \; g_1, g_2 \text{ are continuous} \\ &= \bigsqcup_{i \in \mathbb{N}} (g_1(d_i), g_2(d_i)) \text{ by def of lub of pairs} \\ &= \bigsqcup_{i \in \mathbb{N}} f(d_i) \end{aligned}$$

97) How to prove continuity of a function $f : D \times E \to F$?
Theorem. Given $(D, \sqsubseteq_D)$, $(E_i, \sqsubseteq_{E_i})$, $(F_i, \sqsubseteq_{F_i})$, $f : D \times E \to F$:

$$f_d : E \to F,\ f_d \triangleq \lambda e.f(d, e)$$

$$f_e : D \to F,\ f_e \triangleq \lambda d.f(d, e)$$

then:

$$f \text{ is continuous} \iff \forall d \in D.f_d \text{ are continuous} \wedge \forall e \in E.f_e \text{ are continuous}$$

Proof.

$$(\implies)$$

(The proof for f_e is analogous and omitted).
Assume f is continuous, take a generic $d \in D$, we want to prove f_d is continuous. Take a chain $\{e_i\}_{i\in\mathbb{N}}$ in E, we need to prove

$$f_d\Big(\bigsqcup_{i\in\mathbb{N}} e_i\Big) = \bigsqcup_{i\in\mathbb{N}} f_d(e_i)$$

so,

$$= f_d\Big(\bigsqcup_{i\in\mathbb{N}} e_i\Big) = f\Big(d, \bigsqcup_{i\in\mathbb{N}} e_i\Big)$$

$$= f\Big(\bigsqcup_{i\in\mathbb{N}} d, \bigsqcup_{i\in\mathbb{N}} e_i\Big) \text{ by lub of constant chain}$$

$$= f\Big(\bigsqcup_{i\in\mathbb{N}} (d, e_i)\Big) \text{ by lub of pairs}$$

$$= \bigsqcup_{i\in\mathbb{N}} f(d, e_i) \text{ by continuity of } f$$

$$= \bigsqcup_{i\in\mathbb{N}} f_d(e_i)$$

$$(\impliedby)$$

assume f_d, f_e are continuous for all d, e, we want to prove f is continuous, that is, taken a chain $\{(d_k, e_k)_{k\in\mathbb{N}}\}$ in $D \times E$, we prove

$$f\Big(\bigsqcup_{k\in\mathbb{N}} (d_k, e_k)\Big) = \bigsqcup_{k\in\mathbb{K}} f(d_k, e_k)$$

so,

$$f\Big(\bigsqcup_{k\in\mathbb{N}} (d_k, e_k)\Big) = f\Big(\bigsqcup_{i\in\mathbb{N}} d_i, \bigsqcup_{j\in\mathbb{N}} e_j\Big) \text{ by def of lub of pairs}$$

$$= f_d\Big(\bigsqcup_{j\in\mathbb{N}} e_j\Big) \text{ by def of } f_d \text{ with } d \triangleq \bigsqcup_{i\in\mathbb{N}} d_i$$

$$= \bigsqcup_{j \in \mathbb{N}} f_d(e_j) \text{ by continuity of } f_d$$

$$= \bigsqcup_{j \in \mathbb{N}} f_{e_j}(d) \text{ by def of } f_{e_j}$$

$$= \bigsqcup_{j \in \mathbb{N}} f_{e_j}\left(\bigsqcup_{i \in \mathbb{N}} d_i \right) \text{ by def of } d \triangleq \bigsqcup_{i \in \mathbb{N}} d_i$$

$$= \bigsqcup_{j \in \mathbb{N}} \bigsqcup_{i \in \mathbb{N}} f_{e_j}(d_i) \text{ by continuity of } f_{e_j}$$

$$= \bigsqcup_{j \in \mathbb{N}} \bigsqcup_{i \in \mathbb{N}} f(d_i, e_j) \text{ by def of } f_{e_j}$$

$$= \bigsqcup_{k \in \mathbb{N}} f(d_k, e_k) \text{ by switch lemma (applicable?)}$$

in order to ensure that the switch lemma is applicable, we need to guarantee that

$$\text{if } i \leq n \wedge j \leq m \text{ then } f(d_i, e_j) \sqsubseteq f(d_n, e_m)$$

we know that $\{d_i\}_{i \in \mathbb{N}}, \{e_j\}_{j \in \mathbb{N}}$ are chains so,

$$d_i \sqsubseteq_D d_n \wedge e_j \sqsubseteq_E e_m$$

and since f_d, f_e are continuous and so monotone, we conclude with

$$f(d_i, e_j) = f_{d_i}(e_j) \sqsubseteq f_{d_i}(e_m) = f(d_i, e_m) = f_{e_m}(d_i) \sqsubseteq f_{e_m}(d_n) = f(d_n, e_m)$$

98) How is the apply function defined?
The `apply` function is monotone and continuous, and it is defined over two CPO $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$:

$$\begin{aligned} \text{apply} : [D \to E] \times D &\to E \\ \text{apply}(f, d) &= f(d) \end{aligned}$$

It applies the function $f \in [D \to E]$ to the argument $d \in D$.

99) How is the fix function defined?
The `fix` function is a monotone and continous function over the CPO $(D, \sqsubseteq_D)$:

$$\begin{aligned} \text{fix} &: [D \to D] \to D \\ \text{fix} &= \lambda f. \bigsqcup_{n \in \mathbb{N}} f^n(\bot_D) \end{aligned}$$

The function computes the least-fixed point of a function $f \in [D \to D]$ according to the Kleene's theorem.

100) What is the let notation?

The *let* notation is used as a syntactic sugar to '*de-lift*' a value. Given a CPO with bottom $(E, \sqsubseteq_E)$ and a lambda abstraction $\lambda x.e \in [D \to E]$ and an element $t \in D_\perp$:

$$\mathbf{let}\, x \iff t.e \triangleq (\lambda x.e)^* t = \begin{cases} \perp_E & t = \perp_{D_\perp} \\ e[d/x] & t = \lfloor d \rfloor \end{cases}$$

Intuitively: if t is a lifted valued $\lfloor d \rfloor$ then we de-lift the value and assign it to x in e, otherwise returns $\perp_E$.

101) What are the domains needed to define the denotational semantics of HOFL?
The domains of the denotational semantics of HOFL are defined by structural recursion:

$$\begin{aligned} V_{\text{int}} &= \mathbb{Z} \quad (V_{\text{int}}) = \mathbb{Z}_\perp \\ V_{\tau_1 * \tau_2} &= (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp \quad (V_{\tau_1 * \tau_2})_\perp = ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ V_{\tau_1 \to \tau_2} &= [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp] \quad (V_{\tau_1 \to_2})_\perp = [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp \end{aligned}$$

102) How is the denotational semantics of HOFL defined?
To specify the denotational semantics of a programming language, we have to defined, by structural recursion, an interpretation function from each syntactic domain to a semantic domain. HOFL has a sole syntactic domain (i.e., the set of well-formed terms t) and thus we have only one interpretation function, written $[\![\cdot]\!]$. However, since HOFL terms are typed, the interpretation function is parametric w.r.t the type τ of t and we have one semantic domain V_τ for each type τ. We distinguish between V_τ, where we find the meaning of the terms of type τ with canonical forms, and $(V_\tau)_\perp$, where the additional element $\perp_{(V_\tau)_\perp}$ assigns a meaning to all the terms of type τ *without* canonical forms.

Moreover, we will need an *environment* function ρ which assigns meaning to the free variables.

$$\rho \in \text{Env} = \text{Var} \to \bigcup_\tau (V_\tau)_\perp$$

Now, we are going to define the interpretation function of HOFL by structural recursion.

- **Constants**: $[\![n]\!]_\rho = \lfloor n \rfloor$
- **Variables**: $[\![x]\!]_\rho = \rho(x)$
- **Arithmetic operators**: $[\![t_0 \,\text{op}\, t_1]\!]_\rho = [\![t_0]\!]_\rho \,\text{op}_\perp\, [\![t_1]\!]_\rho$
- **Conditional**:
$$\text{Cond}_\tau(v, d_0, d_1) = \begin{cases} d_0 & v = \lfloor 0 \rfloor \\ d_1 & \exists n \in \mathbb{Z}. v = \lfloor n \rfloor \wedge n \neq 0 \\ \perp_{(V_\tau)_\perp} & v = \perp_{\mathbb{Z}_\perp} \end{cases}$$
- **Pairing**: $[\![(t_0, t_1)]\!]_\rho = \lfloor ([\![t_0]\!]_\rho, [\![t_1]\!]_\rho) \rfloor$
- **Projections**: $[\![\mathbf{fst}(t)]\!]_\rho = \pi_1^*([\![t]\!]_\rho)$ and $[\![\mathbf{snd}(t)]\!]_\rho = \pi_2^*([\![t]\!]_\rho)$
- **Lambda Abstraction**: $[\![\lambda x.t]\!]_\rho = \lfloor \lambda d.[\![t]\!]_{\rho[d/x]} \rfloor$

- **Function Application**: $[\![(t_1 t_0)]\!]_\rho = \mathbf{let}\phi \leftarrow [\![t_1]\!]_\rho.\phi([\![t_0]\!]_\rho) = (\lambda\phi.\phi([\![t_0]\!]_\rho))^*([\![t_1]\!]_\rho)$
- **Recursion**: $[\![\mathbf{rec}\, x.t]\!]_\rho = \mathrm{fix}\, \lambda d.[\![t]\!]_{\rho[d/x]}$

103) What is the Substitution Lemma?
Given $x, t_0 : \tau_0$, $t : \tau$, the *substitution lemma* states:

$$[\![t[t_0/x]]\!]\rho = [\![t]\!]\rho[[\![t_0]\!]\rho/x]$$

In other words, the Substitution Lemma states that the substitution operator commutes with the interpretation function. The lemma is an important result, as it implies the compositionality of denotational semantics, namely for all terms t_1, t_2 and environment ρ, we have:

$$[\![t_1]\!]\rho = [\![t_2]\!]\rho \implies [\![t[t_1/x]]\!]\rho = [\![t[t_2/x]]\!]\rho$$

104) What are the possible denotations of canonical HOFL terms?
We have the following denotations:

$$\frac{}{n \in C_{int}} \qquad [\![n]\!]\rho = \lfloor n \rfloor$$

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad [\![(t_0, t_1)]\!]\rho = \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor$$

$$\frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}} \qquad [\![\lambda x.t]\!]\rho = \lfloor \lambda d.[\![t]\!]\rho[d/x] \rfloor$$

105) Does the denotational semantics of closed HOFL terms depend on the environment?
The answer is no, indeed we have the following theorem.
Theorem. Given a term $t : \tau$,

$$\forall x \in fv(t).\ \rho(x) = \rho'(x) \implies [\![t]\!]\rho = [\![t]\!]\rho'$$

In particular, the answer is given from the following corollary

$$t \text{ closed} \implies \forall \rho, \rho'.\ [\![t]\!]\rho = [\![t]\!]\rho'$$

106) What is the relation between the operational and denotational semantics of HOFL?
The two semantics are not consistent, indeed:

$$\forall t, c.\, t \to c \iff \forall \rho.[\![t]\!]\rho = [\![c]\!]\rho$$

in particular, we have:

1. the **correctness** property holds (the proof proceeds using rule induction):

$$t \to c \implies \forall \rho.[\![t]\!]\rho = \rho[\![c]\!]\rho$$

2. the **completeness** property does **not** hold:

$$\forall \rho.[\![t]\!]\rho = \rho[\![c]\!]\rho \not\implies t \to c$$

To show that, let us make an example. Let $c_0 = \lambda x.x + 0$ and $c_1 = \lambda x.x$ be two HOFL terms, where x : int. Clearly:

$$[\![c_0]\!]\rho = [\![c_1]\!]\rho \quad \text{but} \quad c_0 \not\to c_1$$

The two terms in the operational semantics are already in canonical forms (see the rules above). Instead, in the denotational semantics, we have to apply the interpretation function to discover that:

$$[\![c_0]\!]\rho = \lfloor \lambda d.d +_\perp 0 \rfloor = \lfloor \lambda d.x \rfloor = [\![c_1]\!]\rho$$

107) For which type the operational and denotational semantics are consistent?
Only for integer numbers.

108) Do operational and denotational convergence coincide in HOFL?
We have consistency on convergence.

Theorem 3 *$t : \tau$ closed, $t\downarrow \implies t\Downarrow$*

Proof.

$$t\downarrow \implies t \to c \implies (\forall\rho.[\![t]\!]\rho \implies [\![c]\!]\rho) \implies \forall\rho.[\![t]\!]\rho \neq \perp \implies t\Downarrow$$

109) What is the unlifted denotational semantics of HOFL?
Consider the following *unlifted* domains:

- $U_{int} \triangleq \mathbb{Z}_\perp$
- $U_{\tau_1 * \tau_2} \triangleq U_{\tau_1} \times U_{\tau_2}$
- $U_{\tau_1 \to \tau_2} \triangleq [U_{\tau_1} \to U_{\tau_2}]$

we have the following unlifted denotational semantics:

$$(\!|n|\!)\rho \triangleq \lfloor n \rfloor$$

$$(\!|x|\!)\rho \triangleq \rho(x)$$

$$(\!|t_1 \ \mathbf{op} \ t_2|\!)\rho \triangleq (\!|t_1|\!)\rho \ \underline{op}_\perp \ (\!|t_2|\!)\rho$$

$$(\!|\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2|\!)\rho \triangleq \mathrm{Cond}((\!|t|\!)\rho, (\!|t_1|\!)\rho, (\!|t_2|\!)\rho)$$

$$(\!|\mathbf{rec}\ x.\, t|\!)\rho \triangleq fix\, \lambda d.(\!|t|\!)\rho[d/x]$$

$$(\!|(t_1, t_2)|\!)\rho \triangleq ((\!|t_1|\!)\rho, (\!|t_2|\!)\rho)$$

$$(\!|\mathbf{fst}(t)|\!)\rho \triangleq \pi_1(\,(\!|(t)|\!)\rho\,)$$

$$(\!|\mathbf{snd}(t)|\!)\rho \triangleq \pi_2(\,(\!|(t)|\!)\rho\,)$$

$$(\!|\lambda x.t|\!)\rho \triangleq \lambda d.(\!|t|\!)\rho[d/x]$$

$$(\!|(t_1\ t_0)|\!)\rho \triangleq (\!|t_1|\!)\rho\ (\!|t_0|\!)\rho$$

110) How are new processes created in Erlang?
New processes in Erlang are created using the `spawn` primitive. The processes of Erlang are lightweight and they are handled by the **Erl** virtual machine. The VM can handle many processes and the switch between them is fast. On multiprocessor/multicore machines, processes can be scheduled to run in parallel.

111) What kind of communication is possible in Erlang?
In Erlang, processes can communicate using a message-passing system communication. A process can send a message to another one or receive it. The message passing is an asynchronous system and any value can be sent as a message. All processes have their own *mailbox* with no limits, messages are kept until extracted.

112) What is a process identifier?
A process identifier is a unique ID that identify a process. The concept is similar to the OS processes.

113) What is the syntax for sending messages in Erlang?
In Erlang, to send a message, we use the *bang* operator as shown below:

```
Pid ! message
```

114) What is the syntax for receiving a message in Erlang?
The syntax to receive a message is the following one:

```
receive
    Pattern1 → Expressions1;
    Pattern2 → Expressions2;
    ...
end
```

115) What is the meaning of the clause after?
(From the book: "Programming in Erlang" by *Joe Armstrong*).
Sometimes a `receive` statement might wait forever for a message that never comes. [...] To avoid this problem, we can add a timeout to the `receive` statement. This sets a maximum time that the process will wait to receive the message.

```
receive
    Pattern1 → Expressions1;
    Pattern2 → Expressions2;
    ...
after Time →
    Expressions
end
```

If no matching message has arrived within Time milliseconds of entering the `receive` expression, then the process will stop waiting for a message and evaluate `Expressions`.

Thanks to the `after` clause we could implement a *sleep* function, it is enough to provide an empty receive statement with only the `after Time` clause. If we pass 0 as receiving time, the VM will try to perform all the possible pattern matching.

116) How can pattern matching be used in receiving messages?
In addition to the standard pattern matching, with the `receive` statement we are able to perform the so-called *selective receiving*. From the book "*Programming in Erlang*" by *Joe Armstrong*: "(...), it (the `receive` statement) also queues unmatched messages for later processing and manage timeouts."

```
receive
```

```
    Pattern1 [when Guard1] → Expressions1;
    Pattern2 [when Guard2] → Expressions2;
    ...
after
    Time →
        ExpressionsTimeout
end
```

The above block works as follows:

1. When we enter `receive` statement, we start a timer (but only if an `after` section is present);
2. Take the first message from the mailbox and try to match it against `Pattern1`, `Pattern2` and so on. If the match succeeds, the messages are removed from the mailbox, and the expressions and the following pattern are evaluated;
3. If none of the patterns in the `receive` statement matches the first message in the mailbox, then the first message is removed from the mailbox and put into a "*save queue*". The second message in the mailbox is then tried. This procedure is repeated until a matching message is found or until all the messages in the mailbox have been examined.
4. If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled for execution the next time a new message is put in the mailbox. When a new message arrives, the messages in the save queue are not rematched; only the new message is matched.
5. As soon as a message has been matched, then all messages that have been put into the save queue are re-entered into the mailbox in the order in which they arrived at the process. If a timer was set, it is cleared.
6. If the timer elapses when we are waiting for a message, then evaluate the expressions `ExpressionsTimeout` and put any saved messages back into the mailbox in the order in which they arrived at the process.

117) What is the syntax of CCS?
The syntax of **Calculus of Communicating Systems** is shown below:

$$
\begin{aligned}
p, q ::= & \\
& \mathbf{nil} \text{ inactive process} \\
& \mid x \text{ process variable} \\
& \mid \mu.p \text{ action prefix} \\
& \mid p \setminus \alpha \text{ restricted channel} \\
& \mid p[\phi] \text{ channel relabelling} \\
& \mid p + q \text{ nondeterministic choice} \\
& \mid p|q \text{ parallel composition} \\
& \mid \mathbf{rec}\, x.p \text{ recursion}
\end{aligned}
$$

118) How is iteration achieved in CCS?
The recursion in CCS is achieved using the recursion operator and process names.

119) What is the style of the operational semantics of CCS?
(*From "Models of Computation" by Bruni and Montanari*). The operational semantics of CCS is defined by a suitable labelled transition system.

Definition 6 (Labelled Transition System) *A **Labelled Transition System** is a triple* $(P, L, \rightarrow)$*, where* P *is the set of states of the system,* L *is the set of labels and* $\rightarrow \subseteq P \times L \times P$ *is the transition relation.*

120) What are the labels of the operational semantics of CCS?
There are two kinds of labels in CCS:

- an *action*, that can be input (μ) or output ($\overline{\mu}$);
- an *internal action*, also called silent action indicated with τ (no interaction with environment).

121) What are the rules of the operational semantics of CCS?
The Operational Semantics of CCS is defined using Small Step Semantics:

$$\frac{}{\mu.p \xrightarrow{\mu} p}(\textbf{Act}) \quad \frac{p \xrightarrow{\mu} q}{p \setminus \alpha \xrightarrow{\mu} q \setminus \alpha}(\textbf{Res}) \quad \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}(\textbf{Rel})$$

$$\frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'}(\textbf{SumL}) \quad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}(\textbf{SumR}) \quad \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q}(\textbf{ParL}) \quad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}(\textbf{ParR})$$

$$\frac{p_1 \xrightarrow{\lambda} p_2 \; q_1 \xrightarrow{\bar{\lambda}} q_2}{p_1|q_1 \xrightarrow{\mu} p_2|q_2}(\textbf{Com}) \quad \frac{p[\textbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\textbf{rec}x.p \xrightarrow{\mu} q}(\textbf{Rec})$$

122) What is a finitely branching process?
Finitely branching processes are processes that don't contain infinitely many outgoing transitions.

123) What is a guarded process?
Guarded processes guarantee that process variables occur under a prefix (recursion is guarded by some action) under an action prefix.
Let X be a set of process variables, let p be a process, all recursively defined names are guarded in p if a name in X occurs free in p it is prefixed by an action.
We define $G(p, X)$ as:

- $G(\textbf{nil}, X) \triangleq \textbf{true}$
- $G(x, X) \triangleq x \notin X$
- $G(\mu.p, X) \triangleq G(p, \emptyset)$
- $G(p \backslash \alpha) \triangleq G(p, \alpha)$
- $G(p[\phi], X) \triangleq G(p, X)$
- $G(p + q, X) \triangleq G(p, X) \wedge G(q, X)$
- $G(p|q, X) \triangleq G(p, X) \wedge G(q, X)$

- $G(\textbf{rec}\ x.p, X) \triangleq G(p, X \cup \{x\})$

A closed process p is guarded if $G(p, \emptyset)$ holds true.

124) How can value-passing be encoded in CCS?
We can encode value-passing in CCS adding the following primitives:

$$\frac{}{\alpha!v.p \xrightarrow{\bar{\alpha_v}} p} \qquad \frac{}{\alpha?v.p \xrightarrow{\alpha_v} p[v/x]}$$

If the set of all possible values is finite $V \triangleq \{v_1, \ldots, v_n\}$, we can assume to have a dedicated channel α_{v_i} for each possible value v_i, thus

$$\alpha!v.p \equiv \bar{\alpha_v}.p$$

$$\alpha?v.p \equiv \alpha_{v_1}.p[v_1/x] + \ldots + \alpha_{v_n}.p[v_n/x]$$

125) Why graph isomorphism is not a good candidate as an equivalence for CCS?
Given two graphs $G(V, E)$, $G'(V', E')$ a bijective function $f : V \to V'$, G and G' are **isomorphic** if

$$\forall v, w, \mu.v \xrightarrow{\mu} w \iff f(v) \xrightarrow{\mu} f(w)$$

Graph isomorphism is not good because is too strict. There are some processes considered intuitively equivalent, but they are not isomorphic.

126) What is trace equivalence?
Trace Equivalence is the first attempt to define an equivalence between two agents P and Q. LTSs are essentially automata, and the classic theory of automata suggests a read made notion of equivalence for them, and thus for the CCS processes that denote them. The *trace* of a process P is a sequence having the following structure:

$$P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} P_k$$

We write Traces(P) for the collection of all traces of P. For all processes P and Q, we require that: if P and Q are behaviourally equivalent, then Traces(P) = Traces(Q). Basically, *two processes are equivalent if and only if they afford the same traces.*

127) Why trace equivalence is not a good candidate as an equivalence for CCS?
Because it identifies too many processes, it can be the case that two processes behaves differently, but they have the same trace. For instance, with trace equivalence, we cannot notice deadlocks.

128) What are the rules of the bisimulation game?
Given two processes $p, q \in P$ and two opposing players, Alice and Bob, we have:

- **Alice**, the attacker, aims to prove p and q are not equivalent;
- **Bob**, the defender, aims to prove p and q are equivalent;

The game is turn based, at each turn:

1. Alice chooses one process and one of its outgoing transitions;
2. Bob must reply with a transition of the other process, matching the label of the transition chosen by Alice;

3. At the next turn, if any, the players will consider the equivalence of the target processes of the chosen transitions.

129) What is a strong bisimulation?
Given a set of processes $\mathcal{P}$, a binary relation $\mathbf{R} \subseteq \mathcal{P} \times \mathcal{P}$, we write $p\ \mathbf{R}\ q$ when $(p,q) \in R$.
$\mathbf{R}$ is a strong bisimulation if

$$\forall p,q.(p,q) \in R \implies \begin{cases} \forall \mu, p'.p \xrightarrow{\mu} p' \implies \exists q'.q \xrightarrow{\mu} q' \wedge p'\,\mathbf{R}\,q' \\ \wedge \\ \forall \mu, q'.q \xrightarrow{\mu} q' \implies \exists p'.p \xrightarrow{\mu} p' \wedge p'\,\mathbf{R}\,q' \end{cases}$$

130) What is the difference between bisimulation and bisimilarity?
We define strong bisimilarity as follows:

$$\simeq \triangleq \bigcup_{\mathbf{R} \text{ is a s.b.}} \mathbf{R}$$

A strong bisimulation is not necessarily an equivalence. A strong bisimilarity an equivalence relation.

131) How to prove that two process are strong bisimilar?

$$p \simeq q \text{ iff } \exists \text{ a strong bisimulation } \mathbf{R} \text{ with } (p,q) \in \mathbf{R}$$

132) Under which operations are strong bisimulations closed?
Strong bisimulation is closed under:

- **Union**.
  Lemma If $\mathbf{R}_1$, $\mathbf{R}_2$ are strong bisimulation, then $\mathbf{R}_1 \cup \mathbf{R}_2$ is a strong bisimulation.

- **Inverse**.
  Lemma If $\mathbf{R}$ is a strong bisimulation, then:

  $$R^{-1} \triangleq \{(q,p) \mid p\ \mathbf{R}\ q\}$$

  is a strong bisimulation.

- **Composition**.
  If $\mathbf{R}_1$, $\mathbf{R}_2$ are strong bisimulations, then:

  $$\mathbf{R}_2 \circ \mathbf{R}_1 \triangleq \{(p,q) \mid \exists r.\ p\,\mathbf{R}_1\,r \wedge r\,\mathbf{R}_2\,q\}$$

  is a strong bisimulation.

133) How to prove that strong bisimilarity is an equivalence relation?
An equivalence relation is a relation that satisfy the properties of **reflexivity**, **symmetry** and **transitivity**. Therefore, we have to prove that strong bisimilarity is *reflexive*, *symmetric* and *transitive*:

- *reflexive*: $Id \subseteq \simeq$

- *symmetric*: assume $p \simeq q$, we want to prove $q \simeq p$
  $p \simeq q \implies \exists\, \mathbf{R}.\, p\, \mathbf{R}\, q$ with $(p,q) \in \mathbf{R}$
  then $(q,p) \in \mathbf{R}^{-1}$ and $\mathbf{R}^{-1}$ as a s.b. thus $(q,p) \in \mathbf{R}^{-1} \subseteq \simeq$ i.e. $q \simeq p$

134) How to prove that strong bisimilarity is a strong bisimulation?
Theorem. Strong bisimilarity is a strong bisimulation.
Proof.
Take $p \simeq q$, take $p \xrightarrow{\mu} p'$, we want to prove $\exists q'.\, q \xrightarrow{\mu} q'$, with $p' \simeq q'$.
Saying $p \simeq q$ means that exists a strong bisimulation $\mathbf{R}$ with $(p,q) \in \mathbf{R}$.
Since $\mathbf{R}$ is a strong bisimulation and $p \xrightarrow{\mu} p'$ we have $q \xrightarrow{\mu} q'$ with $(p',q') \in \mathbf{R}$.
Since $\mathbf{R} \subseteq \simeq$ we have $p' \simeq q'$.

Take $q \xrightarrow{\mu} q'$ we want to find $p \xrightarrow{\mu} p'$ with $p' \simeq q'$.
Saying $p \simeq q$ means that exists a strong bisimulation $\mathbf{R}$ with $(p,q) \in \mathbf{R}$.
Since $\mathbf{R}$ is a strong bisimulation and $q \xrightarrow{\mu} q'$ we have $p \xrightarrow{\mu} p'$ with $(p',q') \in \mathbf{R}$.
Since $\mathbf{R} \subseteq \simeq$ we have $p' \simeq q'$.

135) How to prove that strong bisimilarity is a congruence?
Theorem. Strong bisimilarity is a congruence, i.e.

1. $p \simeq q \implies \mu.p \simeq \mu.q$
2. $p \simeq q \implies p \setminus \alpha \simeq q \setminus \alpha$
3. $p \simeq q \implies p[\phi] \simeq q[\phi]$
4. $p_0 \simeq q_0 \wedge p_1 \simeq q_1 \implies p_0 + p_1 \simeq q_0 + q_1$
5. $p_0 \simeq q_0 \wedge p_1 \simeq q_1 \implies p_0 | p_1 \simeq q_0 | q_1$

136) How can strong bisimilarity be expressed as a least fixpoint?
Given a set of processes $\mathcal{P}$, let $\mathcal{P}_f \subseteq \mathcal{P}$ be the set of finitely many branching processes of $\mathcal{P}$, given the operator $\Phi : (\mathcal{P} \times \mathcal{P}) \to (\mathcal{P} \times \mathcal{P})$ as

$$\Phi(\mathbf{R}) \triangleq \{(p,q) \mid (\forall \mu, p'.p \xrightarrow{\mu} p' \implies \exists q'.q \xrightarrow{\mu} q' \wedge p'\, \mathbf{R}\, q') \wedge (\forall \mu, q'.q \xrightarrow{\mu} q' \implies \exists p'.p \xrightarrow{\mu} p' \wedge p'\, \mathbf{R}\, q')\}$$

We can express bisimilarity as a least fixpoint in the following way:

$$\simeq \triangleq \bigsqcup_{n \in \mathbb{N}} \Phi^n(\mathcal{P}_f \times \mathcal{P}_f)$$

(*where* $\bigsqcup = \bigcap$).

137) What is the Knaster-Tarski Theorem?

Definition 7 (Complete Lattice) *A complete lattice is a PO* $(D, \sqsubseteq)$ *such that*

- *any* $X \subseteq D$ *has a least upper bound* $\bigsqcap X$
- *any* $X \subseteq D$ *has a greatest lower bound* $\bigsqcup X$
- *it has a bottom element* $\bot = \bigsqcap D$

- *it has a top element* $\top = \bigsqcup D$

Theorem 4 (Knaster-Tarski) *Let* $(D, \sqsubseteq)$ *be a complete lattice,* $f : D \to D$ *a monotone function, then* f *has*

- *a least fixpoint* $d_{min} \triangleq \cap\{d \in D \mid f(d) \sqsubseteq d\}$.
- *a greatest fixpoint* $d_{max} \triangleq \cup\{d \in D \mid d \sqsubseteq f(d)\}$.

138) What are some equivalence laws satisfied by strong bisimilarity?
Thanks to strong bisimilarity, we have the following equivalence laws shown in Table 1.

$p + 0 \simeq p$	$p\|0 \simeq p$
$p + q \simeq q + p$	$p\|q \simeq q\|p$
$p + (q + r) \simeq (p + q) + r$	$p(q\|r) \simeq (p\|q)\|r$
$p + p \simeq p$	
$0 \setminus \alpha \simeq 0$	$0[\phi] \simeq 0$
$(\mu.p) \setminus \alpha \simeq 0$ if $\{\alpha, \overline{\alpha}\}$	
$(\mu.p) \setminus \alpha \simeq \mu.(p \setminus \alpha)$ if $\mu \notin \{\alpha, \overline{\alpha}\}$	
$p \setminus \alpha \setminus \alpha \setminus \simeq p \setminus \alpha$	

Table 1: Equivalence Laws for strong bisimilarity

139) What is the syntax of Hennessy-Milner logic?
The Hennessy-Milner logic, also known as HML, is defined by the following syntax:

$\langle F \rangle ::= \mathit{true} \mid \mathit{false} \mid \wedge_{i \in I} \langle F \rangle_i \mid \vee_{i \in I} \langle F \rangle_i \mid \Diamond_\mu \langle F \rangle \mid \Box_\mu \langle F \rangle$

The formulas of HML express properties over the states of an LTS. The meanings of the logic operators are the following:

- true: is the formula satisfied by every agent.
- false: is the formula never satisfied by any agent.
- $\wedge_{i \in I} F_i$: corresponds to the conjunction of the formulas $\{F_i\}_{i \in I}$;
- $\vee_{i \in I} F_i$: corresponds to the disjunction of the formulas $\{F_i\}_{i \in I}$;
- $\Diamond_\mu F$: is a modal operator; an agent p satisfied this formula if there exists a μ-labelled transition from p to some state q that satisfied the formula F.
- $\Box_\mu F$: is a modal operator; an agent p satisfied this formula if for any q such that there is a μ-labelled transition from p to q the formula F is satisfied F by q.

140) What is the converse of a HML formula?
The converse of a HML formula is analogous to the negation operator in a boolean formula. Given any HML formula F, we can easily compute another formula F^c such that:

$$\forall p \in \mathrm{P}.p \models F \iff p \not\models F^c$$

The converse formula F^c is defined by structural recursion as follows:

$$\text{true}^c = \text{false} \quad \text{false}^c = \text{true}$$
$$(\wedge_{i\in I} F_i)^c = \vee_{i\in I} F_i^c \quad (\vee_{i\in I} F_i)^c = \wedge_{i\in I} F_i^c$$
$$(\Diamond_\mu F)^c = \Box_\mu F^c \quad (\Box_\mu F)^c = \Diamond_\mu F^c$$

141) How is the satisfaction relation defined for HML formulas?
The *satisfaction relation* $\models \subseteq \mathcal{P} \times \mathcal{L}$ is defined as follows (for any $p \in \mathcal{P}, F \in \mathcal{F}$ and $\{F_i\}_{i\in I} \subseteq \mathcal{L}$):

$$p \models \text{true}$$
$$p \models \wedge_{i\in I} F_i \iff \forall i \in I.p \models F_i$$
$$p \models \vee_{i\in I} F_i \iff \exists i \in I.p \models F_i$$
$$p \models \Diamond_{i\in I} F_i \iff p'.p \xrightarrow{\mu} p' \wedge p' \models F$$
$$p \models_{i\in I} F_i \iff \forall p'.p \xrightarrow{\mu} p' \implies p' \models F$$

If $p \models F$ we say that the process p satisfied the HML-formula F.

142) How is HML logical equivalence between processes defined?
Let p, q be two CCS processes. We say that p and q are *HML-logic equivalent*, written $p \equiv_{\text{HM}} q$ if

$$\forall F \in \mathcal{L}.p \models F \iff q \models F$$

143) What is the relation between HML logical equivalence and strong bisimilarity?

Theorem 5 *For any finitely branching processes $p, q \in P$:*

$$p \simeq q \iff p \equiv_{HM} q$$

Consequences:

- to show that two processes are strong bisimilar: exhibit a strong bisimulation relation that relates them.
- to show that two processes are not strong bisimilar: exhibit a HML formula that distinguishes between them.

144) How to prove that two finitely branching processes are not strong bisimilar?
To show that two processes are not strong bisimilar: exhibit a HML formula that distinguishes between them.

145) What is a weak transition?
A weak transition is defined as follows:

- for the silent transitions τ we have:

$$p \stackrel{\tau}{\Rightarrow} q \iff p(\xrightarrow{\tau})^{*q}$$

- for the labelled transitions λ we have:

$$p \stackrel{\lambda}{\Rightarrow} q \iff \exists p, q'.p \stackrel{\tau}{\Rightarrow} p' \xrightarrow{\lambda} q' \stackrel{\tau}{\Rightarrow} q$$

146) How is the weak bisimulation game defined?
In the weak bisimulation game, the defender has more power! The game proceeds as follows:

- Alice picks a process and an ordinary transition;
- Bob replies possibly using many additional silent tranistions (arbitrarily many, but finitely many, such sequences are called *weak* transitions);
- what if Alice picks a silent transition?;
- Bob can just leave the other process idle(i.e. can choose to to move).

147) What is a weak bisimulation?
Given an LTS $(P, \Lambda, \rightarrow)$ where P is a set of *processes*, Λ a set of labels and the relation $\rightarrow \subseteq (P \times \Lambda \times P)$, we say that a relation $\mathbf{R}$ is a **weak bisimulation** if:

$$\forall p, q.(p,q) \in \mathbf{R} \implies \begin{cases} \forall \mu, p'.p \xrightarrow{\mu} p' \implies \exists q'.q \stackrel{\mu}{\Rightarrow} q' \wedge p' \mathbf{R} q' \\ \wedge \\ \forall \mu, q'.q \xrightarrow{\mu} q' \implies \exists p'.p \stackrel{\mu}{\Rightarrow} p' \wedge p' \mathbf{R} q' \end{cases}$$

148) What is a weak bisimilarity?
We define weak bisimilarity with

$$\approx \triangleq \bigcup_{\mathbf{R} \text{ is a w.b.}} \mathbf{R}$$

Two processes p, q are weak bisimilar when

$$p \approx q \text{ iff } \exists . \mathbf{R} \text{ a weak bisimulation such that } (p,q) \in \mathbf{R} \ .$$

149) Is weak bisimilarity an equivalence relation?
Yes, weak bisimilarity is an *equivalence relation* (there exists a theorem analogous to the strong one).

150) Is weak bisimilarity a weak bisimulation?
Proof.
Take $p \approx q$.

Take $p \stackrel{\mu}{\Rightarrow} p'$ we want to prove that $\exists\, q' .\, q \stackrel{\mu}{\Rightarrow} q'$ such that $p' \approx q'$.
$p \approx q$ means that exists a w.b. $\mathbf{R}$ such that $p \ \mathbf{R} \ q$.
Since $\mathbf{R}$ is a w.b. and $p \stackrel{\mu}{\Rightarrow} p'$ we have $q \stackrel{\mu}{\Rightarrow} q'$ with $(p', q') \in \mathbf{R}$.
Since $\mathbf{R} \subseteq \approx$ we have $p' \approx q'$.

Take $q \stackrel{\mu}{\Rightarrow} q'$ we want to prove that $\exists\, p' .\, p \stackrel{\mu}{\Rightarrow} p'$ such that $p' \approx q'$.
$p \approx q$ means that exists a w.b. $\mathbf{R}$ such that $p \ \mathbf{R} \ q$.
Since $\mathbf{R}$ is a w.b. and $q \stackrel{\mu}{\Rightarrow} q'$ we have $p \stackrel{\mu}{\Rightarrow} p'$ with $(p', q') \in \mathbf{R}$.
Since $\mathbf{R} \subseteq \approx$ we have $p' \approx q'$.

151) Why can weak bisimilarity be more convenient to use than strong bisimilarity?
Consente di astrarre dalle transizioni τ che sono silenti e non osservabili, in modo tale da poter individuare più processi bisimili.

152) Why is weak bisimilarity not a congruence?
Weak bisimilarity is not a congruence. For instance, let's take $P \triangleq \alpha.0, Q \triangleq \tau.\alpha.0$. If $P \xrightarrow{\alpha} 0$ then $Q \stackrel{\alpha}{\Rightarrow} 0$. If $Q \xrightarrow{\tau} \alpha$ then $P \stackrel{\tau}{\Rightarrow}$. Now, if we take the following context with a hole:

$$\mathbb{C}[\cdot] = [\cdot] + \beta$$

and let Alice and Bob play, we obtain the following state:

- Alice plays: $\mathbb{C}[Q] \xrightarrow{\tau} \alpha$;
- Bob can only reply: $\mathbb{C}[P] \stackrel{\tau}{\Rightarrow} \mathbb{C}[P]$;
- Alice plays: $\mathbb{C}[P] \xrightarrow{\beta} 0$;
- Bob cannot reply: $\alpha \stackrel{\beta}{\not\Rightarrow}$;
- Therefore, Alice wins!

153) What is the weak observational congruence?
The weak observational congruence is the following law:

$$p \cong q \iff p \approx q \wedge \forall r. p + r \approx q + r$$

At the level of bisimulation game: Bob is not allowed to use an idle move at the very first turn.

154) What are Milner's tau laws?
Milner's τ-laws are:

- $p + \tau.p \cong \tau.p$
- $\mu.(p + \tau.q) \cong \mu.(p + \tau.q) + q$
- $\mu.\tau.p \cong \mu.p$

155) How can imperative programs be encoded in CCS?
CCS can be used to model imperative programs. The encoding is not pretty straightforward and requires some attention to it. We begin introducing a new process called Done $= \overline{\text{done}}.0$ which sends an output message on the channel done when a command terminate its execution.
The command `skip` does nothing, so we can encode it τ.Done. The variables are represented as agents which operates in parallel. Essentially, they are servers which can be queried to ask a value, or we can communicate the new value. Assuming we have a finite set of possible values, we can use the channel xr_i to read a value from the variable x and we can write any value using the channel $xw_i.X_i$:

$$XW \triangleq \sum_{i=1}^{n} xw_i.X_i$$
$$X_i \triangleq XW + xr_i.X_i$$

To declare a variable in the form of `var x` we can use the processes $XW|$Done (is a sort of spawn of the variable handler, i.e. the variable server). To model the assignment, we use the process $\overline{xw_i}|Done$ (we emit in output that we want to change the value of a variable, and then we send *done*

to signal the system).
The conditional command `if x = v_i then c1 else c2` is modelled as follows (supposing that p_i models c_i):

$$xr_i + p_1 + \sum_{j \neq i} xr_j.p_2$$

The iteration `while x = v_i do c` is a bit tricky, but it can be modelled as follows (supposing that p models c):

$$Y \triangleq xr_i.(p \frown Y) + \sum_{j \neq i} xr_j.\text{Done}$$

156) How can sequential composition of commands be encoded in CCS?
To encode the sequential composition of two commands, $c_1; c_2$, let us suppose to have p_1 models c_1 and p_2 models c_2. Right now, each processed that we modelled so far emits a $\overline{done}$ output message to signal the termination of a command. So a naive approach would be to model the composition as follows:

$$p1|\overline{done}.p2$$

That is, when $p1$ terminates (emitting the *done* message), performs $p2$. The big problematic with this approach is that it does not scale well (just compose a third command, the third process p_3 could capture the done message destined for $p2$!). Therefore, we have to use a relabelling to limit the scope of the message, between only the two processes p_1 and p_2. To do so, we use the relabelling and restriction operators as follows:

$$\phi_d(\text{done}) = d \qquad p_1 \frown p_2 = (p_1[\phi_d]|d.p2) \setminus \{d\}$$

Now other processes cannot capture the done message destined for p_2, we say that the message d is local to p_1 and p_2 (the scope).

157) How can CCS be used to verify program properties?
We saw how to encode an imperative program in CCS. We can use a concurrent workbench tool (as CAAL for instance) to verify some useful program properties. For example, during the lectures, we saw how to model the *Peterson* algorithm for mutual exclusion and see if it respects the *deadlock-free* and *starvation-free* properties.

158) What is CAAL?
CAAL is a web-based tool developed in Aalborg (Denmark) to model concurrent systems, visualize and verify them. CAAL stands for Concurrency Workbench Aalborg Edition.

159) What are the features implemented in CAAL?
CAAL allows to:

- *visualize* concurrent processes;
- *model* concurrent processes;
- *verify* concurrent processes, seeing if some HML formula are valid and if two processes are Strong/Weak bisimilar or Trace Equivalent;
- *play* the bisimulation game.

160) Which equivalences can be checked using CAAL?
In CAAL we can check *strong/weak/trace* equivalences.

161) Which formulas can be checked using CAAL?
In CAAL we can check HML formulas.

162) What is the syntax of LTL?
The syntax of **Linear Temporal Logic** is the following one:

$$\phi ::= \text{true} \mid \text{false} \mid \neg\phi \mid \phi_0 \vee \phi_1 \mid \phi_0 \wedge \phi_1 \mid p \mid O\phi \mid F\phi \mid G\phi \mid \phi_0 U \phi_1$$

where $p \in P$ is any atomic proposition. The meaning of each *temporal operator* is explained as follows:

- O is called the *next operator*. The formula ϕO means that ϕ is true in the next state.
- F is called the *finally operator*. It means that ϕ will hold sometime in the future.
- G is called the *global operator*. It means that ϕ it will be always valid in the future.
- U is called the *until operator*. It means that ϕ_0 is true until the first time ϕ_1 is true.

163) What is a linear structure?
A **linear structure** is a *pair* (S, P), where P is a set of *atomic propositions* and $S : P \to \mathfrak{P}(\mathbb{N})$ is a function assigning to each proposition $p \in P$ the set of times instants in which it is valid:

$$\forall p \in P.S(p) = \{n \in \mathbb{N} | n \textit{ satisfies } p\}$$

164) How is the satisfaction relation defined for LTL formulas?
Given a *linear structure*, (S, P), we define the *satisfaction relation* $\models$ for LTL formulas by structural induction:

$S \models \text{true}$	
$S \models \neg\phi$	if it is not true that $S \models \phi$
$S \models \phi_0 \wedge \phi_1$	$S \models \phi_0 \wedge S \models \phi_1$
$S \models \phi_0 \vee \phi_1$	$S \models \phi_0 \vee S \models \phi_1$
$S \models p$	$0 \in S(p)$
$S \models \phi$	$S^1 \models \phi$
$S \models F\phi$	$\exists k \in \mathbb{N}$such that $S^k \models \phi$
$S \models G\phi$	$\forall k \in \mathbb{N}$it holds $S^k \models \phi$
$S \models \phi_0 U \phi_1$	if $\exists k \in \mathbb{N}$ such that $S^k \models \phi_1$ and $\forall i < k.S^i \models \phi_0$

165) When are two LTL formulas equivalent?
Two Linear Temporal Logic formulas ϕ and ψ are equivalent, $\phi \equiv \psi$, if for any S we have $S \models \phi$ and $S \models \psi$.

166) How can Finally be expressed using Until?
The *finally* operator can be expressed using the until operator in the following way:

$$F\phi \equiv \text{true}\, U \phi$$

167) How can Globally be expressed using Finally?
The *globally* operator can be expressed using the finally operator in the following way:

$$G\phi \equiv \neg(F\neg\phi)$$

168) What is the syntax of CTL*?
The Computational Tree Logic* is a temporal logic which makes use of tree to model the time. CTL* includes two new operators, which are: "*possibly*" (E) and "*inevitably* (*always*)" (A) (these two are called **path operators)**:

$$\phi ::= \text{true} \mid \text{false} \mid \neg\phi \mid \phi_0 \vee \phi_1 \mid \phi_0 \wedge \phi_1 \mid p \mid O\phi \mid F\phi \mid G\phi \mid \phi_0 U \phi_1 \mid E\phi \mid A\phi$$

The meaning of the path operator is explained as follows:

- E: the formula $E\phi$ means that there exists some path that satisfies ϕ.
- A: the formula $A\phi$ means that each path of the tree satisfies ϕ.

169) What is a branching structure?
A **branching structure** is a *triple* (T, S, P) where P is a set of *atomic propositions*, T is an *infinite tree* and $S : P \to V$ is a function from the atomic propositions to subsets of nodes of V defined as follows:

$$\forall p \in P.S(p) = \{x \in V | x \text{ satisfies } p\}$$

In CTL* computations are described as *infinite paths* on *infinite trees*.

170) How is the satisfaction relation defined for CTL* formulas?
Given a *branching structure*, (T, S, P) and $\phi = v_0 v_1 \ldots$ be an *infinite path*. We define the *satisfaction relation* $\models$ for LTL formulas by structural induction (see Table 2).

$S, \pi \models \text{true}$	
$S, \pi \models \neg\phi$	if it is not true that $S, \pi \models \phi$
$S, \pi \models \phi_0 \wedge \phi_1$	$S, \pi \models \phi_0 \wedge S, \pi \models \phi_1$
$S, \pi \models \phi_0 \vee \phi_1$	$S, \pi \models \phi_0 \vee S, \pi \models \phi_1$
$S, \pi \models p$	$v_0 \in S(p)$
$S, \pi \models \phi$	$S, \pi^1 \models \phi$
$S, \pi \models F\phi$	$\exists k \in \mathbb{N}$such that $S, \pi^k \models \phi$
$S, \pi \models G\phi$	$\forall k \in \mathbb{N}$it holds $S, \pi^k \models \phi$
$S, \pi \models \phi_0 U \phi_1$	if $\exists k \in \mathbb{N}$ such that $S, \pi^k \models \phi_1$ and $\forall i < k.S, \pi^i \models \phi_0$
$S, \pi \models E\phi$	if there exists $\pi' = v_0, v'_1, v'_2, \ldots$ such that $S, \pi' \models \phi$
$S, \pi \models A\phi$	if for all paths $\pi' = v_0, v'_1, v'_2, \ldots$ we have $S, \pi' \models \phi$

Table 2: Satisfaction relation for CTL* formulas.

171) When are two CTL* formulas equivalent?
Two CTL* formulas, ϕ and ψ, are *equivalent*, written $\phi \equiv \psi$, if for any S, π we have $S, \pi \models \phi$ if and only if $S, \pi \models \psi$.

172) How can Always be expressed using Possibly?
The *always* path operator can be expressed using *possibly* as follows:

$$\neg E\neg\phi \equiv A\phi$$

173) How is CTL defined?
The formulas of CTL are obtained by restricting CTL*. Let $\{O, F, G, U\}$ be the *linear* operators and $\{E, A\}$ the set of *path* operators. A CTL* formula is a *CTL formula* if all the following holds:

1. each path operator appears only immediately before a linear operator.
2. each linear operator appears immediately after a path operator.

174) What is a minimal set of operators for CTL?
All the CTL formulas can be written in terms of the *minimal set* of operators *true*, $\neg$, $\vee$, EG, EU, EO.

175) What is the relation between the expressive powers of LTL, CTL, CTL*?
CTL and LTL are both subsets of CTL*, but they are *not equivalent* to each other:

1. no CTL formula is equivalent to the LTL formula FGp;
2. no LTL formula is equivalent to the CTL formula $AG(p \implies (EOq \wedge EO\neg q))$.

Moreover, the *fairness* (properties expressing that something good will happen infinitely often) is not expressible in CTL.

176) What is the syntax of the mu-calculus?
The syntax of μ-Calculus is the following one:

$$\phi ::= \text{true} \mid \text{false} \mid \neg\phi \mid \phi_0 \vee \phi_1 \mid \phi_0 \wedge \phi_1 \mid p \mid \neg p \mid x \mid \diamond \phi \mid \Box\phi \mid \mu x.\phi \mid \nu x.\phi$$

where $p \in P$ is any atomic proposition and $x \in X$ is any predicate variable. The box and diamond operators mean:

1. diamond: there is a next state where ϕ holds;
2. square: ϕ holds at every next state.

177) Why only positive normal forms are considered?
Since there are the minimal/maximum fixpoint operators, the negation is not allowed in the syntax. The motivation relies on the fact that we are using monotone functions, and the negation *is not monotone* at all. For instance, if we have a set of states $S_1 \subseteq S_2$ and we apply the negation, we would obtain $\neg S_2 \subseteq \neg S_1$, therefore the monotonicity is not valid (the negation operator here gives us the complementary set, where a formula does not hold). This is the reason because we use the *normal positive form* for formulas.

178) How is the semantics of mu-calculus formulas defined?
The semantics of mu-calculus uses a graph $G = (V, \to)$. The interpretation of a formula $[\![\phi]\!]\rho$ gives us the set of states v where ϕ is valid. The *environment* $\rho : P \cup X \to \mathcal{P}(V)$ tell us in which nodes a given proposition (o predicate variable) is valid. The variables are needed to solve the recursive equations. Since we are solving recursive equations, we require a CPO with bottom where to interpret the functions $(\mathcal{P}(V), \subseteq)$. Given the set of formula $\mathcal{F}$. The interpretation function is defined as follows:

$$[\![\cdot]\!] : \mathcal{F} \to ((P \cup X) \to \mathcal{P}(V)) \to \mathcal{P}(V)$$

The function is defined by structural induction:

$$
\begin{aligned}
&[\![\text{true}]\!]\rho = V \\
&[\![\text{false}]\!]\rho = \emptyset \\
&[\![\phi_0 \wedge \phi_1]\!]\rho = [\![\phi_0]\!] \cap [\![\phi_1]\!] \\
&[\![\phi_0 \vee \phi_1]\!]\rho = [\![\phi_0]\!] \cup [\![\phi_1]\!] \\
&[\![p]\!]\rho = \rho(p) \\
&[\![\neg p]\!]\rho = V \setminus \rho(p) \\
&[\![x]\!]\rho = \rho(x) \\
&[\![\Box\phi]\!] = \{v | \exists w \in [\![\phi]\!]\rho . v \to w\} \\
&[\![\diamond\phi]\!] = \{v | \forall w . v \to w \implies w \in [\![\phi]\!]\rho\} \\
&[\![\mu x.\phi]\!] = \text{fix}(\lambda S.[\![\phi]\!][S/x]) \\
&[\![\nu x.\phi]\!] = \text{FIX}(\lambda S.[\![\phi]\!][S/x])
\end{aligned}
$$

Where we can compute the fixpoint functions as follows:

$$
\begin{aligned}
\text{fix} f &= \bigcup_{n \in \mathbb{N}} f^n(\bot) \\
\text{FIX} f &= \bigcap_{n \in \mathbb{N}} f^n(V)
\end{aligned}
$$

179) How to express deadlock freedom in the mu-calculus?
Through μ-calculus, we can express deadlock freedom using the following formula:

$$\nu x.\neg(\Box false \vee \diamond\neg x) = \nu x. \diamond \text{true} \wedge \Box x$$

That is read as: 'I am able to perform a move and in the next instant I can perform x'.

180) How to mutual exclusion in the mu-calculus?
The mutual exclusion in the mu-calculus can be expressed using the following formula:

$$\mu x.p \vee \diamond x$$

181) How to write an invariant property in the mu-calculus?
To express the *invariant* property we use the following formula:

$$\text{Invariant} \triangleq \nu x.\phi \wedge \Box x$$

182) How to write a possible formula in the mu-calculus?
To express the *possible* property we use the following formula:

$$\text{Possibly} \triangleq \mu x.\phi \vee \diamond x$$

183) How can bisimilarity be exploited in model verification?
We can exploit bisimilarity to check if a given *implementation* of a process satisfies a given *specific*.

184) How are goroutines launched in Google Go?
The goroutines are a concurrent primitive of go. They are a lightweight thread managed by the Google Go Runtime. The goroutine runs in the same space address of the thread started the execution.

```
go function_name(params)
```

185) How is a channel created in Google Go?
A channel in Go is created using the `make` primitive to allocate it.

```
// A channel sending/receiving integers
ich := make(chan int)

// Buffered channel of 16 integers
bich := make(chan int, 16)
```

186) How is asynchronous communication possible in Google Go?
In GoLang the asynchronous communication is possible using the channel mechanism, such as, send and receive. The channel are first-value citizens.

187) What are the possible decorations for channel types in Google Go?
A channel can be specialized only for *receiving* or *sending*:

```
// Only receiving
func foo(rchi <-chan int) {
 // ...
}

// Only sending
func bar(schi chan<- int) {
 // ...
}
```

188) What is the syntax for sending messages in Google Go?

```
chi <- value
```

189) What is the syntax for receiving messages in Google Go?

```
v := <- chi
```

190) What is the syntax and meaning of the select construct in Google Go?
(From the *GoLang Language Specification*). A `select` statement chooses which of a set of possible *send* or *receive* operations will proceed. It looks similar to a switch statement, but with the cases all referring to communication operations.
Execution of a select statement proceeds in several steps:

1. For all the cases in the statement, the channel operands of receive operations and the channel and right-hand-side expressions and send statements are evaluated exactly only once, in source order, upon entering the "select" statement. If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform *pseudorandom* selection. Otherwise, if there is a default case, that case is chosen. If there is no default case, the "select" statement blocks until at least one of the communications can proceed.

2. Unless the selected case is the default case, the respective communication operation is executed.

3. If the select case is a `receive` statement with a short variable declaration or an assignment, the left-hand side expressions are evaluated and the received value (or values) are assigned.

4. The statement list of the selected case is executed.

The syntax of the `select` construct is the following one:

```
var a []int
var c, c1, c2, c3, c4 chan int
var i1, i2 int
select {
case i1 = ←c1:
  print("received ", i1, " from c1\n")
case c2 ← i2:
  print("sent ", i2, " to c2\n")
case i3, ok := (←c3):  // same as: i3, ok := ←c3
  if ok {
    print("received ", i3, " from c3\n")
  } else {
    print("c3 is closed\n")
  }
case a[f()] = ←c4:
  // same as:
  // case t := ←c4
  //  a[f()] = t
default:
  print("no communication\n")
}

for {  // send random sequence of bits to c
  select {
  case c ← 0:  // note: no statement, no fallthrough, no folding of cases
  case c ← 1:
  }
}
```

191) What is the syntax of pi-calculus?

The syntax of pi-calculus is the following one:

π ::=

- $\overline{x}y$ send y on x
- $| \; x(y)$ receive on and store in y
- $| \; \tau$ internal action

p ::=

- 0 inactive process
- $| \; \pi.p$ action prefix
- $| \; [x = y]p$ matching
- $| \; p + p$ non-determinism
- $| \; p|p$ parallel composition
- $| \; (y)p$ restriction, creation of a **new** channel
- $| \; !p$ replication, **bang** operator

The replication operator is like a server for requests. The only binders in the pi-calculus are the *input prefix* and *name restriction* operators. They act as we are declaring a new variable in an imperative language. The notion of capture avoiding substitution is defined as usual.

192) What is scope extrusion?
The *scope extrusion* is an "extension" of the scope of a named channel to another running process. Think to the example seen during the lectures, when the channel ab is moved from A to S and then from S to B.

193) What is the style of the operational semantics of the pi-calculus?
As for the CCS, the pi-calculus uses a *small-step* operational semantics using an LTS.

194) What are the labels of the operational semantics of the pi-calculus?
The labels to consider in the operational semantics of the pi-calculus are the following one:

- τ: the silent transition;
- $\overline{x}y$: the sending of value y using the channel x;
- $x(y)$: the receiving of the value y from the channel x. Be aware, y is a bound name (or a *fresh name*, which is a sort of placeholder, or *symbolic name*, because we will know the real value when the system is running).
- $\overline{x}(y)$: bound output, we are transmitting a fresh name (a symbolic one), see below in the (Open) rule.

195) What are the rules of the operational semantics of the pi-calculus?

The rules for operational semantics are the following one:

$$\frac{}{\pi.p \xrightarrow{\pi} p} \textbf{(Act)}$$

$$\frac{p \xrightarrow{\alpha} q}{[x = x]p \xrightarrow{\alpha} q} \textbf{(Match)}$$

$$\frac{p_1 \xrightarrow{\alpha} q}{p_1 + p_2 \xrightarrow{\alpha} p_1} \textbf{(SumL)} \quad \frac{p_2 \xrightarrow{\alpha} q}{p_1 + p_2 \xrightarrow{\alpha} q} \textbf{(SumR)}$$

$$\frac{p_1 \xrightarrow{\alpha} q_1}{p_1|p_2 \xrightarrow{\alpha} q_1|p_2} \textbf{(ParL)} \quad \frac{p_2 \xrightarrow{\alpha} q_q}{p_1|p_2 \xrightarrow{\alpha} p_1|q_2} \textbf{(ParR)}$$

$$\frac{p_1 \xrightarrow{\overline{x}z} q_1 \quad p_2 \xrightarrow{x(y)} q_2}{p_1|p_2 \xrightarrow{\tau} q_1|(q_2[z/y])} \textbf{(ComL)} \quad \frac{p_1 \xrightarrow{x(y)} q_1 \quad p_2 \xrightarrow{\overline{x}z} q_2}{p_1|p_2 \xrightarrow{\tau} (q_1[z/y])|q_2} \textbf{(ComR)}$$

$$\frac{p \xrightarrow{\alpha} q}{(y)p \xrightarrow{\alpha} (y)q}\ y \notin \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)\ \textbf{(Restriction)}$$

$$\frac{p \xrightarrow{\overline{x}y} q}{(y)p \xrightarrow{\overline{x}(z)} q[z/y]}\ y \neq x \wedge z \notin \mathrm{fn}((y)p)\ \textbf{(Open)}$$

$$\frac{p_1 \xrightarrow{\overline{x}(z)} q_1 \quad p_2 \xrightarrow{x(z)} q_2}{p_1|p_2 \xrightarrow{\tau} (z)(q_1|q_2)} \textbf{(CloseL)} \quad \frac{p_1 \xrightarrow{x(z)} q_1 \quad p_2 \xrightarrow{\overline{x}(z)} q_2}{p_1|p_2 \xrightarrow{\tau} (z)(q_1|q_2)} \textbf{(CloseR)}$$

$$\frac{p|!p \xrightarrow{\alpha} q}{!p \xrightarrow{\alpha} q} \textbf{(Replication 1)}$$

$$\frac{p|p \xrightarrow{\alpha} q}{!p \xrightarrow{\alpha} q|!p} \textbf{(Replication 2)}$$

Some notes:

- be aware of fresh names when considering the parallel rules.
- the most characteristic rule of pi-calculus is the *(Open)* rule. It is used when we are trying to transmit a restricted name to another one. The label in the conclusion allows the creation of a new fresh name $\overline{x}(z)$, in the target we remove the restriction temporarily, because the scope of the name y will be expanded when it will be received by another process. TLDR: the rule makes public a private channel name. The restriction will be placed by the (*CloseL/R*) rules.

- the rule (*Replication 1*) suffers from "*infinitely branching*", to avoid that see the rule (*Replication 2*).

196) What is a fresh name?
A fresh name is a symbolic name used to not capture free names during the substitution of a name with a new one in a process. For instance, if we have the following transition:

$$x(y).p \overset{x(y)}{\to} p$$

We have that the process $x(y).p$ is equivalent to the following one:

$$x(y).p = x(z).(p[z/y]) \text{ if } z \notin \text{fn}(p)$$

The z is a fresh name, a symbolic one. So, the transition becomes:

$$x(y).p \overset{x(z)}{\to} p[z/y] \text{ if } z \notin \text{fn}(p)$$

The name z acts as a placeholder, not a value!

197) What is the difference between early and late bisimulation?
Before explaining the differences between the two bisimulation, let us introduce the definition of early and late strong bisimulation. Let $R \subseteq P \times P$:

Definition 8 (Strong Early Bisimulation)

$$p \,\mathbf{R}\, q \implies \begin{cases} p \overset{\alpha}{\to} p' \implies \exists q'.q \overset{\alpha}{\to} q' \wedge p' \;\boldsymbol{R}\; q' & bn(\alpha) \cup fn(q) = \emptyset, \alpha \neq x(y) \\ p \overset{x(y)}{\to} p' \implies \forall z.\exists q'.q \overset{x(y)}{\to} q' \wedge p'[z/y] \,\mathbf{R}\, q'[z/y] & y \notin fn(q) \\ \textit{and vice versa for } q \end{cases}$$

Definition 9 (Strong Late Bisimulation)

$$p \,\mathbf{R}\, q \implies \begin{cases} p \overset{\alpha}{\to} p' \implies \exists q'.q \overset{\alpha}{\to} q' \wedge p' \;\boldsymbol{R}\; q' & bn(\alpha) \cup fn(q) = \emptyset, \alpha \neq x(y) \\ p \overset{x(y)}{\to} p' \implies \exists q'.q \overset{x(y)}{\to} q' \wedge \forall z.p'[z/y] \,\mathbf{R}\, q'[z/y] & y \notin fn(q) \\ \textit{and vice versa for } q \end{cases}$$

The biggest change from the two definitions are in the Bob moves. In the *strong early bisimulation* we are requesting that for all values z that can be received from the channel x, p' and q' will behave the same. For instance, if we assume that the value z can be an integer, we are requesting that for all integers p' and q' will act as the same. The name "early" derives from the moment the value z will be received: for each value that can we receive we are requesting that q is capable of performing an input $x(y)$ transiting to q' and that p' and q' behave the same, as we said before (the move of q can depend on the value z).

In the *strong late bisimulation*, Bob will not know the value of z that he has to simulate.

198) What is the relation between early and late bisimilarities?

Definition 10 (Early Bisimilarity) *$p \sim_E q$ if and only if exists a strong early ground bisimulation $\boldsymbol{R}$ with $p \,\mathbf{R}\, q$.*

Definition 11 (Late Bisimilarity) *$p \sim_L q$ if and only if exists a strong late ground bisimulation $\boldsymbol{R}$ with $p \,\mathbf{R}\, q$.*

If two processes are *early bisimilar* then they will be *late bisimilar*. Instead, the vice versa does not hold at all. The early bisimilarity is more **strict** than the late one.

$$P \sim_L Q \implies P \sim_E Q$$
$$P \sim_E Q \not\implies P \sim_L Q$$

- *early*: for every input, Bob can choose a different move: $\forall z.\exists q' \ldots$
- *late*: Bob must choose one move, valid for every input: $\exists q'.\forall z \ldots$

199) Why early/late bisimilarities are not congruences?
Early and Late bisimilarity *are not congruences* because there are some relation that are *not preserved due to the input prefix*. $\sim_L$ and $\sim_E$ are *equivalences*.

Questions not answered

200) What is a sigma-field?
201) What is a probability space?
202) How is conditional probability defined?
203) What is a random variable?
204) What is a stochastic process?
205) What is the Markov property?
206) What is time independence property?
207) What is a Markov chain?
208) What is a homogeneous Markov chain?

210) What is a DTMC?
221) How can a DTMC be represented as a matrix?
222) How can a DTMC be represented as a transition system?
223) What is a transition function?
224) How to compute the state distribution at a next instant of time?
225) How to compute the state distribution at a time t?
226) How to compute a finite path probability?
227) What is the steady state distribution?
228) What is an ergodic DTMC?
229) How is the steady state distribution computed for an ergodic DTMC?

230) How are the probability law and the probability density related?
231) What is the (negative) exponential distribution?
232) What is a CTMC?
233) What is the embedded DTMC of a CTMC?

234) What is the infinitesimal matrix generator of a CTMC?
235) How are stationary distributions computed for a CTMC?

236) How can the notion of bisimulation be exported to DTMC/CTMC?
237) How is CTMC bisimulation defined?
238) How is DTMC bisimulation defined?
239) What is the effect of deadlock states for DTMC bisimilarity?
240) What is a reactive probabilistic transition system?
241) How is reactive bisimulation defined?
242) What is the syntax of Larsen-Skou logic?
243) How is the notion of satisfaction defined for Larsen-Skou formulas?
244) What is the relation between reactive bisimilarity and Larsen-Skou logic?

245) What is the motivation for introducing PEPA?
246) What are the mutual benefits for Process Algebras and CTMC in PEPA?
247) What is the PEPA workflow?
248) What is the difference between qualitative and quantitative analysis?
249) What are the differences between CCS and CSP?
250) What is the syntax of PEPA?
251) What is the style of the operational semantics of PEPA?
252) What are the labels of the operational semantics of PEPA?
253) What are the rules of the operational semantics of PEPA?
254) What is the apparent rate of a process?

Extra

Definition 12 (Operational Semantics) *In the **operational semantics**, it is of interest how the effect of a computation is achieved. Some kind of abstract machine is first defined, then the operational semantics describes the meaning of a program in terms of the steps/actions that this machine executes. The focus of operational semantics is thus on states and state transformations.*

Definition 13 (Denotational Semantics) *In **denotational semantics**, the meaning of a well-formed program is some mathematical object (e.g., a function from input data to output data). The steps taken to calculate the output and the abstract machine where they are executed are unimportant: only the effect is of interest, not how it is obtained. The essence of denotational semantics lies in the principle of compositionality: the semantics takes the form of a function that assigns an element of some mathematical domain to each individual construct, in such a way that the meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings. The denotational semantics describes an explicit interpretation function over a mathematical domain. The interpretation function for a typical imperative language is a mapping that, given a program, returns a function from any initial state to the corresponding final state, if any (as programs may not terminate).*

Definition 14 (Small-Step Semantics) ***Small-step semantics** formally describes how individual steps of a computation take place on an abstract device, but it ignores details such as the use of registers and storage addresses.*

Definition 15 (Big-Step Operational Semantics) ***Big-step semantics*** *formally describes how the overall results of the executions are obtained. It hides even more details than the small-step operational semantics. Like small-step operational semantics, natural semantics shows the context in which a computation step occurs, and like denotational semantics, natural semantics emphasizes that the computation of a phrase is built from the computations of its sub-phrases.*

Definition 16 (Modal Logics) *This is not properly a definition, but an intuition on what modal logics are.* ***Modal logics*** *were conceived in philosophy to study different modes of truth, for example an assertion being false in the current world but possibly true in some alternate world, or another holding always true in all worlds*

Definition 17 (Type Systems) *(From: Types and Programming Languages by Benjamin C. Pierce). Type systems are lightweight formal methods. The definition gave by Pierce is the following one: "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.".*