Formal Methods & Functional Programming

Janis Hutz
https://janishutz.com

July 28, 2026

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash z :: \sigma \to (\tau_2, \tau_3)} \quad \overline{\Gamma \vdash y :: \sigma}}{\Gamma \vdash (z\ y) :: (\tau_2, \tau_3)}\ \text{App}}{\Gamma = y : \texttt{Int}, z : \tau_1 \vdash \texttt{fst}\ (z\ y) :: \tau_2}\ \text{fst}}{y : \texttt{Int} \vdash \lambda z.\texttt{fst}\ (z\ y) :: \tau_0}\ \text{Abs}}{\vdash \lambda y.\lambda z.\texttt{fst}\ (z\ y) :: \texttt{Int} \to \tau_0}\ \text{Abs} \qquad \dfrac{}{\vdash 0 :: \texttt{Int}}\ \text{Int}}{\vdash (\lambda y.\lambda z.\texttt{fst}\ (z\ y))\ 0 :: \tau_0}\ \text{App}$$

Constraints:

- $\tau_0 = \tau_1 \to \tau_2$
- $\tau_1 = \sigma \to (\tau_2, \tau_3)$
- $\tau_2 = \sigma = \texttt{Int}$

Thus: $\tau_0 = \texttt{Int} \to (\tau_2, \tau_3) \to \tau_2$

"*A funny quote by a professor - If you know one, please let me know*"

- Prof. Dr. Professor Name, 2026

FS2026, ETHZ

Summary of the Lecture Slides,
Overview over common exercise types

https://infsec.ethz.ch/education/ss2026/fmfp.html

Contents

1 Haskell

Haskell is a functional programming language. As such, its functions can be thought of as being similar to mathematical functions and as such are side-effect-free.

Haskell's type system is very robust and an interesting topic to learn about. The basic data types you already know from other programming languages are also present here. This includes all primitives like integers, floating point numbers, chars and booleans.

Strings are handled similarly to how `C` does it, in that strings are char arrays.

Arrays however are dynamic length in Haskell as opposed to many other statically typed programming languages.

Since Haskell is an imperative language (i.e. you describe *what* you want achieve) as opposed to a declarative language (i.e. you describe *how* you achieve what you want to achieve), there are no loops in Haskell, as loops don't appear in mathematical formulas and functions either. What we can do however is recursion and this is the main way of doing iterative work in Haskell.

Additionally, Haskell features *lazy evaluation* (i.e. statements are evaluated only as needed) as opposed to *eager evaluation* (i.e. statemetns are evaluated immediately).

In this course the `Glasgow Haskell Compiler`, short `ghc` is used. Installation is really easy (as long as you're on Linux)

1.1 The bad news: The syntax

In short: It's quite bad, but you will get used to it and some of the (arguably) poor looking syntax choices will start to make more sense.

You should use 2 space indents (yuck) and indents matter, just like in Python.

We can use binary functions in infix or prefix notation, i.e. `x ‘mod‘ z` and `mod x z` are equivalent.

For integers the following functions are available: Normal arithmetic operations `+, -, *, /, mod, abs`, as well as `^` which is used for exponentiation.

To use prefix notation on non-alphanumeric function names, wrap them in parenthesis like this: `(+) x z`. Using `+ x z` does not work.

We can use the normal comparison operators that return a boolean on evaluation. ***Booleans*** are `True` and `False`

2 Induction Proofs

To prove recursive formulas, or more precisely formulated, a formula P (with free variable n) for all $n \in \mathbb{N}$, we can use weak or strong induction. Weak induction may be a *slightly* misleading term, because it isn't necessarily weaker than strong induction, for mathematical induction, for structural induction, there is a difference.

This section has been moved to the very start of the theory part, even though many of the topics mentioned have not been covered in the summary yet, such that all the induction proofs can be covered in the same place.

2.1 Mathematical Induction

To prove something of the form $\forall n \in \mathbb{N}.P$ (with n free in P), we can use one of these two schemes

2.1.1 Weak Mathematical Induction

If P is not defined yet (e.g. we just have a description of what we need to prove), give a full definition of P, which is our induction hypothesis. Then state that we are proving $\forall n \in \mathbb{N}.P$ using *weak* mathematical induction.

Base case We show that $P[n \mapsto 0]$ holds.

Step case We proceed by stating the following: "Let $m \geq 0$ be arbitrary. We assume $P[n \mapsto m]$ holds, and we prove $P[n \mapsto m+1]$." Finally, do the actual proof. It is also possible to state $P[n \mapsto m]$ simply as $P(m)$

The same, but expressed as a Natural Deduction rule:

$$\frac{\Gamma \vdash P(0) \qquad \Gamma, P(n) \vdash P(n+1)}{\Gamma \vdash \forall n.P(n)} \; n \text{ not free in } \Gamma$$

2.1.2 Strong Mathematical Induction

If P is not yet defined, then define it. It is our induction hypothesis. Then state: "We prove $\forall k.P(k)$ by strong (mathematical) induction. Let k be arbitary and let us assume $P(j)$ for all $j < k$". Finally, do case distinction on the different values k can take.

The same, but expressed as a Natural Deduction rule:

$$\frac{\Gamma, \forall m < n.P(m) \vdash P(n)}{\Gamma \vdash \forall n.P(n)} \; n \text{ not free in } \Gamma,\ m \text{ not free in } P(n)$$

2.2 Structural Induction

In this subsection, we present structural induction over lists. The same concept also applies similarly to trees and more. Weak Structural Induction on trees is presented in the subsequent subsection.

For both, we only provide the natural deduction rules, as the proofs proceed analogously to the ones for mathematical induction.

2.2.1 Weak Structural Induction

$$\frac{\Gamma \vdash P(\texttt{[]}) \qquad \Gamma, P(\texttt{xs}) \vdash P(\texttt{x : xs})}{\Gamma \vdash \forall \texttt{xs}.P(\texttt{xs})} \; \texttt{x, xs not free in } \Gamma$$

2.2.2 Strong Structural Induction

Definition 2.2.1 *(Subterm relation)* denoted $\sqsubset$ is defined as follows:

$$\forall xs, ys.xs \sqsubset ys \Leftrightarrow (\exists x.ys = x : xs) \vee (\exists zs.xs \sqsubset zs \wedge zs \sqsubset ys)$$

$$\frac{\Gamma, \forall ys \sqsubset xs.P(ys) \vdash P(xs)}{\Gamma \vdash \forall xs.P(xs)} \; xs \text{ not free in } \Gamma,\ ys \text{ not free in } P(xs)$$

2.2.3 Generalizing expressions

Sometimes, an expression can be hard to proof straight away because there is a constant in it. In that case, replace that constant, making sure to update both sides of the expression to reflect the change.

2.3 Induction on Trees

```
data T t = Leaf t | Node1 (T t) | Node2 t (T t) (T t)
```

Base Case $T_0 = \{\texttt{Leaf}\ a \mid a \in t\}$

Step Case $T_i = T_{i-1} \cup \{\texttt{Node1}\ s \mid s \in T_{i-1}\} \cup \{\texttt{Node2}\ a\ l\ r \mid a \in t \text{ and } l, r \in T_{i-1}\}$

A natural deduction rule structural induction is:

$$\frac{\Gamma \vdash P[x \mapsto \texttt{Leaf}\ a] \qquad \Gamma, P[x \mapsto s] \vdash P[xs \mapsto \texttt{Node1}\ s] \qquad \Gamma, P[x \mapsto l], P[x \mapsto r] \vdash P[\mapsto \texttt{Node2}\ a\ l\ r]}{\Gamma \vdash \forall x \in \texttt{T}\ t.P}\ (*)$$

(*) a, l, r, s not free in Γ, P

2.4 Other types of induction

2.4.1 Induction over Lists

To prove P for all xs in `[T]`, we do the following:

Base case We prove that $P[xs \mapsto []]$ is correct

Step case We prove that $\forall y :: T, ys :: [T].P[xs \mapsto ys] \to P[xs \mapsto y : ys]$, or in other words: We fix arbitrary $y :: T$ and $ys :: [T]$, which both are not free in P. We then apply our induction hypothesis $P[xs \mapsto ys]$ to prove $P[xs \mapsto y : ys]$

2.5 Induction on Proof Trees

This type of induction is used to prove that a statement exhibits given behaviour.

Typically, these are statements are of type $\forall \ldots \vdash \texttt{s1} \implies \vdash \texttt{s2}$.

Since often we are not restricted to just simple statements, such where we know that we are applying the Ass_{NS} rule, we need to perform case distinction on all possible last rules applied in the derivation tree T. If all are to be proven, this will yield 7, one for each rule of the big-step semantics.

When we have multiple options in a second stage that also differ from the premise, we may want use another case distinction there, drawing separate trees for them, or stating that we draw the common tree and then use a subtree T_N for the case distinction to reduce the amount of writing required.

Also be sure that you *always* mention the side conditions and also mention the Induction Hypothesis, if applicable or needed.

Definition 2.5.1 *(Subderivation)* We define $T' \sqsubset T$, where T is a derivation tree. T' is called a *subderivation* of T, or more simply, a *subtree* of T. Definition is analogous to the subterm relation.

Rundown Below a rundown of how such proofs are expected to be laid out.

We define $P(T) \equiv \forall \sigma, \sigma', s.(\texttt{root}(T) \equiv \texttt{s1} \implies \vdash \texttt{s2})$. The quantifier-bound variables / states / statements can of course differ, so adjust accordingly. This step is the crucial step to setting up this type of induction.

We then prove $\forall T.P(T)$ by strong induction on the shape of the derivation tree T. Thus, for some arbitrary T, the induction hypothesis is $\forall T' \sqsubset T.P(T')$ and we prove $P(T)$.

Let σ, σ', s (and more, if applicable) be arbitrary and assume the LHS of the implication.

Then, we show the RHS of the implication by case analysis on the last rule applied in T. We do this drawing up a derivation tree for each rule, such as WHT_{NS}, as shown below (from solutions of Exercise Session 11, FS2026):

$$\frac{\overset{T_4}{\langle s', \sigma\rangle \to \sigma''} \qquad \overset{T_5}{\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma''\rangle \to \sigma'}}{\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma\rangle \to \sigma'}\ (\text{WHT}_{NS})$$

After it, we need to put final constraints and bind free variables in some way, e.g. here:

For some $b, s', \sigma'', T_4, T_5$, such that $s \equiv \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}$ and $\mathcal{B}[\![b]\!]\sigma = \texttt{tt}$.

Then use the induction hypothesis on the subderivations T_i to obtain the desired result. That is, we show that there is another tree (or possibly derivation sequence), which fulfils the RHS with the same subtrees.

Depending on our Statement, we may need to provide a second tree for the RHS.

It is also important that it's ***always*** the last rule we are looking at. So if we are given a statement to prove where there are only a few options, we only need to prove for those options.

2.6 Induction on Derivation Sequence

A finite derivation sequence has a length $n \in \mathbb{N}$. We typically reason about finite derivation sequences using ***strong induction on the length of a derivation sequence***, i.e. we reason about a multi-step execution $\gamma \to_1^k \gamma'$ using strong induction on the number of steps k.

We define $P(k) \equiv$ "for all executions of length k, our property holds"

We then prove $P(k)$ for arbitrary k, with induction hypothesis $\forall k' < k.P(k')$, as is usually the case with induction on natural numbers.

After the setup, the proof *typically* proceeds by case distinction on:

- $k = 0$ step execution
- $k > 0$ step execution, by splitting off the first execution (i.e. an execution $\langle s, \sigma\rangle \to_1^k \sigma''$ can be split up as $\langle s, \sigma\rangle \to_1^1 \gamma \to_1^{k-1} \sigma''$.)

In the $k > 0$ case, we can then apply the induction hypothesis. Sometimes a further case distinction can become necessary, depending on the rules that are possible. For instance, if $s = s_1; s_2$, two rules are possible, SEQ1 and SEQ2 from the SOS.

2.7 Workflow

the notes in this section may seem a bit over the top, but in the exercise sessions it was mentioned repeatedly that they will heavily deduct points for not following the scheme. One example that was named is that in some exams out of 36 points for a proof, 9 were awarded for simply following the expected structure

1. Determine the exact expression to proof (possibly generalizing it, and if so, state something like "Consider the generalized statement")
2. Then, start the proof by stating something along the lines of "We prove $\forall n \in \mathbb{N} P$ by weak mathematical induction on n".
3. Depending on the proof type, proceed with base cases, or case analysis. Important: State "Let m be arbitrary" for free variables, and if there are extra variables introduced by the application (e.g. a `Leaf` type has value x in it), state "Fix x and we show $P[t \mapsto \texttt{Leaf}x]$"
4. Then, do the step case (if applicable). Important: State "Let n be arbitrary", or in the case of structural induction, there may be arguments for the type, such as for `Tree`, then state "Let `l::Tree` and `r::Tree` be arbitrary".

2.8 Proofs using CYP syntax

Using CYP (Check Your Proof) syntax is allowed at the exams and can be a bit less to write depending on your writing style. However, if you are very concise, you can achieve an even shorter version using a mix of CYP and non-CYP syntax.

In CYP, if doing it with pen and paper, we typically state that we use CYP for the proof, when doing it on the computer, we need a `defs.txt` file, containing all things we assume, as well as the function definitions, similar to Haskell syntax. Any property we don't want to, or don't have to prove, we can denote with `axiom axiom_name: definition of the axiom`. Finally, we state the proof's goal, exactly as the statement to prove: `goal statement`

If we have to generalize, we add a generalized lemma and prove it, then followed by proving the specific case

For the actual proofs, it start like this:

```
Lemma lemma_name: statement to prove

Proof by induction on DataStructureHere x generalizing y -- or any other variable, or possibly
  ↪ without generalization
Case Leaf -- or any other of course
    For fixed y -- only if we generalized
    Show: statement to prove for this case -- (e.g. substitute x with Leaft in this case)
    Proof
        -- The proof goes here
        statement
        (by def a_definition) .=. statement'
        (by any_axiom)        .=. statement''
    QED
```

```
13
14 Case (Node x y) -- another case
15     Fix x, y -- same as otherwise saying for arbitrary x, y
16     Assume
17         IH1: forall y: induction_hypothesis here
18         IH2: forall y: another IH here -- only needed for something like this, if there are
               ↪ two vars.
19         -- The second IH will (typically) be the same, simply a different var name
20     Show: statement to prove for this case
21     Proof
22     -- The proof (as above)
23     QED
24 QED
```

3 Formal Reasoning

3.1 Formal proofs

Given a language like $\mathcal{L} = \{\oplus, \otimes, +, \times\}$, and derivation rules

- α: If $+$, then $\otimes$
- β: If $+$, then $\times$
- γ: If $\otimes$ and $\times$, then $\oplus$
- δ: $+$ holds

or displayed using graphical notation:

$$\frac{+}{\otimes}\,\alpha \qquad \frac{+}{\times}\,\beta$$

$$\frac{\otimes \quad \times}{\oplus}\,\gamma \qquad \frac{}{+}\,\delta$$

Rules like δ above are also commonly referred to as an *axiom*.

To prove $\oplus$ in this language, we can either write the following or draw a derivation tree:

- $+$ holds by δ
- $\otimes$ holds by α with 1.
- $\times$ holds by β with 1.
- $\oplus$ holds by γ with 2 and 3

Or as derivation tree

$$\frac{\dfrac{}{+}\,\delta}{\otimes}\,\alpha \qquad \frac{\dfrac{}{+}\,\delta}{\times}\,\beta$$

$$\frac{\otimes \qquad \times}{\oplus}\,\gamma$$

3.2 Natural deduction

The rules from above here are used to construct derivations under assumptions, e.g. $A_1, \dots, A_n \vdash A$, which is read as "A follows from $A_1, \dots, A_n$".

The derivations are always represented as derivation trees and a ***proof*** is a derivation whose root has no assumptions.

Since we have to prove a statement, we have to draw the derivation trees from the bottom up, with the goal of reaching an axiom or a rule that is an assumption using the other rules of the rule set.

3.3 Propositional logic

3.3.1 Syntax

Definition 3.3.1 For a set of variables $\mathcal{V}$, the ***language of propositional logic*** $\mathcal{L}_P$ is the smallest set where

- $X \in \mathcal{L}_P$ if $X \in \mathcal{V}$
- $\bot \in \mathcal{L}_P$
- $A \wedge B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$
- $A \vee B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$
- $A \to B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$

3.3.2 Semantics

Definition 3.3.2 *(Valuation)* $\sigma : \mathcal{V} \to \{\texttt{True}, \texttt{False}\}$ maps variables to truth values. They are the simple models (i.e. *interpretations*). ***Valuations*** is the set of valuations.

Definition 3.3.3 *(Satisfiability)* smallest relation $\models \subseteq$ *Valuations* $\times \mathcal{L}_P$ such that

- $\sigma \models X$ if $\sigma(X) = \texttt{True}$
- $\sigma \models A \wedge B$ if $\sigma \models A$ and $\sigma \models B$
- $\sigma \models A \vee B$ if $\sigma \models A$ or $\sigma \models B$
- $\sigma \models A \to B$ if whenever $\sigma \models A$ then $\sigma \models B$

Definition 3.3.4 *(satisfiable formula)* A formula $A \in \mathcal{L}_P$ is ***satisfiable*** if $\sigma \models A$ for **some** valuation σ

Definition 3.3.5 *(tautology, valid formula)* A formula $A \in \mathcal{L}_P$ is ***valid*** (a ***tautology***) if $\sigma \models A$ for **all** valuations σ

Definition 3.3.6 *(Semantic entailment)* $A_1, \dots, A_n \models A$ if $\forall \sigma$ we have $\sigma \models A_1, \dots, \sigma \models A_n$, then $\sigma \models A$

3.3.3 Requirements for a deductive system

The derivation rules (syntactic entailment) and truth tables (semantic entailment) should agree. For that we have two requirements, for $\Gamma = A_1, \dots, A_n$ a collection of formulas:

- ***Soundness:*** If $\Gamma \vdash A$ can be derived, then $\Gamma \models A$
- ***Completeness:*** If $\Gamma \models A$, then $\Gamma \vdash A$ can be derived

Decidability is also desirable, i.e. having checks of the attributes be of low complexity.

3.3.4 Natural deduction for propositional formulas

Definition 3.3.7 *(Sequent)* Is an assertion of form $A_1, \dots, A_n \vdash A$, with $A, A_1, \dots, A_n$ being propositional formulas.

Intuition: A follows from the A_i and if the system is sound, the A_i semantically entail A.

Definition 3.3.8 *(Axiom)* is the starting point (usually the leaves) of the derivation trees and are usually of the form

$$\frac{}{\dots, A, \dots \vdash A} \text{ axiom}$$

i.e. when coming up with a derivation tree for a ***proof***, we want to reach a leaf where A is contained in Γ.

Definition 3.3.9 *(Proof)* of A is a derivation tree with $\texttt{root} \vdash A$. If a deductive system is *sound*, then A is a tautology.

There are two kinds of rules, ***introduce*** and ***eliminate*** connectives. If you are confused about the order when applying them when coming up with a deduction tree, they are oriented top-down, so e.g. the introduction rule is inverted when coming up with the deduction tree.

If all rules are sound (i.e. they preserve semantic entailment), then the logic is sound.

3.3.5 Derivation rules for propositional logic

Remember that E means *elimination* and I means *introduction*, with L and R being the side (so ER means elimination on the right)

3.3.5.1 Conjunction

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-I} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-EL} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-ER}$$

3.3.5.2 Disjunction

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-IL} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-IR} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee\text{-E}$$

3.3.5.3 Implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\text{-I} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to\text{-E}$$

3.3.5.4 Others

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\text{-E} \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B} \neg\text{-E} \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A} \text{ RAA} \qquad \frac{}{\dots, A, \dots \vdash A} \text{ axiom}$$

3.4 First-Order Logic

3.4.1 Syntax

Definition 3.4.1 *(Signature)* consists of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$, as well as their arities.

We write f^k (or p^k, for predicates) to indicate that it has *arity* $k \in \mathbb{N}$. Constant functions have arity 0, linear functions have arity 1, thus, the arity of a given function (or predicate) is given by the number of parameters to uniquely describe it, minus one.

Definition 3.4.2 *(Term)* is the terms of first-order logic and is the smallest set, where (with $\mathcal{V}$ again a set of variables)

1. $x \in$ *Term* if $x \in \mathcal{V}$
2. $f^n(t_1, \dots, t_n) \in$ *Term* if $f^n \in \mathcal{F}$ and $t_i \in$ *Term*, $\forall 1 \leq i \leq n$ (description of form of formulas)

Definition 3.4.3 *(Form)* is the formulas of first-order logic, is the smallest set where

1. $\bot \in$ *Form*
2. $p^n(t_1, \dots, t_n) \in$ *Form* if $p^n \in \mathcal{P}$ and $t_j \in$ *Term*, $\forall 1 \leq j \leq n$ (description of form of predicates)
3. $A \circ B \in$ *Form* if $A \in$ *Form*, $B \in$ *Form* and $\circ \in \{\wedge, \vee, \rightarrow\}$ (i.e. formulas with logic symbols)
4. $Qx.A \in$ *Form* if $A \in$ *Form*, $x \in \mathcal{V}$ and $Q \in \{\forall, \exists\}$ (i.e. formulas with quantifiers)

3.4.1.1 Binding

Definition 3.4.4 *(Bound variable)* A variable that occurs in a quantifier in scope (blue in example below)

Definition 3.4.5 *(Free variable)* A variable that is not bound by a quantifier in scope (red in example below)

Example 3.4.6 $(q(x) \vee \exists x.\forall y.p(f(x), z) \wedge q(a)) \vee \forall x.r(x, z, g(x))$

Definition 3.4.7 *(α-conversion)* A ***bound*** variable can be renamed at any time, but must preserve binding structure.

3.4.1.2 Precedences

$\neg > \wedge > \vee > \rightarrow$, with $>$ a total order of precedences. Quantifiers extend as far right as possible, bounded by the end of line or a going out of scope by closing parenthesis.

3.4.2 Semantics

Definition 3.4.8 *(Structure)* is a pair $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$, where $U_\mathcal{S}$ is the universe (non-empty set) and $I_\mathcal{S}$ is a mapping:

1. $I_\mathcal{S}(p^n)$ is an n-ary relation on $U_\mathcal{S}$ for $p^n \in \mathcal{P}$ (short $p^\mathcal{S}$)
2. $I_\mathcal{S}(f^n)$ is an n-ary (total) function on $U_\mathcal{S}$ for $f^n \in \mathcal{F}$ short ($f^\mathcal{S}$)

Intuition: The $I_\mathcal{S}$ is essentially assigning to each predicate and formula its definition in the universe of the structure, noted as a relation.

Definition 3.4.9 *(Interpretation)* is a pair $\mathcal{I} = \langle \mathcal{S}, v \rangle$, with $v : \mathcal{V} \rightarrow U_\mathcal{S}$ a valuation

Intuition: it assigns definitions to the formulas and predicates (through the structure), as well as values to the variables (through the valuation).

Definition 3.4.10 *(Value)* of a term t under $\mathcal{I}$ is written as $\mathcal{I}(t)$ and defined by $\mathcal{I}(x) = v(x)$ for $x \in \mathcal{V}$ and $\mathcal{I}(f(t_1, \dots, t_n)) = f^\mathcal{S}(\mathcal{I}(t_1), \dots, \mathcal{I}(t_n))$

Definition 3.4.11 *(Satisfiability)* $\models \subseteq$ Interpretations $\times$ *Form* is the smallest relation satisfying

$$\begin{aligned}
&\langle \mathcal{S}, v \rangle \models p(t_1, \dots, t_n) && \text{if } (\mathcal{I}(t_1), \dots, \mathcal{I}(t_n)) \in p^\mathcal{S} \text{ where } \mathcal{I} = \langle \mathcal{S}, v \rangle \\
&\langle \mathcal{S}, v \rangle \models \forall x.A && \text{if } \langle \mathcal{S}, v[x \mapsto a] \rangle \models A, \text{ for all } a \in \mathbb{U}_\mathcal{S} \\
&\langle \mathcal{S}, v \rangle \models \exists x.A && \text{if } \langle \mathcal{S}, v[x \mapsto a] \rangle \models A, \text{ for some } a \in \mathbb{U}_\mathcal{S}
\end{aligned}$$

Definition 3.4.12 *(Model)* When $\langle \mathcal{S}, v \rangle \models A$, then $\langle \mathcal{S}, v \rangle$ is a ***model*** for A. If A does not have free variables, the satisfaction does not depend on the valuation v and we write $\mathcal{S} \models A$

Definition 3.4.13 *(Validity)* When every interpretation is a model, we write $\models A$, and we say that A is ***valid***

Definition 3.4.14 *(Satisfiability)* A is ***satisfiable***, if there exists at least one model for A.

Example 3.4.15 Given $\forall x.p(x, s(x))$, we a model would be

$$\begin{aligned} U_{\mathcal{S}} &= \mathbb{N} \\ p^{\mathcal{S}} &= \{(m,n) \mid m,n \in U_{\mathcal{S}} \text{ and } m < n\} \\ s^{\mathcal{S}} &= \text{ successor function on } U_{\mathcal{S}}, \text{ i.e. } s^{\mathcal{S}}(x) = x+1 \end{aligned}$$

3.4.2.1 Substitution

Definition 3.4.16 *(Substitution)* Replace all occurrences of a free variable x with some term t in A. To denote a substitution, we write $A[x \mapsto t]$. Important All free variables in t must still be free in $A[x \mapsto t]$. If that would not be true anymore, do a α-conversion first.

3.4.3 Quantifiers

3.4.3.1 Universal quantification

Additional rules are needed for the universal quantifier. * side condition is that x is not free in any assumption of Γ

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ \forall\text{-I*} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \to t]}\ \forall\text{-E}$$

Again here be mindful not to capture free variables.

3.4.3.2 Existential quantification

Additional rules are needed for the existential quantifier. ** side condition is that x is neither free in B nor free in Γ

$$\frac{\Gamma \vdash A[x \mapsto t]}{\Gamma \vdash \exists x.A}\ \exists\text{-I} \qquad \frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash B}{\Gamma \vdash A[x \to t]}\ \exists\text{-E**}$$

Again here be mindful not to capture free variables.

3.5 Equality

Since equality is such an important concept, it isn't just a predicate, but a separate First-Order Logic (FOL), called ***FOL with equality***.

The language is extended by $t_1 = t_2 \in \mathit{Form}$ if $t_1, t_2 \in \mathit{Term}$, the semantic entailment $\models$ is also extended by "$\mathcal{I} \models t_1 = t_2$ if $\mathcal{I}(t_1) = \mathcal{I}(t_2)$". This definition is the exact intuition of equality of two terms, in that they are equal if their value under the interpretation is equal.

Equality is an equivalence relation, so the following rules apply (ref = reflexivity, sym = symmetry, trans = transitivity):

$$\frac{}{\Gamma \vdash t = t}\ \text{ref} \qquad \frac{\Gamma \vdash t = s}{\Gamma \vdash s = t}\ \text{sym} \qquad \frac{\Gamma \vdash t = s \qquad \Gamma \vdash s = r}{\Gamma \vdash s = r}\ \text{sym}$$

Equality is also a congruence on terms and (definable) relations

$$\frac{\Gamma \vdash t_1 = s_1 \qquad \dots \qquad \Gamma \vdash t_n = s_n}{\Gamma f(t_1, \dots, t_n) = f(s_1, \dots, s_n)}\ cong_1$$

$$\frac{\Gamma \vdash t_1 = s_1 \qquad \dots \qquad \Gamma \vdash t_n = s_n \qquad \Gamma \vdash p(t_1, \dots, t_n)}{\Gamma p(s_1, \dots, s_n)}\ cong_2$$

3.6 Correctness

For many programs, termination is an important aspect when talking about correctness, as is, of course, that the return value is "correct".

3.6.1 Termination

Theorem 3.6.1 For a function f is defined in terms of other functions $g_1, \dots, g_k$, for all of which we have $g_i \neq f$ and each g_i terminates, then so does f.

Lemma 3.6.2 Sufficient condition for termination: The arguments are smaller along a ***well-founded*** order on the function's domain

Definition 3.6.3 *(Well-Founded Order)* An order $>$ on a set S is ***well-founded*** if and only if there is no infinite decreasing chain $x_1 > x_2 > \dots$ for $x_i \in S$ An example of such an order is $>$ on $\mathbb{N}$, denoted $>_{\mathbb{N}}$. Counter example: $>_{\mathbb{Z}}$ (not bounded from below)

Lemma 3.6.4 Let $R \subseteq S \times S$ be a relation on S. Let $s_0, s_i \in S$ and $i \geq 1$. Then $s_0 \; R^i \; s_i$ if and only if $s_1, \dots s_{i-1} \in S$ such that $s_0 \; R \; s_1 \mathbb{R} \dots R \; s_{i-1} \; R \; s_i$

Definition 3.6.5 $R^+ \equiv \bigcup_{n \geq 1} R^n$, where $R^n \equiv R \circ R^{n-1}$ for $n \geq 2$ and $R^1 \equiv R$

Theorem 3.6.6 If $>$ is a well-founded order on S, then $>^+$ is also well-founded on S

3.6.2 Correctness - Behaviour

We denote that two functions `fac` and `fac2` compute the same function usually as

$$\forall n \in \mathbb{N}.\texttt{fac} \; n = \texttt{fac2} \; (n, 1)$$

The two functions are given as follows:

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

```haskell
fac2 :: (Int, Int) -> Int
fac2 (0, a) = a
fac2 (n, a) = fac2 (n - 1, n * a)
```

An important fact to consider is that testing, while useful to find errors can't replace a formal proof to show that a function is correct!

These proofs are based on a simple idea: ***functions are equations*** and thus, we can reason about them through equational reasoning, or more generally, proofs in first-order logic with equality.

Often, especially in Haskell programs, we have cases depending on values. Logically, to prove such a function, we also use case distinction, also referred to as reasoning by cases.

Example 3.6.7 Consider the Haskell function below

```haskell
maxi :: Int -> Int -> Int
maxi n m
    | n >= m    = n
    | otherwise = m
```

To prove that it is correct, we can use reasoning by cases:

We have $n \geq m \vee \neg(n \geq m)$.

We then show that the function is correct for both cases (i.e. LHS and RHS of OR):

C1 $n \geq m$: Then `maxi` $n \; m = n$ and $n \geq n$

C2 $\neg(n \geq m)$: Then `maxi` $n \; m = m$. But $m > n$, so we have `maxi` $n \; m \geq n$

In this proof we used the **TND** and $\vee$-**E** (here also called ***Case Split***) rules.

What we have to show, given $Q \vee R$ for any proposition P with case split, is that ***(1)*** P follows from Q and ***(2)*** P follows from R

4 Typing

A great type system is essential for all programming languages, but especially so for functional programming languages.

The issue however is that the problem of deciding which expressions are good and which ones aren't is undecidable.

Thus, languages only allow a subset of good expressions. The goal is to make the type system as unrestrictive as possible while still retaining quick, static code analysis.

4.1 Mini-Haskell

This is a stripped down version of Haskell, used here to explore the type system Haskell uses

4.1.1 Syntax

Programs are terms, the core is the lambda-calculus, where $\mathcal{V}$ is the set of variables and $\mathbb{Z}$ the set of integers:

$$t ::= \mathcal{V} \mid (\lambda x.t) \mid (t_1 t_2) \mid True \mid False \mid (\textbf{iszero } t) \mid \mathbb{Z} \mid (t_1 + t_2) \mid t_1 * t_2 \mid \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \mid (t_1, t_2) \mid (\textbf{fst } t) \mid (\textbf{snd } t)$$

It is easily possible to add additonal syntax and types and we employ syntactic sugar, such as omitting parenthesis.

The types are given by $\tau ::= \mathcal{V}_T \mid \texttt{Bool} \mid \texttt{Int} \mid (\tau, \tau) \mid (\tau \to \tau)$, where $\mathcal{V}_T$ is a set of type variables. The type system is based on typing judgement of form $\Gamma \vdash t :: r$, where Γ is a set of bindings $x_i : \tau_i$ that maps variables to types and can be understood as a typing symbol table.

4.1.2 Lambda calculus

To prove that types are correct, the lambda calculus comes in handy. The proofs are again structured as derivation trees.

4.1.2.1 Core rules for Lambda-Calculus

$$\frac{}{\Gamma, x : \tau \vdash x :: \tau}\ \text{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\forall x.t) :: \sigma \to \tau}\ \text{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau}\ \text{App}$$

For rule `Abs`, we require that $x \notin \Gamma$

4.1.3 Further rules for mini-Haskell

4.1.3.1 Base types

$$\frac{}{\Gamma \vdash n :: \texttt{Int}}\ \text{Int} \qquad \frac{}{\Gamma \vdash \texttt{True} :: \texttt{Bool}}\ \text{True} \qquad \frac{}{\Gamma \vdash \texttt{False} :: \texttt{Bool}}\ \text{False}$$

4.1.3.2 Operations

Let **op** $\in \{+, *\}$

$$\frac{\Gamma \vdash t :: \texttt{Int}}{\Gamma \vdash (\textbf{iszero } t) :: \texttt{Bool}}\ \text{iszero} \qquad \frac{\Gamma \vdash t_1 :: \texttt{Int} \qquad \Gamma \vdash t_2 :: \texttt{Int}}{\Gamma \vdash (t_1 \textbf{ op } t_2) :: \texttt{Int}}\ \text{BinOp}$$

$$\frac{\Gamma \vdash t_0 :: \texttt{Bool} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2) :: \tau}\ \text{if}$$

4.1.3.3 Tuples

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ \text{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\textbf{fst } t) :: \tau_1}\ \text{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\textbf{snd } t) :: \tau_2}\ \text{snd}$$

4.1.4 Type inference

Type inference in general fails, if two (or more) branches fail to resolve to unifiable types.

We start a ***type judgement*** with judgement $\vdash t :: \tau_0$, then build a derivation tree bottom-up. Finally, apply constraints / unification to get possible types.

4.1.4.1 Self application

This means that you apply a function to itself. In Haskell, this is not typeable because there would need to be an infinite function type, but all ***Haskell types are finite***

4.1.4.2 Curry-Howard isomorphism

We can also apply the implication introduction and implication elimination rules:

$$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}\ \to\text{-I} \qquad \frac{\Gamma \vdash \sigma \to \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\ \to\text{-E}$$

4.2 Natural Number Proofs

To prove $\forall n \in \mathbb{N}.P$, we of course again use induction:

Base Case Show $P[n \mapsto 0]$

Step Case Let $m \in \mathbb{N}$ be arbitrary and not free in P. We then assume that $P[n \mapsto m]$ and show that $P[n \mapsto m+1]$

Or the same as a natural deduction rule:

$$\frac{\Gamma \vdash P[n \mapsto 0] \qquad \Gamma, P[n \mapsto m] \vdash P[n \mapsto m+1]}{\Gamma \vdash \forall n \in \mathbb{N}.P} \; m \text{ not free in } \Gamma, P$$

4.2.1 Induction over the natural numbers

In Haskell, we can also define all the natural numbers using

```
data Nat = Zero | Succ Nat deriving (Eq, Ord, Show)
```

Thus the natural numbers are (isomorphic to) the set

$$\texttt{Nat} = \{\texttt{Zero}, \texttt{Succ Zero}, \texttt{Succ (Succ Zero)}, \ldots\}$$

The data type provides two crucial rules for constructing members of `Nat`:

- $\texttt{Zero} \in \texttt{Nat}$
- If $x \in \texttt{Nat}$, then $\texttt{Succ}x \in \texttt{Nat}$

The induction stated as a natural deduction rule:

$$\frac{\Gamma \vdash P[n \mapsto \texttt{Zero}] \qquad \Gamma, P[n \mapsto m] \vdash P[n \mapsto \texttt{Succ } m]}{\Gamma \vdash \forall n \in \texttt{Nat}.P} \; m \text{ not free in } \Gamma, P$$

4.2.2 Lists

A possible data type for lists in Haskell is:

```
data L t = Nil | Cons t (L t)
```

A natural deduction rule for induction over lists is:

$$\frac{\Gamma \vdash P[xs \mapsto \texttt{Nil}] \qquad \Gamma, P[xs \mapsto ys] \vdash P[xs \mapsto \texttt{Cons } y \; ys]}{\Gamma \vdash \forall xs \in \texttt{L } t.P} \; y, ys \text{ not free in } \Gamma, P$$

4.2.3 Trees

A possible data type for trees in Haskell is:

```
data Tree t = Leaf | Node t (Tree t) (Tree t)
```

A natural deduction rule for induction over trees is:

$$\frac{\Gamma \vdash P[x \mapsto \texttt{Leaf}] \qquad \Gamma, P[x \mapsto l] \vdash P[xs \mapsto \texttt{Node } a \; l \; r]}{\Gamma \vdash \forall x \in \texttt{Tree } t.P} \; a, l, r \text{ not free in } \Gamma, P$$

4.3 Interpreters

Interpreters are prevalent in programming languages, database systems, text processors, HDLs, search engines, etc.

They are in concept very simple, as they perform three steps read, evaluate, print.

The implementation of one however is not trivial by any stretch of the imagination.

4.3.1 Read step

During this step, text is turned from text into a more easily handlable format

4.3.1.1 Lexical Analysis

During lexical analysis, the input is turned into tokens. For example: The source code is `position := initial + rate + 60`

The translation is:

1. Identifier `position`
2. Assignment symbol `:=`
3. Identifier `initial`
4. Addition symbol +
5. Identifier `rate`
6. Addition symbol +
7. Number 60

It also removes whitespaces and comments

4.3.1.2 Parsing

The tokens are then turned into an abstract syntax tree.

The syntax is specified by a grammar such as:

$$Expr ::= Identifier \mid Number \mid Expr \text{ `+` } Expr$$
$$Assign ::= Identifier \text{ `:=` } Expr$$

This can also be represented as a haskell type:

```haskell
data Expr = Identifier Ident | Number Num | Plus Expr Expr
data Assign = Assignment Ident Expr
type Ident = String
type Num = Int
```

Since some to be parsed statements are more complex to parse, we may do combinatory parsing.

This is much more powerful as it can handle ambiguous grammars typically found in real programming languages.

We can use for example these parser combinators:

```haskell
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}

{-# HLINT ignore "Use lambda-case" #-}
{-# HLINT ignore "Use newtype instead of data" #-}

import Prelude hiding (return, (>>), (>>=))

data Parser a = Prs (String -> [(a, String)])

-- Main parser function
parse :: Parser a -> String -> [(a, String)]
parse (Prs p) = p

--------------------
-- Basic parsers --
--------------------
-- Trivial failure ([] signifies parse failed)
failure :: Parser a
failure = Prs (const [])

-- Trivial success without progress
return :: a -> Parser a
return x = Prs (\inp -> [(x, inp)])

-- Trivial success with progress
item :: Parser Char
item =
  Prs
    ( \inp -> case inp of
        "" -> []
        (x : xs) -> [(x, xs)]
    )

----------
-- Glue --
----------
-- Apply both parsers
(|||) :: Parser a -> Parser a -> Parser a
p ||| q = Prs (\s -> parse p s ++ parse q s)

-- If first parser fails, apply second parser
(+++) :: Parser a -> Parser a -> Parser a
p +++ q =
  Prs
    ( \s -> case parse p s of
        [] -> parse q s
        res -> res
    )

-- Sequencing (first parser p, then parser q)
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= g = Prs (\s -> [(u, s'') | (t, s') <- parse p s, (u, s'') <- parse (g t) s'])

-- Simple version of the above
```

```haskell
(>>) :: Parser a -> Parser b -> Parser b
p >> q = p >>= const q

-- Parse single character with property p
sat :: (Char -> Bool) -> Parser Char
sat p = item >>= \x -> if p x then return x else failure

char :: Char -> Parser Char
char x = sat (== x)

string :: String -> Parser String
string "" = return ""
string (x : xs) = char x >> string xs >> return (x : xs)

-- 0 or more repetitions of p
many :: Parser a -> Parser [a]
many p = many1 p ||| return []

-- 1 or more repetitions of p
many1 :: Parser a -> Parser [a]
many1 p = p >>= \t -> many p >>= \ts -> return (t : ts)
```

These are just some of the functions defined. You can find a full mini-haskell and lambda-calculus parser on CodeExpert (at the time of writing that was the case at least)

4.4 Evaluation

Evaluation is then done using tree traversal as we have already seen in the Haskell section

4.4.1 Lazy Evaluation

Expressions are substituted before evaluation recursively until there are no more expressions to substitute, at which point the expression is evaluated.

This can obviously lead to duplicated evaluation, i.e. a computation reoccurring.

4.4.1.1 In Haskell

In Haskell, this is solved using sharing where the terms are represented in a directed graph.

In pattern matching, the arguments are evaluated only as much as is needed to determine a pattern match.

For guards, the execution proceeds sequentially until success occurs. For instance in an `OR` statement, only the first statement is evaluated if it evaluates to true

Local definitions are also lazily evaluated (i.e. bound with `where` clauses)

4.4.1.2 Applications

This concept can be used for data-driven programming. For instance, to determine the minimum value of a list, we could use insertion sort and take the head. Due to lazy evaluation, we have way fewer evaluations that need to happen.

Additionally for infinite lists or other infinite data structures, lazy evaluation allows creating a finite representation for the infinite data. It also allows operating on the infinite data given the operation only operates on a finite subset of the data structure.

An application of that is the prime number algorithm Sieve of Eratosthenes:

1. Generate list: `[2 ..]` (list of all natural numbers)
2. Mark the first unmarked number: `head :: [a] -> a` from prelude determines first element
3. Cross out all multiples: `dropMults x ys = filter (\y -> y `mod` x /= 0) ys`
4. Repeat with recursions: `sieve xs = head xs : sieve (dropMults (head xs) (tail xs))`

Another example is Newton's Algorithm to find roots.

4.4.1.3 Correctness

Lazy evaluation makes reasoning about complexity and correctness harder, as types like `[Int]` include

1. finite, everywhere defined lists (e.g. `[1, 3, 5]`)
2. finite lists with undefined elements like `[1, 2, undef]`
3. infinite lists with defined or undefined elements such as `[1..]`

However, induction is only sound for the first kind. More on this later on

5 Language Semantics

5.1 IMP Language

5.1.1 The syntax

The allowed characters:

```
Letter = 'A' | . . . | 'Z' | 'a' | . . . | 'z'
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

And the allowed tokens:

```
Ident = Letter { Letter | Digit }*
Numeral = Digit | Numeral Digit
Var = Ident
```

Arithmetic and Boolean statements could be defined in Haskell as follows

```
data Aexp = Bin Op Aexp Aexp | Var String | Num Integer
data Op = Add | Sub | Mul
data Bexp = Or Bexp Bexp | And Bexp Bexp | Not Bexp | Rel Rop Aexp Aexp
data Rop = Eq | Neq | Le | Leq | Ge | Geq
```

with the abbreviations for `Eq` ($=$), `Neq` ($\#$), `Le` ($<$), `Leq` ($\leq$), `Ge` ($>$) and `Geq` ($\geq$)

Statements could be defined in Haskell as follows

```
data Stm = Skip | Assign String Aexp | Seq Stm Stm | If Bexp Stm Stm | While Bexp Stm
```

We use the following naming conventions for meta-variables:

Variable	Type
n	for numerals (`Numeral`)
x, y, z	for variables (`Var`)
e, e', e_1, e_2	for arithmetic expressions (`Aexp`)
b, b_1, b_2	for boolean expressions (`Bexp`)
s, s', s_1, s_2	for Statements (`Stm`)

Meta-variables stand for arbitrary program variables, whereas program variables are concrete variables in a program.

To denote ***syntactic equality*** of two variables or statements, we use $\equiv$

5.1.2 The semantics

5.1.2.1 Numerals

The semantic function $\mathcal{N} : \texttt{Numeral} \to \texttt{Val}$ maps a numeral n to an integer value $\mathcal{N}[\![n]\!]$, with $x \in \{0, \dots, 9\}$:

$$\mathcal{N}[\![x]\!] = x \qquad \mathcal{N}[\![nx]\!] = \mathcal{N}[\![n]\!] \cdot 10 + x$$

5.1.2.2 States

A state assigns a value to each program variable. It is a total function and is typically denoted by the meta-variable σ

$$\sigma : \texttt{Var} \to \texttt{Val}$$

To update states, we use the notation $\sigma[y \mapsto v]$, which is given by

$$(\sigma[y \mapsto v])(x) = \begin{cases} v & \text{if } x \equiv y \\ \sigma(x) & \text{if } x \not\equiv y \end{cases}$$

Two states σ_1, σ_2 are equal if they are equal as functions: $\sigma_1 = \sigma_2 \Leftrightarrow \forall x.(\sigma_1(x) = \sigma_2(x))$

5.1.2.3 Arithmetic Expressions

$\mathcal{A} : \texttt{Aexp} \to \texttt{State} \to \texttt{Val}$ maps an arithmetic expression e and a state σ to a value $\mathcal{A}[\![e]\!]\sigma$, given by:

$$\mathcal{A}[\![x]\!]\sigma = \sigma(x)$$
$$\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$$
$$\mathcal{A}[\![e_1 \texttt{ op } e_2]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma \ \overline{\texttt{op}} \ \mathcal{A}[\![e_2]\!]\sigma$$

For $\texttt{op} \in \texttt{Op}$, $\overline{\texttt{op}}$ is the corresponding operation $\texttt{Val} \times \texttt{Val} \to \texttt{Val}$

5.1.2.4 Boolean Expressions

$\mathcal{B} : \texttt{Bexp} \to \texttt{State} \to \texttt{Bool}$ maps boolean expression b and a state σ to a truth value $\mathcal{B}[\![b]\!]\sigma$, given by:

$$\mathcal{B}[\![e_1 \texttt{ op } e_2]\!]\sigma = \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![e_1]\!]\sigma \ \overline{\texttt{op}} \ \mathcal{A}[\![e_2]\!]\sigma \\ \texttt{ff} & \text{otherwise} \end{cases}$$

Thus, for the op `or`, we would have (analogous for `and`)

$$\mathcal{B}[\![b_1 \texttt{ or } b_2]\!]\sigma = \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![b_1]\!]\sigma = \texttt{tt or } \mathcal{A}[\![b_2]\!]\sigma = \texttt{tt} \\ \texttt{ff} & \text{otherwise} \end{cases}$$

`not` is defined as follows:

$$\mathcal{B}[\![\texttt{not } b]\!]\sigma = \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![b]\!]\sigma = \texttt{ff} \\ \texttt{ff} & \text{otherwise} \end{cases}$$

5.1.3 Properties of expression semantics

Since we have recursive definitions for the semantics and syntax, we can use structural induction.

Structural Induction **Recall**

For the data structure `Nat`, given by

```
data Nat = Zero | Succ Nat
```

the structural induction derivation rule is given by

$$\frac{\Gamma \vdash P(\texttt{Zero}) \qquad \Gamma, P(m) \vdash P(\texttt{Succ } m)}{\Gamma \vdash \forall n \in \texttt{Nat}.P(n)} \ m \text{ not free in } \Gamma$$

Where we now write $P(m)$ instead of $P[n \mapsto m]$ and the second premise needs to be proven for all m

5.1.3.1 Inductive Definitions

If we are to introduce a new arithmetic expression $-e$, we could do this in two ways. For one, we could define $\mathcal{A}[\![-e]\!]\sigma = 0 - \mathcal{A}[\![e]\!]\sigma$. This *is* an inductive definition because e is a subterm of $-e$.

If on the other hand we define $\mathcal{A}[\![-e]\!]\sigma = \mathcal{A}[\![0 - e]\!]\sigma$, it is *not* an inductive definition because $0 - e$ is *not* a subterm of $-e$

5.1.3.2 Free Variables

For Arithmetic Expressions

$$\begin{aligned} FV(e_1 \texttt{ op } e_2) &= FV(e_1) \cup FV(e_2) \\ FV(n) &= \varnothing \\ FV(x) &= \{x\} \end{aligned}$$

For Boolean Expressions

$$\begin{aligned} FV(b_1 \texttt{ op } b_2) &= FV(b_1) \cup FV(b_2) \\ FV(\texttt{not } b) &= FV(b) \\ FV(b_1 \texttt{ or } b_2) &= FV(b_1) \cup FV(b_2) \\ FV(b_1 \texttt{ and } b_2) &= FV(b_1) \cup FV(b_2) \end{aligned}$$

And finally for Statements:

$$\begin{aligned} FV(\texttt{skip}) &= \varnothing \\ FV(x := e) &= \{x\} \cup FV(s_2) \\ FV(s_1; s_2) &= FV(s_1) \cup FV(s_2) \\ FV(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}) &= FV(b) \cup FV(s_1) \cup FV(s_2) \\ FV(\texttt{while } b \texttt{ do } s \texttt{ end}) &= FV(b) \cup FV(s) \end{aligned}$$

5.1.3.3 Substitution

We have already seen this kind of expression in the states, with this explanation it should make a lot more sense intuitively.

A substitution $f[x \mapsto e]$ replaces each free occurrence of variable x in f by e, where f is any expression.

Detailed rules for arithmetic expressions (for the last, if variable y is x, it is replaced, otherwise not):

$$\begin{aligned} (e_1 \texttt{ op } e_2)[x \mapsto e] &\equiv (e_1[x \mapsto e] \texttt{ op } e_2[x \mapsto e]) \\ n[x \mapsto e] &\equiv n \\ y[x \mapsto e] &\equiv \begin{cases} e & \text{if } x \equiv y \\ y & \text{otherwise} \end{cases} \end{aligned}$$

The same for boolean expressions:

$$\begin{aligned} (e_1 \texttt{ op } e_2)[x \mapsto e] &\equiv (e_1[x \mapsto e] \texttt{ op } e_2[x \mapsto e]) \\ (\texttt{not } b)[x \mapsto e] &\equiv \texttt{not } (b[x \mapsto e]) \\ (b_1 \texttt{ or } b_2)[x \mapsto e] &\equiv (b_1[x \mapsto e] \texttt{ or } b_2[x \mapsto e]) \\ (b_1 \texttt{ and } b_2)[x \mapsto e] &\equiv (b_1[x \mapsto e] \texttt{ and } b_2[x \mapsto e]) \end{aligned}$$

Substitution Lemma — **Lemma 5.1.1**

$$\mathcal{B}[\![b[x \mapsto e]]\!] \Leftrightarrow \mathcal{B}[\![b]\!](\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])$$

5.2 Operational Semantics

Big-step semantics describe how the ***overall*** results of the execution are obtained and use Natural semantics rules. **Small-step semantics** describe how the individual steps of the computations take place and use Structural Operational Semantics (SOS)

5.2.1 Big-Step Semantics

5.2.1.1 Transition Systems

Definition 5.2.1 *(Transition System)* is a tuple $(\Gamma, T, \to)$, where Γ is a set of ***configurations***, T is a set of terminal configurations, with $T \subseteq \Gamma$ and $\to$ is a transition relation, with $\to \subseteq \Gamma \times \Gamma$, which describes how executions take place. Big-step transitions are of form $\langle s, \sigma\rangle \to \sigma'$, e.g. $\langle \texttt{skip}, \sigma\rangle \to \sigma$

The transition relations are specified as rules of the form (* optional side-condition, φ_i and ψ transitions)

$$\frac{\varphi_1 \qquad \cdots \qquad \varphi_n}{\psi}\ (\text{Name})^*$$

or spelled out, "If $\varpi_1, \ldots, \varphi_n$ are transitions (and the *side-condition* is true), then ψ is a transition".

Herein, $\varphi_1, \ldots, \varphi_n$ are called ***premises*** of the rule and ψ is the ***conclusion***. A rule without premises is an ***axiom rule***.

5.2.1.2 Big-Step Semantics of IMP

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}\ \textsc{Skip}_{NS} \qquad \frac{}{\langle x := e, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{NS}$$

Sequential Composition $s; s'$ (s is executed in state σ, then s' in resulting σ', resulting in σ'')

$$\frac{\langle s, \sigma\rangle \to \sigma' \qquad \langle s', \sigma'\rangle \to \sigma''}{\langle s; s', \sigma\rangle \to \sigma''}\ \textsc{Seq}_{NS}$$

Conditional Statements $\texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}$ (If b holds, execute s, otherwise execute s')

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to \sigma'}\ \textsc{IfT}_{NS} \qquad \frac{\langle s, \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to \sigma'}\ \textsc{IfF}_{NS}$$

Where the first rule applies if $\mathcal{B}[\![b]\!]\sigma = \texttt{tt}$

Loop statements $\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}$ (If b holds, execute s once, whole statement executed in resulting state σ)

$$\frac{\langle s, \sigma\rangle \to \sigma' \qquad \langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma'\rangle \to \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \to \sigma''}\ \textsc{WhT}_{NS} \qquad \text{if } \mathcal{B}[\![b]\!]\sigma = \texttt{tt}$$

If b does not hold, the while statement does *not* modify the state

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \to \sigma}\ \textsc{WhF}_{NS} \qquad \text{if } \mathcal{B}[\![b]\!]\sigma = \texttt{ff}$$

5.2.1.3 Rule Schemes and Instantiations

Each inference rule is actually a rule scheme, where the meta-variables are placeholders for statements, states, etc. Each rule scheme describes infinitely many ***rule instances***.

A rule is ***instantiated*** when all meta-variables are replaced with syntactic elements. Assignment rule scheme vs. instance:

$$\frac{}{\langle x := e, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{NS} \qquad \frac{}{\langle v := v + 1, \sigma_{\text{zero}}\rangle \to \sigma[v \mapsto \mathcal{A}[\![v + 1]\!]\sigma_{\text{zero}}]}\ \textsc{Ass}_{NS}$$

The ***rule instances*** can be combined to derive a transition $\langle s, \sigma\rangle \to \sigma'$ and we get a derivation tree.

5.2.1.4 Termination

Termination **Definition 5.2.2**

The execution of a statement s in a state σ

- ***terminates successfully*** if and only if there exists a state σ' such that $\vdash \langle s, \sigma \rangle \to \sigma'$
- ***fails to terminate*** if and only if there is no state σ' such that $\vdash \langle s, \sigma \rangle \to \sigma'$

5.2.1.5 Semantic equivalence

Definition 5.2.3 Two statements s_1 and s_2 are ***semantically equivalent***, denoted $s_1 \simeq s_2$, if and only if

$$\forall \sigma, \sigma'. (\vdash \langle s_1, \sigma \rangle \to \sigma' \iff \vdash \langle s_2, \sigma \rangle \to \sigma')$$

5.2.1.6 Unfolding loops

In languages like `C` (and by extension `C++`) and `Java`, unfolding a loop leads to non-equivalent code:

```c
int i = 0;
while ( i < 2 ) {



    while ( i < 1 )
        if ( i == 0 ) break;

    i++;
}
printf( "i = %d", i );
```

Prints `i = 2`

```c
int i = 0;
while ( i < 2 ) {
    if ( i == 0 ) {
        if ( i == 0 ) break;
        while ( i < 1 )
            if ( i == 0 ) break;
    }
    i++;
}
printf( "i = %d", i );
```

Prints `i = 0`

In IMP however, this kind of action leads to equivalence:

$$\forall b, s. (\texttt{while b do s end} \simeq \texttt{if b then s; while b do s end end})$$

So what we have to prove is this:

$$\forall b, s, \sigma, \sigma'. (\vdash \langle \texttt{while b do s end}, \sigma \rangle \Leftrightarrow \vdash \langle \texttt{if b then s; while b do s end end}, \sigma \rangle \to \sigma')$$

A proof idea is to show equivalence by proving the implication in both directions. For each of them, we show that there is a derivation tree.

Proof available in the lecture slides for Formal Methods, Slides 78 - 80 (Slide Deck 3, pages 23 - 25)

5.2.1.7 Deterministic Semantics

Lemma 5.2.4 The big-step semantics of IMP is deterministic

Proof We need to show $\forall s, \sigma, \sigma', \sigma''. (\vdash \langle s, \delta \rangle \to \sigma' \land \vdash \langle s, \sigma \rangle \to \sigma'' \implies \sigma' = \sigma'')$

The full proof for this is available on the slides for Formal Methods, pages 82 - 91 (Slide Deck 3, pages 27 - 40)

In there, ***Induction on Derivation Trees*** is used. Similar like other induction proofs, we show that a property $P(T)$ holds for all derivation trees T, we prove that $P(T)$ holds for an arbitrary derivation tree T under the assumption (the induction hypothesis) that $P(T')$ holds for all sub-trees T' of T

This kind of induction is a special case of ***well-founded (Noetherian) induction***. We define $T' \sqsubset T$ (with T, T' derivation trees) to mean that T' is a proper sub-tree of T. $\sqsubset$ is a well-founded ordering, because derivation trees are finite and we call T' a ***sub-derivation*** of T if $T' \sqsubset T$. Typically, *case distinction* is used on the rule applied at the root of T because that provides more information about the structure of the derivation, such as telling us about sub-derivation to which the induction hypothesis applies.

5.2.1.8 Extensions of IMP

5.2.1.8.1 Local Variable Declarations

A statement `var x := e in s end` declares a local variable that is visible in the sub-statement of the declaration, `s`.

Here, the Expression e is evaluated in the initial state, then s is executed in a state in which x has the value of e and after the execution, the original value of x is restored.

The corresponding Big-step semantics rule is:

$$\frac{\langle s, \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]\rangle \to \sigma'}{\langle \texttt{var x := e in s end}, \sigma\rangle \to \sigma'[x \mapsto \sigma(x)]}\ \text{Loc}_{NS}$$

5.2.1.8.2 Procedure Declaration and Calls

`procedure p(`$x_1, \dots, x_n; y_1, \dots, y_m$`) begin s end`

Here, the x_i are the ***value*** parameters and the y_j are the ***variable*** parameters. The latter can be used to assign values back to the procedure caller (return values).

In a ***procedure declaration*** the ***formal*** parameter names x_i and y_j must be pairwise distinct and the only free variables in s.

For the ***procedure call*** $p(e_1, \dots, e_n; z_1, \dots, z_m)$, the ***actual*** variable parameters must be pairwise distinct.

The corresponding Big-step semantics rule is:

$$\frac{\langle s, \sigma_{\text{zero}}[\overrightarrow{x_i} \mapsto \overrightarrow{\mathcal{A}[\![e_i]\!]\sigma}][\overrightarrow{y_j} \mapsto \overrightarrow{\sigma(z_j)}]\rangle \to \sigma'}{\langle p(\overrightarrow{e_i}; \overrightarrow{z_j}), \sigma\rangle \to \sigma[\overrightarrow{z_j} \mapsto \overrightarrow{\sigma'(y_j)}]}\ \text{Call}_{NS}$$

where the notation $\sigma[\overrightarrow{x_i} \mapsto \overrightarrow{v_i}]$ is an abbreviation of $\sigma[x_1 \mapsto v_1] \dots [x_n \mapsto v_n]$

5.2.1.8.3 Non-determinism

For a statement $s \square s'$, either s or s' is non-deterministically chosen to be executed.

Example 5.2.5 In the statement `x := 1` $\square$ `(x := 2; x := x + 2)`, we either get $x = 1$ or $x = 4$ (either the statement on the left or right of the box is evaluated)

The corresponding rules are:

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle s \square s', \sigma\rangle \to \sigma'}\ \text{ND1}_{NS} \qquad \frac{\langle s', \sigma\rangle \to \sigma'}{\langle s \square s', \sigma\rangle \to \sigma'}\ \text{ND2}_{NS}$$

An important note on non-terminating branches, we will not "see" them, because big-step semantics can't encompass that concept.

5.2.1.8.4 Parallelism

In a statement `s par s'`, both s and s' are executed, but their execution can be ***interleaved***.

Example 5.2.6 `x := 1 par (x := 2; x := x + 2)` could result in:

- 4: first `x:=1`, then `x:=2` and finally `x:=x+2`
- 3: first `x:=2`, then `x:=1` and finally `x:=x+2`
- 1: first `x:=2`, then `x:=x+2` and finally `x:=1`

In Big-step semantics however, there are no rules for this, because it can only define atomic steps.

5.2.2 Small-Step semantics

5.2.2.1 Structural Operational Semantics (SOS)

Expressing each individual steps of execution allows one to express the ***order of execution*** of these steps, thus allowing us to describe properties of non-terminating programs and parallelization.

γ is used as the meta-variable for terminal and non-terminal configurations. For the transitions, we denote the relation $\to_1$, which can have two forms:

- $\langle s, \sigma\rangle \to_1 \langle s', \sigma'\rangle$ is for any non-complete computation, where the next computation is expressed by the intermediate configuration $\langle s', \sigma'\rangle$
- $\langle s, \sigma\rangle \to_1 \sigma'$ is the final execution that reaches a terminal state.

Finally, a transition $\langle s, \sigma\rangle \to_1 \gamma$ describes the ***first step*** of the execution of s in state σ.

A non-terminal configuration $\langle s, \sigma\rangle$ is ***stuck***, if there does not exist a configuration γ such that $\langle s, \sigma\rangle \to_1 \gamma$.

5.2.2.1.1 The rules

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to_1 \sigma}\ \textsc{Skip}_{SOS} \qquad \frac{}{\langle x := e, \sigma\rangle \to_1 \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{SOS}$$

Sequential Composition $s; s'$ (s is executed in state σ, then s' in resulting σ', resulting in σ''). Then, either s executes completely in one step ($\textsc{Seq}1$), or does not ($\textsc{Seq}2$).

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s; s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{Seq}1_{SOS} \qquad \frac{\langle s', \sigma'\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s; s', \sigma\rangle \to_1 \langle s''; s', \sigma'\rangle}\ \textsc{Seq}2_{SOS}$$

Conditional Statements $\texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}$ (If b holds, execute s, otherwise execute s')

$$\frac{}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{IfT}_{SOS} \qquad \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{IfF}_{SOS}$$

Where the first rule applies if $\mathcal{B}[\![b]\!]\sigma = \texttt{tt}$. Below a further two rules for the true case:

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to_1 \sigma'}\ \textsc{IfT}1_{SOS} \qquad \frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma\rangle \to_1 \langle s''', \sigma'\rangle}\ \textsc{IfT}2_{SOS}$$

Loop statements $\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}$ (If b holds, execute s once, whole statement executed in resulting state σ)

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \to_1 \langle \texttt{if}\ b\ \texttt{then}\ s;\ \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}, \sigma\rangle}\ \textsc{While}_{SOS}$$

5.2.2.2 Multi-Step Execution

We use the definitions we have to define a k-***step execution***, denoted $\gamma \to_1^k \gamma'$. Of course, this means intuitively that there is an execution of exactly k steps from γ to γ'.

We define the relation inductively over k:

- $\gamma \to_1^0 \gamma'$ if and only if $\gamma = \gamma'$
- $\gamma \to_1^k \gamma'$ if and only if there exists γ'' such that both $\vdash \gamma \to_1$ and $\gamma'' \to_1^{k-1} \gamma'$

Resulting from this, $\gamma \to_1^{k_1+k_2} \gamma'$ if and only if $\exists \gamma''.\gamma \to_1^{k_1} \gamma'' \land \gamma'' \to_1^{k_2} \gamma'$

We write $\gamma \to_1^* \gamma'$ to signify that there is an execution from γ to γ' in some finite number of steps, or more formally:

$$\exists k.\gamma \to_1^k \gamma'$$

5.2.2.3 Derivation Sequences

Definition 5.2.7 A ***derivation sequence*** is a non-empty sequence of configurations $\gamma_0, \ldots$, for which $\gamma_i \to_1^1 \gamma_{i+1}$ for each $0 \leq i$, such that $i+1$ is in the range of the sequence. If the sequence is finite, then the last configuration in the sequence is either a terminal or stuck configuration.

The ***length*** of the derivation sequence is the number of transitions (thus number of states minus one!)

5.2.2.4 Termination

Theorem 5.2.8 The execution of a statement s in a state σ

- ***terminates*** if and only if there is a finite derivation sequence starting with $\langle s, \sigma\rangle$
- ***runs forever*** if and only if there is an infinite derivation sequence starting with $\langle s, \sigma\rangle$

Theorem 5.2.9 The execution of statement s in state σ ***terminates*** successfully, if and only if $\exists \sigma'. \langle s, \sigma\rangle \to_1^* \sigma'$

Of note is that these are properties of ***configurations*** and not just statements.

5.2.2.5 Proving properties of Derivation Sequences

For reasoning about finite derivation sequences, we commonly reason about a multi-step execution $\gamma \to_1^k \gamma'$ by ***strong induction on the number of steps*** k, where we define $P(k) \equiv$ "for all executions of length k, our property holds" and we prove $P(k)$ for arbitrary k with the ***induction hypothesis*** $\forall k' < k. P(k')$ holds

After the setup, it *often* proceeds by case distinction on the 0 step and the other steps

5.2.2.6 Semantic Equivalence and Determinism

Definition 5.2.10 Under the small-step semantics, two statements s_1 and s_2 are ***semantically equivalent*** if for all states σ both:

- for all stuck or terminal configurations γ we have $\langle s_1, \sigma\rangle \to_1^* \gamma$ if and only if $\langle s_2, \sigma\rangle \to_1^* \gamma$, and
- there is and infinite derivation sequence starting in $\langle s_1, \sigma\rangle$ if and only if there is one starting in $\langle s_2, \sigma\rangle$

Lemma 5.2.11 The small-step semantics of IMP is ***deterministic***. That is:

$$\forall s, \sigma, \gamma, \gamma'. \vdash \langle s, \sigma\rangle \to_1 \gamma \wedge \vdash \langle s, \gamma\rangle \to_1 \gamma' \implies \gamma = \gamma'$$

Corollary 5.2.12 There is exactly one derivation sequence starting in configuration $\langle s, \sigma\rangle$

5.2.2.7 Extensions of IMP

5.2.2.7.1 Local Variable Declarations

As already partially established in Section 5.2.1.8.1, the steps to define a local variable are (for `var x:=e in s end`):

1. Assign e to x
2. Execute s (possibly many steps)
3. Restore the initial value of x

Since we need to somehow inject he restore instruction into the statements s, we extend the `Stm` category with a `restore` statement, defined as `restore (Var, Val)`.

With that, we can define the rules:

$$\frac{}{\langle \texttt{var x:= e in s end}, \sigma\rangle \to_1 \langle \texttt{s;restore } (x, \sigma(x)), \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]\rangle}\ \textsc{Loc}_{SOS}$$

$$\frac{}{\langle \texttt{restore } (x, \sigma(x)), \sigma\rangle \to_1 \sigma[x \mapsto v]}\ \textsc{Ret}_{SOS}$$

Of course, we could also just model execution stacks, as most languages do.

5.2.2.7.2 Non-determinism

The rules here are analogous to the ones from the big-step rules

$$\frac{}{\langle s \square s', \sigma\rangle \to_1 \langle s, \sigma\rangle}\ \text{ND1}_{SOS} \qquad \frac{}{\langle s \square s', \sigma\rangle \to_1 \langle s, \sigma'\rangle}\ \text{ND2}_{SOS}$$

5.2.2.7.3 Parallelism

$$\frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \ \texttt{par}\ s', \sigma\rangle \to_1 \langle s'' \ \texttt{par}\ s', \sigma'\rangle}\ \textsc{Par1}_{SOS} \qquad \frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s \ \texttt{par}\ s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{Par2}_{SOS}$$

$$\frac{\langle s', \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \ \texttt{par}\ s', \sigma\rangle \to_1 \langle s \ \texttt{par}\ s'', \sigma'\rangle}\ \textsc{Par3}_{SOS} \qquad \frac{\langle s', \sigma\rangle \to_1 \sigma'}{\langle s \ \texttt{par}\ s', \sigma\rangle \to_1 \langle s, \sigma'\rangle}\ \textsc{Par4}_{SOS}$$

5.2.3 Equivalence

Equivalence Theorem **Theorem 5.2.13**

For every statement s of IMP, we have

$$\vdash \langle s, \sigma \rangle \to \sigma' \Leftrightarrow \langle s, \sigma \rangle \to_1^* \sigma'$$

If the execution of s from some state terminates successfully in one of the semantics, than so will it in the other. The execution fails to terminate in the big-step semantics if and only if it either gets stuck, or runs forever in the small-step semantics

Lemma 5.2.14 *(Equivalence Lemma 1)* For every statement s of IMP and states σ and σ' we have:

$$\vdash \langle s, \sigma \rangle \to \sigma' \implies \langle s, \sigma \rangle \to_1^* \sigma'$$

If the execution of s from σ terminates successfully in the big-step semantics, so will it in the small-step semantics (in the same state, too!)

Lemma 5.2.15 *(Equivalence Lemma 2)* For every statement s of IMP and states σ and σ' and $k \in \mathbb{N}$, we have:

$$\langle s, \sigma \rangle \to_1^k \sigma' \implies \vdash \langle s, \sigma \rangle \to \sigma'$$

If the execution of s from σ terminates successfully in the small-step semantics, so will it in the big-step semantics (in the same state, too!)

5.2.4 Operational Semantics Rules Overview

5.2.4.1 Big-Step Semantics

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}\ \textsc{Skip}_{NS} \qquad \frac{}{\langle x := e, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{NS}$$

$$\frac{\langle s, \sigma\rangle \to \sigma' \qquad \langle s', \sigma'\rangle \to \sigma''}{\langle s; s', \sigma\rangle \to \sigma''}\ \textsc{Seq}_{NS}$$

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to \sigma'}\ \textsc{IfT}_{NS} \qquad \frac{\langle s, \sigma\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to \sigma'}\ \textsc{IfF}_{NS}$$

$$\frac{\langle s, \sigma\rangle \to \sigma' \qquad \langle \texttt{while } b \texttt{ do } s \texttt{ end}, \sigma'\rangle \to \sigma''}{\langle \texttt{while } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma''}\ \textsc{WhT}_{NS} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = \texttt{tt} \qquad \frac{}{\langle \texttt{while } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma}\ \textsc{WhF}_{NS} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = \texttt{ff}$$

$$\frac{}{\langle x := e, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{NS} \qquad \frac{}{\langle v := v + 1, \sigma_{\mathsf{zero}}\rangle \to \sigma[v \mapsto \mathcal{A}[\![v + 1]\!]\sigma_{\mathsf{zero}}]}\ \textsc{Ass}_{NS}$$

5.2.4.2 Small-Step Semantics

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to_1 \sigma}\ \textsc{Skip}_{SOS} \qquad \frac{}{\langle x := e, \sigma\rangle \to_1 \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}\ \textsc{Ass}_{SOS}$$

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s; s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{Seq1}_{SOS} \qquad \frac{\langle s', \sigma'\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s; s', \sigma\rangle \to_1 \langle s''; s', \sigma'\rangle}\ \textsc{Seq2}_{SOS}$$

$$\frac{}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{IfT}_{SOS} \qquad \frac{}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{IfF}_{SOS}$$

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \sigma'}\ \textsc{IfT1}_{SOS} \qquad \frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \langle s''', \sigma'\rangle}\ \textsc{IfT2}_{SOS}$$

$$\frac{}{\langle \texttt{while } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to_1 \langle \texttt{if } b \texttt{ then } s; \texttt{ while } b \texttt{ do } s \texttt{ end else skip end}, \sigma\rangle}\ \textsc{While}_{SOS}$$

5.2.4.3 Extensions

5.2.4.3.1 Big-Step Semantics

$$\frac{\langle s, \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]\rangle \to \sigma'}{\langle \texttt{var x := e in s end}, \sigma\rangle \to \sigma'[x \mapsto \sigma(x)]}\ \textsc{Loc}_{NS} \qquad \frac{\langle s, \sigma_{\mathsf{zero}}[\overrightarrow{x_i} \mapsto \overrightarrow{\mathcal{A}[\![e_i]\!]\sigma}][\overrightarrow{y_j} \mapsto \overrightarrow{\sigma(z_j)}]\rangle \to \sigma'}{\langle p(\overrightarrow{e_i}; \overrightarrow{z_j}), \sigma\rangle \to \sigma[\overrightarrow{z_j} \mapsto \overrightarrow{\sigma'(y_j)}]}\ \textsc{Call}_{NS}$$

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle s \square s', \sigma\rangle \to \sigma'}\ \textsc{ND1}_{NS} \qquad \frac{\langle s', \sigma\rangle \to \sigma'}{\langle s \square s', \sigma\rangle \to \sigma'}\ \textsc{ND2}_{NS}$$

5.2.4.3.2 Small-Step Semantics

$$\frac{}{\langle \texttt{var x:= e in s end}, \sigma\rangle \to_1 \langle \texttt{s;restore } (x, \sigma(x)), \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]\rangle}\ \textsc{Loc}_{SOS}$$

$$\frac{}{\langle \texttt{restore } (x, \sigma(x)), \sigma\rangle \to_1 \sigma[x \mapsto v]}\ \textsc{Ret}_{SOS}$$

$$\frac{}{\langle s \square s', \sigma\rangle \to_1 \langle s, \sigma\rangle}\ \textsc{ND1}_{SOS} \qquad \frac{}{\langle s \square s', \sigma\rangle \to_1 \langle s, \sigma'\rangle}\ \textsc{ND2}_{SOS}$$

$$\frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \texttt{ par } s', \sigma\rangle \to_1 \langle s'' \texttt{ par } s', \sigma'\rangle}\ \textsc{Par1}_{SOS} \qquad \frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s \texttt{ par } s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}\ \textsc{Par2}_{SOS}$$

$$\frac{\langle s', \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \texttt{ par } s', \sigma\rangle \to_1 \langle s \texttt{ par } s'', \sigma'\rangle}\ \textsc{Par3}_{SOS} \qquad \frac{\langle s', \sigma\rangle \to_1 \sigma'}{\langle s \texttt{ par } s', \sigma\rangle \to_1 \langle s, \sigma'\rangle}\ \textsc{Par4}_{SOS}$$

5.3 Axiomatic Semantics

5.3.1 Program Correctness

Definition 5.3.1 *(Partial Correctness) if* a program terminates *then* there will be a certain relationship between the initial and final state

Definition 5.3.2 *(Total Correctness)* Program terminates *and* is partially correct, i.e.
`total correctness = partial correctness + termination`

Definition 5.3.3 *(Formal Specification)* is used to expressed the aforementioned relationship between the initial and final state. It does not include guarantees for termination and can thus only be used to provide a starting point for a prove of partial correctness.

We typically express ***Formal Specifications*** using Small- or Big-Step semantics.

Example 5.3.4 Consider the factorial statement

```
y := 1;
while not x = 1 do
    y := y * x;
    x := x - 1
end
```

The specification that we want to prove is here given by "The final value of y is the factorial of the initial value x". Do note that this statement is only partially correct, as it does not terminate for $x < 1$.

We can express the specification formally as follows using big-step semantics:

$$\forall \sigma, \sigma'. \vdash \langle y := 1; \texttt{while not x = 1 do y := y * x; x := x - 1 end}, \sigma \rangle \to \sigma' \implies \sigma'(\texttt{y}) = \sigma(\texttt{x})! \land \sigma(\texttt{x}) > 0$$

Since these kinds of proofs take a lot of effort and get very complicated rather quickly, axiomatic semantics provide a nice and fast way of proving correctness, because we can focus on the essential properties.

5.3.2 Hoare Logic

5.3.2.1 Hoare Triples

Definition 5.3.5 Properties are specified as ***Hoare Triples*** $\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$, with statement s, $\boldsymbol{P}$ the precondition and $\boldsymbol{Q}$ the postcondition. Here, $\boldsymbol{P}, \boldsymbol{Q}$ are assertions and are, together with $\boldsymbol{R}$, meta-variables over assertions.

Definition 5.3.6 *(Logical Variables)* To keep the original value of a variable, we can "reassign" values in the precondition (e.g. $x = N$ to then in the postcondition state something regarding N).

Pre- and Postconditions are assertions, i.e. they are boolean expressions plus logical variables. Often, quantification is used in assertions as well in practice, as well as other expressions such as $x!$ when it is convenient and we also assume that the ***substitution lemma*** still holds:

$$\mathcal{B}[\![\boldsymbol{P}[x \mapsto e]]\!]\sigma = \mathcal{B}[\![\boldsymbol{P}]\!]\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

We use $P_1 \wedge P_2$ instead of P_1 `and` P_2, $P_1 \vee P_2$ instead of P_1 `or` P_2 and $\neg P$ instead of `not` P

5.3.2.2 Derivation Systems

We again use derivation trees, where their rules specify which triples can be derived for each statement. The premises and conclusions of the derivation rules are Hoare Triples.

Again, we write $\vdash \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$ if there exists a (finite) derivation tree ending in $\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$, and

$$\vdash \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \Leftrightarrow \exists T.\texttt{root}(T) \equiv \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$$

5.3.2.2.1 The rules

$$\frac{}{\{\boldsymbol{P}\}\ \texttt{skip}\ \{\boldsymbol{P}\}}\ \textsc{Skip}_{Ax} \qquad \frac{}{\{\boldsymbol{P}[x \mapsto e]\}\ \texttt{x := e}\ \{\boldsymbol{P}\}}\ \textsc{Ass}_{Ax}$$

Sequential Composition `s;s'`, loop (`while b do s end`) and conditional statements (`if b then s1 else s2 end`)

$$\frac{\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \qquad \{\boldsymbol{Q}\}\ s'\ \{\boldsymbol{R}\}}{\{\boldsymbol{P}\}\ \texttt{s;s'}\ \{\boldsymbol{P}\}}\ \textsc{Seq}_{Ax} \qquad \frac{\{b \wedge \boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \qquad \{\neg b \wedge \boldsymbol{P}\}\ s'\ \{\boldsymbol{Q}\}}{\{\boldsymbol{P}\}\ \texttt{if b then s else s'}\ \texttt{end}\ \{\boldsymbol{Q}\}}\ \textsc{If}_{Ax}$$

$$\frac{\{b \wedge \boldsymbol{P}\}\ s\ \{\boldsymbol{P}\}}{\{\boldsymbol{P}\}\ \texttt{while b do s end}\ \{\neg b \wedge \boldsymbol{P}\}}\ \textsc{Wh}_{Ax}$$

With these rules, we can only evaluate *syntactically*, thus expressions like $\{x = 4 \wedge y = 5\}$ `skip` $\{y = 5 \wedge x = 4\}$ are not an instance of the $\textsc{Skip}_{Ax}$ because the precondition and postcondition are not identical. Thus we often need to apply *semantic* reasoning, e.g. applying mathematical properties.

Definition 5.3.7 *(Semantic entailment)* expresses the reasoning steps "We write $\boldsymbol{P} \models \boldsymbol{Q}$ if and only if for all states σ, $\mathcal{B}[\![\boldsymbol{P}]\!]\sigma = \texttt{tt}$ implies $\mathcal{B}[\![\boldsymbol{Q}]\!]\sigma = \texttt{tt}$"

This leads to the ***Rule of Consequence***

$$\frac{\{\boldsymbol{P}'\}\ s\ \{\boldsymbol{Q}'\}}{\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}}\ \textsc{Cons}_{Ax} \qquad \text{if } \boldsymbol{P} \models \boldsymbol{P}' \text{ and } \boldsymbol{Q}' \models \boldsymbol{Q}$$

a rule, where we can ***strengthen preconditions*** ($\boldsymbol{P}$ cannot be weaker than $\boldsymbol{P}'$) and ***weaken postconditions*** ($\boldsymbol{Q}$ cannot be stronger than $\boldsymbol{Q}'$)

Since the derivation trees often get quite large, we can group them around each line in the program text.

Inline Notation can be used to make the changes more easily legible. For example, to express instances of $\textsc{Skip}_{Ax}$, instead of writing $\vdash \{\boldsymbol{P}\}$ `skip` $\{\boldsymbol{P}\}$, we can write. More examples on slides 167 - 170 (pages 30 - 33 in Slide Deck 4)

$$\begin{array}{l}\{\boldsymbol{P}\}\\ \quad\texttt{skip}\\ \{\boldsymbol{P}\}\end{array}$$

Forward Assignment

$$\frac{}{\{\boldsymbol{P}\}\ x := e\ \{\exists V.\boldsymbol{P}[x \mapsto V] \wedge x = e[x \mapsto V]\}}\ \textsc{AssF}_{Ax}$$

5.3.2.3 Total Correctness

We use a different form of Hoare triple $\{\boldsymbol{P}\}\ s\ \{\Downarrow \boldsymbol{Q}\}$, which describes total correctness.

All rules for total correctness are equivalent to the ones for partial correctness, apart from the rule for loops.

$$\frac{\{b \land \boldsymbol{P} \land e = Z\}\ s\ \{\Downarrow \boldsymbol{P} \land e < Z\}}{\{\boldsymbol{P}\}\ \texttt{while b do s end}\ \{\Downarrow \neg b \land \boldsymbol{P}\}}\ \textsc{WhTot}_{Ax} \qquad \text{if } b \land \boldsymbol{P} \models 0 \leq e$$

5.3.3 Soundness and Completeness

Definition 5.3.8 *(Soundness)* If a property can be proven, then it holds. As a result, an unsound derivation system does not provide any guarantees, as we may miss errors (we may have false negatives).

Definition 5.3.9 *(Completeness)* If a property holds, then it can be proven. As a result, we may have a correct program that we can't prove in an incomplete derivation system (we may have false positives)

Both can be proven with regard to an operational semantics.

Example 5.3.10 The partial correctness triple $\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$ is valid, written $\models \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$ if and only if

$$\forall \sigma, \sigma'. \mathcal{B}[\![\boldsymbol{P}]\!]\sigma = \texttt{tt} \land \vdash \langle s, \sigma\rangle \to \sigma' \implies \mathcal{B}[\![\boldsymbol{Q}]\!]\sigma' = \texttt{tt}$$

More generally:

- ***Soundness*** $\vdash \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \implies \models \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$
- ***Soundness*** $\models \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \implies \vdash \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$

Soundness, Completeness — **Theorem 5.3.11**

For all partial correctness triples $\{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$ of IMP we have

$$\vdash \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\} \Leftrightarrow \models \{\boldsymbol{P}\}\ s\ \{\boldsymbol{Q}\}$$

6 Modelling

6.1 Model Checking

Model Checking **Definition 6.1.1**

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Model checkers enumerate all possible states of a system, either through explicitly representing state through concrete values or symbolically through (boolean) formulas.

They are primarily used to analyze system ***designs***, and not implementations and are often used to analyze deadlocks, the reachability of undesired states and protocol violations.

6.1.1 The Model Checking Process

The first and most important phase is the ***modeling phase***, where we model the system in the description language of the model checker (here Promela). It also includes formalizing the properties to be checked in said language.

Next, we run the model checker to check the validity of the model. In the case of this course, we use `spin`, and we can run a Promela model using `spin -x <promela file>.pml`, which wraps `spin -a <promela file>.pml`, `gcc <promela file>.c` and `./a.out` into a single command.

After running, it is time to analyze the output of the model checker. If the property is violated, analyze the found counter example. If the model is too large, it can happen that the checker runs out of memory. In that case, reduce the model and try again.

6.2 Promela

Promela has C-like syntax, and its main objects are processes, channels, and variables.

An important consideration always is the number of states there are for each model, if spin can complete executing.

The number of states is given by

$$\prod_{i=1}^{N}(l(p_i) \times \prod_{\texttt{var} x_i \in p_i} |\texttt{dom}(x_i)|) \times \prod_{j=1}^{K} |\texttt{dom}(c_j)|^{\texttt{cap}(c_j)}$$

where $l(p_i)$ returns the number of program locations for process i, $|\texttt{dom}(x_i)|$ denotes the number of values a variable can take, $\texttt{dom}(c_j)$ denotes the number of values each message in the channel can take and $\texttt{cap}(c_j)$ returns the capacity of the buffer of the channel.

THUS: Keep the model as small as possible to prevent the above, which is called ***state space explosion***

On the next page, the basic syntax of Promela is presented.

```
// --- Constants ------------------------------------------------
// As C preprocessor macros
#define N 5

// Symbolic constant
mtype = { ack, req };


// --- Typedef --------------------------------------------------
// Structure declarations
typedef vector { int x; int y };


// --- Channels -------------------------------------------------
// Channels are used to exchange messages between processes
chan buf = [2] of { int }; // can store up to 2 integers
chan buf2 = [2] of { mtype, bit, chan }; // Messages are Triples
chan channel = [0] of { int }; // No buffer
chan unassigned_chan; // This channel is unassigned


// --- Variables ------------------------------------------------
// Note that there are no floats and unbounded integers.
// Variables are initialized to 0 (ish) values
// If defined outside a process, they are global, if inside,
// they are scoped to said process
bit bit_val; // 1 bit
bool bool_val; // Equivalent to bit, also 1 bit
byte counter; // 1 byte (0...255)
short short_val; // 2 bytes (-2^15 ... 2^15 - 1)
int val; // 4 bytes
int arr[4]; // 16 bytes, array of four integers
vector v; // using the custom vector type
mtype msg = ack; // Using the symbolic constant


// --- Functions ------------------------------------------------
// Note that these are not full functions and recursion is not supported!
inline fun() {
    // function body goes here
}


// --- Processes ------------------------------------------------
// Process declarations
proctype myProc(int p) {
    // Like the init body, any promela statements go in here
    // This includes any variable or channel declarations
    printf("My process");
}

active [N] proctype myActiveProc(int p) {
    // Body goes here
}
```

```
55
56
57 // --- Initial state ------------------------------------------
58 init {
59     printf("Hello World");
60 }
```

6.2.1 Expressions

Expressions in Promela can be:

- Variables, constants, and literals
- Structure and array accesses
- Unary and binary expressions with operators. The operators correspond to the `C` operators
- Function applications
- Ternary operators / conditional expressions `E1 -> E2 : E3`

Promela has a number of built in functions, which are:

- `len()`
- `nempty()`
- `nfull()`
- `eval()`
- `pcvalue()`
- `empty()`
- `full()`
- `run <proc>`
- `enabled()`

6.2.2 Statements

The following statement types are supported by Promela:

- `skip`: Does not change the state (except the location counter). Always executable
- `assert(E)`: Aborts execution if `E` evaluates to zero, otherwise is equivalent to `skip`. Always executable
- Assignment: `x = E` assigns value of `E` to variable `x`. For arrays, use `a[n] = E`. Always executable
- `s1;s2` (Sequential composition): Executable if `s1` is executable
- Expression statement: Evaluates expression `E`, executable if `E` evaluates $\neq 0$. `E` must be ***side effect free***.

In addition, selection statements (i.e. if / switch) and repetitions (loops) are supported:

```
1  if
2  :: s1 -> code;
3  :: s2 -> code;
4  :: code; // The else statement, executes if no other option executable
5  fi
6
7  do
8  :: s1 -> loop_body_1; // We can use this technique to combine if and loops
9  :: s2 -> loop_body_2;
10 :: else -> break;
11 od
```

Then, we have atomic statements, which has signature `atomic { s }`, which executes `s` atomically.

6.2.3 Macros

Promela does *not* support procedures. However, many of the effects (apart from recursion) can be achieved with macros.

We define them using

```
inline name(arg1, arg2) { /* body */ }
```

As is the case in `C`, they are simply replaced in the code and thus have no new variable scope, support no recursion and have no return value.

6.3 Linear Temporal Logic

6.3.1 Transition System of a Promela Model

For the transition systems, compared to earlier, we use a fixed initial configuration because we only model one program or system, as opposed to all programs of a programming language, as previously. In addition, we omit terminal configurations from the definition, which simplifies the theory, and termination can be modelled by a transition into a special state, the ***sink state***, which only allows further transitions back to itself.

6.3.1.1 Converting a Promela model into a transition system

The configurations are the states of the model, such as global variables and channels, as well as per active process the local variables, channels, and location counters.

The initial configuration is the initial state of the model.

The transition relation is defined by the operational semantics of the statements. They are kept informal in this course.

A Promela model has a finite number of states, since there are a finite number of allowed active processes (255), as well as finite numbers of variables and channels, finite ranges of variables and finite buffers of channels.

Computations

S^ω is the set of infinite sequences of elements of set S and $s_{[i]}$ denotes the i-th element of the sequence $s \in S^\omega$

$\gamma \in \Gamma^\omega$ is a ***computation*** of a transition system if:

- $\gamma_{[0]} = \sigma_I$
- $\gamma_{[i]} \to \gamma_{[i+1]}$ (for all $i \geq 0$)

Note that we use σ to range over the states Γ of a transition system. Here $\gamma_{[i]}$ denotes the i-th element in $\gamma = \sigma_0\sigma_1 \dots$

We use $\mathcal{C}(TS)$ to denote the set of all computations of a transition system TS

6.3.2 Linear Time Properties

LT-Properties can be used to specify the permitted computations of a transition system. A linear time property P over Γ is a subset of Γ^ω, where P specifies a particular set of infinite sequences of configurations.

A transition system TS ***satisfies*** the LT-Property P (over Γ), denoted $TS \models P$, if and only if $\mathcal{C}(TS) \subseteq P$. Intuitively, this means that all computations of TS belong to the set P.

LT-Properties ***precisely*** express properties of computations. Non-termination is handled using infinite sequences and non-determinism is handled by considering each computation separately.

To make notation cleaner and quicker, we define ***atomic propositions*** (AP) for a transition system TS, which is a proposition that does not contain any logical connectives. For example, $AP = \{open, closed\}$ (for files).

We then provide a ***labeling function*** $L : \Gamma \to \mathcal{P}(AP)$ that maps configurations to sets of atomic propositions from AP. $L(\sigma)$ is called an ***abstract state***. AP and L are both considered to be part of the transition system.

6.3.2.1 Traces

A trace is an abstraction of a computation, which only observes the propositions of each state, not the concrete state itself. It is an infinite sequence of abstract states $\mathcal{P}(AP)^\omega$.

$t \in \mathcal{P}(AP)^\omega$ is a trace of a transition system TS if $t = L(\gamma_{[0]}L(\gamma_{[1]}) \dots$ and γ is a computation of TS. The set of all traces of TS is $\mathcal{T}(TS)$. The LT-Properties are typically specified over infinite sequences of abstract states, rather than over sequences of configurations.

6.3.2.2 Safety Properties

Definition 6.3.1 An LT-property P is a safety property if for all infinite sequences $t \in \mathcal{P}(AP)^\omega$: if $t \notin P$ then there is a finite prefix $\widehat{t}$ of t such that for every t' with prefix $\widehat{t}$, $t \notin P$.

Intuition: More informally, it *does not allow anything bad to happen*. Or, more exhaustively, if a sequence t is not allowed by the LT-property, then there exists a finite prefix $\widehat{t}$, which contains everything that makes the sequence violate the LT-property. Thus, whatever sequence we append to it, it will always remain in violation of the property P.

6.3.2.3 Liveness Properties

Definition 6.3.2 An LT-property P is a liveness property if every finite sequence $\widehat{t} \in \mathcal{P}(AP)^*$ is a prefix of an infinite sequence $t \in P$

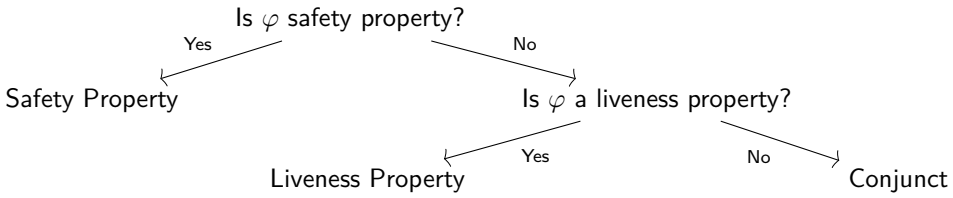
Intuition: More informally, it states that *something good will happen eventually*. Or, more exhaustively, if every finite sequence of atomic propositions can be extended to a sequence that is allowed by the LT-property.

6.3.2.4 On proves and examples

Some remarks for common exam multiple-choice questions

- Safety Property and Liveness Property: Only `true` is such an example
- Neither: There are no examples, since every LT-property is a conjunct of a safety and liveness property and every LT-property can be rewritten as such a conjunct.

Typically, the exam includes questions on whether or not a given LT-property φ is a liveness property, safety property or a conjunct of both. To determine that, use the following flow chart:



6.3.3 Linear Temporal Logic

This is used to formalize LT-properties of traces.

6.3.3.1 Operators

The basic operators are, with p a proposition from $AP \neq \emptyset$, and Φ and Ψ LTL formulas:

- p states that it is true "*now*"
- $\Phi\ U\ \Psi$ states that Φ holds "*until*" Ψ holds. I.e. there are no other valid propositions that hold in between.
- $\bigcirc\Phi$ states that the Φ holds for the "*next*"

6.3.3.2 Semantics

$t \models \Phi$ expresses that a trace $t \in \mathcal{P}(AP)^\omega$ satisfies the LTL formula Φ

- $t \models p$ if and only if $p \in t_{[0]}$
- $t \models \neg\Phi$ if and only if not $t \models \Phi$
- $t \models \Phi \wedge \Psi$ if and only if $t \models \Phi$ and $t \models \Psi$
- $t \models \Phi U \Psi$ if and only if there is a $k \geq 0$ with $t_{(\geq k)} \models \Psi$ and $t_{(\geq j)} \models \Phi$ for all j with $0 \leq j \leq k$
- $t \models \bigcirc\Phi$ if and only if $t_{(\geq 1)} \models \Phi$

6.3.3.3 Derived operators

There are a number of handy derived operators defined here that we can use, unless otherwise specified.

- Eventually: $\Diamond\Phi \equiv (\texttt{true}\ U\Phi)$
- Always (from now on): $\Box\Phi \equiv \neg\Diamond\neg\Phi$

Precedences are unary operators (such as $\Diamond\Phi \Rightarrow \Psi$ means $(\Diamond\Phi) \Rightarrow \Psi$), but we use parenthesis to avoid ambiguities.

6.3.3.4 Useful patterns

- Strong invariant $\square\Psi$: Ψ always holds and this is a safety property if Ψ is a safety property. For instance, a file is always open or closed ($\square(\texttt{open} \vee \texttt{closed})$)
- Monotone invariant $\square(\Psi \Rightarrow \square\Psi)$: Once Ψ is `true`, it will always be `true`. It is a safety property if Ψ is a safety property. For instance, once information is public, it can never be private again ($\square(\texttt{public} \Rightarrow \square\texttt{public})$)
- Establishing an invariant $\diamond\square\Psi$: Eventually, Ψ will *always* hold. It is a liveness property if Ψ is satisfiable. For instance, system initialization starts server ($\diamond\square\texttt{serverRunning}$)
- Responsiveness $\square(\Psi \Rightarrow \diamond\Phi)$: Every time that Ψ holds, Φ will eventually hold. It is a liveness property if Φ satisfiable. For instance, all opened files must be closed eventually ($\square(\texttt{open} \Rightarrow \diamond\texttt{closed})$)
- Fairness $\square\diamond\Psi$: Ψ holds infinitely often. It is a liveness property if Ψ is satisfiable. For instance, a producer does not wait infinitely long before entering the critical section ($\square\diamond\texttt{critical}$)

7 Exercises

7.1 Introduction

This section aims to give you an overview over how to solve different exercise types. An understanding of the rest of the topics covered in the lecture is expected, as well as understanding of haskell. There is a short section on haskell in this section, too.

FMFP is known to feature an exam where time is at a premium, so the most important preparation likely is solving as many old exams as you can, to really get fast at solving the exercises in there, as the exams have been following a similar format, with predictable exercise types, for a long time.

7.2 Functional Programming

7.2.1 Evaluation strategies

At exam: It is likely that one such task will appear.

Evaluation strategies formalize how programming languages evaluate the code. Most of the commonly used programming languages use *eager evaluation*, whereas functional programming languages tend to prefer *lazy evaluation*

Tasks typically involve doing the compiler / interpreter's work, evaluating a given statement. This can either be full haskell code, or in the mini-Haskell, often with lambda functions instead of pattern matching and recursion, as that *tends* to be more challenging due to the possibility of losing track of what you expanded or not. An easy way around that is to use different colour highlighters.

7.2.1.1 Lazy Evaluation

In Lazy Evaluation, expressions are substituted until all possible values are substituted and the expression is then evaluated.

7.2.1.1.1 Lambda functions

For expressions of form `t1 t2`, `t1` is evaluated by substituting every occurrence of the function arguments by `t2`, without evaluation `t2`, e.g. `(\x -> x y) (\x -> x)`, the evaluation will lead to `(\x -> x) y` Below a more complicated example, highlighted with colours for you to track what goes where:

1. `(\x -> x (\y -> x y)) (\x -> (\y -> y) x)`
2. Substitute `t2` into `t1` (no eval): `(\x -> (\y -> y) x) (\y -> (\x -> (\y -> y) x) y)`
3. Recolour for next changes: `(\x -> (\y -> y) x) (\y -> (\x -> (\y -> y) x) y)`
4. Substitute again: `(\y -> y) (\y -> (\x -> (\y -> y) x) y)`
5. And again: `(\y -> (\x -> (\y -> y) x) y)`

Now, evaluation stops, because there is nothing left to apply. In some cases, this can go on until you get a final result, but *not always*. This evaluation example also makes it evident why using colours can be very handy.

7.2.1.1.2 Pattern Matching

Pattern matching is fairly straight forward, however, conditional evaluation requires evaluation to the smallest extent possible to determine the option to pick from.

7.2.1.2 Eager Evaluation

For expressions of form `t1 t2`, `t1` is evaluated by substituting every occurrence of the function arguments by an *evaluated* `t2`, i.e. `t2` is evaluated *before* substituting it. Coming back to the short example from above `(\x -> x y) (\x -> x)` is evaluated to the same `(\x -> x) y`. The difference only becomes evident with a more complicated example. We'll use the same example again as above, again highlighted.

1. `(\x -> x (\y -> x y)) (\x -> (\y -> y) x)`
2. Evaluate orange parts (now purple): `(\x -> x (\y -> x y)) (\x -> x)`
3. Substitute purple: `(\x -> x) (\y -> (\x -> x) y)`
4. Evaluate green parts (now cyan): `(\x -> x) (\y -> y)`
5. Substitute cyan: `(\y -> y)`

As you can clearly see, both strategies don't result in the same evaluation.

7.2.2 Haskell

At exam: typically either short coding task or proof of program

7.2.2.1 Programming

The best tip here is to read the Haskell book, and to solve the exercises during the semester.

Remember that application is left-associative (i.e. `func x y` parenthesized is `(func x) y`, even if `x` is a function).

A handy to know shortening technique for functions is `Leaf .  f` is equivalent to `leaf x = Leaf (f x)`. Thus, the `.` operator chains functions.

For all functions with recursion, don't forget the base cases. In addition, for guards (i.e. statements with a pipe character (|)), there is no equal sign before the pipe characters. For the cases notation, there are equal signs. We can use underscores as a "don't care" character.

Always consider making use of functions defined in previous subtasks. This can save a lot of time.

7.2.2.1.1 Lists

In list comprehensions, to draw from a list, <- is used, to delimit the description of the list contents from the generator part, we use a pipe character and to separate each statement in the generator part, we use a comma. Remember that list comprehension does allow duplicates, so it is not entirely equivalent to sets. We can use the `nub` function from `Data.List` to remove duplicates.

We can initialize infinite lists using the `..` syntax. We define the interval using `[1, 2..]`, or `[0, 0..]` to create an infinite lists of zeros.

More advanced types can be "disassembled" like this: `Node x l r` for type `Node a (Tree x) (Tree x)`

Note that for lists, `(==)` is defined if and only if it is defined for the types in the lists that we are comparing, i.e.

```
comp :: (Eq a, Eq b) => [a] -> [b] -> Bool

comp ls rs = ls == rs
```

7.2.2.1.2 Fold

One of the most important functions to understand is `foldr` (and `foldl`). If you have used `reduce` functions before, in e.g. JavaScript / TypeScript, they are similar to a `map` combined with a `reduce`.

For instance, in TypeScript, the `reduce` function has the type signature

```
array.reduce( ( accumulator: A, current: A, idx: number, array: A[] ) => A, initialValue: A )
```

In the Haskell prelude, they are defined as follows:

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z []     = z
3 foldr f z (x:xs) = f x (foldr f z xs)
4
5 foldl :: (a -> b -> a) -> a -> [b] -> a
6 foldl f z []     = z
7 foldl f z (x:xs) = foldl f (f z x) xs
```

When extending them to more complex data structures such as trees, we may need to add one function to the arguments per type of possible element in the data structure. Ideally, we first write the function, then infer its type. Remember that in the definition of these two functions, the `-> b ->` (and `-> a ->`, respectively) denote the type of the base case.

For more elaborate data structures, the functions for each subtype should be in the same order as in the data type definition, for canonical definition of the fold function. Note that these functions' type doesn't contain the data structure typically, but simply, e.g. for `Mlist a = Bot | Node [a] (Mlist a)`, the type of the canonical fold function is simply `b -> ([a] -> a -> b) -> Mlist a -> b` and not `(Mlist b) -> ([a] -> Mlist a -> Mlist b) -> (Mlist a) -> (Mlist b)`.

7.2.2.1.3 zipWith

To combine a `map` and a `zip` function, use `zipWith`, type:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

The `zip` function creates a tuple:

```
zip :: [a] -> [b] -> [(a, b)]
```

7.2.2.2 Proofs

These proofs use structural induction, often it is easiest to use strong structural induction, see Section 2 for that. In many cases, generalizing the statement is what enables the proof. So whenever there is a constant, generalize the constant before doing the proof, as otherwise the proof will likely be hard to impossible to pull off under the time constraints. In that case, we set $P(t) \equiv \forall n \in \mathbb{N}_0$ the generalized statement, or equivalent. Remember:

- for each case state the fixed variables (which are all free variables in this case)
- in the base case / simple case, fix n
- in the end state that since it holds for all n, it, in particular, holds for $n = 0$ (or equivalent)

For generalizing, a very helpful tactic is to think about the statement some, come up with the generalization you think is correct, then doing the base case in the proof after only a very short amount of time, as the correct generalized statement will become apparent there very quickly.

Remember to always check that parenthesis are set correctly and to only apply one rule exactly once.

Depending on your writing style, it may be worth using the CYP syntax (see Section 2.8), as it *can* be more concise.

7.2.3 Natural Deduction

7.2.3.1 Parenthesis

This task (if it were to even ever appear in the exams) is simply applying precedences, as well as remembering associativity. The precedences are as follows:

$$\neg > \wedge > \vee > \rightarrow$$

The associativities are right for $\wedge$ and $\vee$, whereas $\rightarrow$ is left associative. This means that

$$A \rightarrow B \rightarrow C \text{ is parenthesized as } A \rightarrow (B \rightarrow C)$$

7.2.3.2 Derivation trees

The most important tip here is to write as little as possible, by defining variables (such as Γ, Γ', etc) for each step, so you can simply write that instead of all the content.

7.2.3.3 Defining rules

This is often the hardest type of task. First state the obvious cases, the basic introduction and elimination rules according to the given definition. Remember that the rules are defined top-down, so, for introducing $A \leftrightarrow B$, defined as $(A \rightarrow B) \wedge (B \rightarrow A)$, we define the rules as follows:

$$\frac{\Gamma \vdash (A \rightarrow B) \wedge (B \rightarrow A)}{\Gamma \vdash A \leftrightarrow B} \leftrightarrow\text{-intro} \qquad \frac{\Gamma \vdash A \leftrightarrow B}{\Gamma \vdash (A \rightarrow B) \wedge (B \rightarrow A)} \leftrightarrow\text{-elim}$$

The hard part is coming up with useful extra rules. Here, and with all rules defined by an AND, we can also define it slightly differently:

$$\frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash B \rightarrow A}{\Gamma \vdash A \leftrightarrow B} \leftrightarrow\text{-I}$$

Similarly, we can also provide elimination rules for the left or right:

$$\frac{\Gamma \vdash A \leftrightarrow B}{\Gamma \vdash (A \rightarrow B)} \leftrightarrow\text{-EL} \qquad \frac{\Gamma \vdash A \leftrightarrow B}{\Gamma \vdash (B \rightarrow A)} \leftrightarrow\text{-ER}$$

7.2.4 Type Inference

At exam: There typically is one such task. They provide the inference rules

The proof trees are again drawn up bottom up, applying rules from the outside in.

1. Create a proof tree using the typing rules. An easy way to avoid mistakes is to always write out the structure of each type in the tree.
2. During the proof, keep track of what the applications of rules tell us about types in a list.
3. After reaching leafs in the prooftree, resolve the types by inserting the now known types into the types you kept in the list.

Remember that not all expressions are typeable, so if you end up getting an invalid tree or a conflict of types, this is very much possible.

To prove that the type of an expression is correct, we can simply draw up a type derivation tree, using the typing rules from Section 4.1.

7.3 Induction

For induction proofs, see Section 2, which were moved up, since they are used in both FP and FM, as well as being mentioned a few times in the rest of the summary.

7.4 Formal Methods

7.4.1 IMP Proofs

At exam: Typically, the exam contains some operational semantics tasks, rarely proofs of properties of IMP.

The property proofs can be acheived either using a direct proof of the implication, or using an induction proof. The latter of which is more common and we typically define P in such a way that we bind everything apart from variables, natural numbers and constant values using quantifiers, then prove $\forall n \in \mathbb{N}.P(n)$, if we have a free natural numbers. Before the definition of P we typically state "Let <variables, constants> be arbitrary", as with any induction proof.

7.4.2 Operational Semantics

7.4.2.1 Big Step Semantics (Natural Semantics, NS)

Proving something using the big step semantics rules tends to be fairly easy if you understand how natural deduction and the like work. The concept is very similar, just with other rules. It is advisable to name each sub-statement in the main statement, to reduce the required amount writing.

Sometimes, Induction on (the shape of) Proof Trees may be required, for that, see Section 2.5

Where it gets more interesting and challenging, is providing rules in the big step semantics. Here, it is important to think about the different cases that could be introduced by the statement.

For instance, for loops, there will (almost certainly) be two cases, one in which some boolean expression (denoted $\mathcal{B}[\![e]\!]\sigma$) is `ff`, the other in which it is `tt`. For these, you need to make sure to state the side condition (i.e. that the boolean expression is true or false, respectively).

In addition, think thoroughly about how to define the states of the rules. For instance, for a `time` statement, that counts the number of assignments, the state we map to would ideally be defined as $\sigma'[x \mapsto n]$, where n is the number of assignments.

7.4.2.2 Small Step Semantics (Structural Operational Semantics, SOS)

Small Step semantics proofs tend to be a bit more challenging. Unlike Natural Semantics, here we can't create a single derivation tree, but one per step. This means, that we do one single step transition at a time, denoted $\rightarrow_1^1$, and justify each step with a proof tree and SO Semantics (SOS). Typically, only *some* of the statements are asked to be proven in the exam, as proving all is time intensive due to the typically fairly large number of steps in the proofs. For the proof trees, they need to be complete for each transition that we prove, i.e. no hypothesis in any branches.

Defining rules here is similar in concept as with the Natural Semantics. We want to make sure, to also consider the effects it has on other rules, as with Natural Semantics.

The sole difference here is that we return *both* the next statement to be evaluated *and* the next state, unlike with Natural Semantics, where we only return the next state.

7.4.2.3 Other proofs

Induction proofs for these two are also outlined in Section 2.5 and subsequent subsections.

7.4.3 Axiomatic Semantics

Here, we need to find pre- and postconditions for expressions and prove that they are correct.

To do the proofs, we again apply transition rules, as was the case already with operational semantics. Contrary to those however, we have pre- and postconditions, which we typically need to define ourselves.

This typically involves finding a loop invariant that holds before and after each iteration of the loop. This invariant should mention every variable used in the loop. Any other variable should also be mentioned in it. The loop *variant* may also be added for proving termination. A typical for-loop loop variant would be `n - x = Z`, or the same for a while loop like `while i < n do s end`, as the next value of the loop variant has to be lower than the previous one.

The preconditions and postconditions in the loops then reflect that the update to the loop variant variable(s) leads to them being smaller than Z, which allows proving that the loop variant is decreasing, signifying that the loop terminates *eventually*. Of course, if `x` is decreasing, it itself can become the variant, as it fulfils the condition.

Proof outlines work by providing post and pre-condition for each sub-statement in a statement. If we need to rewrite a statement (e.g. before the first loop body to change to our loop invariant from the overall precondition), we use the $\models$ symbol.

There are also tasks in which we need to find errors in rule applications. This works "simply" by checking that the rules are applied correctly and semantic entailment and general transformation rules hold for the transformations.

7.4.4 Modelling

Many of the tasks here are pretty straight forward, converting IMP into Promela, and putting an `assert` (or more) into the `init` block, to check if the required state is reached.

Note that for non-determinism, we use Promela conditionals without conditions. This can also be used to simulate all possible combinations / actions that can be taken.

For parallelism, we can use the `run` keyword for a `proctype` (rest of syntax is just as with a function in most C-like programming languages), then we wait for the processes to terminate using `_nr_pr == 1` and do our assertion.

We can use an `atomic` block to ensure atomicity.

Promela ***does not*** support functions, instead, the `inline` keyword works similarly to macros in `C`, they are simply substituted into the code at compile time. In some solutions, they use double-dash arrows (`-->`). From testing, it seems that it is also fine to use normal arrows (`->`) and it will still work. For completeness, the double-dash arrows are used in if statements with conjuncts or disjuncts.

If we are given a text description only however, things get more challenging, as we need to model the properties given there. A possible simplification of the process may be to model things one property at a time, then combine them as needed.

To check if something is achievable (i.e. there is a solution to the task), we can use an `assert false` on the case where we have success If there is a way, then the assertion will fail (because it will always fail and is only called if `success` is true)

7.4.5 Linear Time Properties (LTL)

We refer to Section 6.3 for intuition, as these tend to mostly be intuition exercises.

For doing verification of liveness / safety properties using `spin` and Promela, the operators are translated to Promela as follows: `[] <> ! && || -> U` for $\square\diamond\neg\wedge\vee\Rightarrow U$, respectively. In addition, we have the equality operator as a primitive propositional formula. We have `p == true` for p in the operators.

Execute `spin -a file.pml`, followed by `gcc pan.c` and `./a.out`.

As a reminder, these are the operators (details in Section 6.3.3):

- Now: p states that the proposition is true "*now*"
- Holds until: $\Phi U \Psi$ states that Φ holds (with no other valid proposition holding between) until Ψ holds.
- Next: $\bigcirc\Phi$ states that Φ holds for the "next" (if nothing else specified from start state)
- Eventually: $\diamond\Phi$ states that Φ holds *eventually*, $\diamond\Phi \equiv (\texttt{true} U \Phi)$
- Always (from now on): $\square\Phi$ states that from now on, Φ will always hold, $\square\Phi \equiv \neg\diamond\neg\Phi$

For the implication (where the left hand side is called the *antecedent*, right hand side is called the *consequent*), remember to show the case where the antecedent is true and state that for antecedent false, it is trivially true

7.4.5.1 Tips and tricks

- You can't just write $\diamond\neg s_1$, where s_1 is a state, you need to specify the propositions (as a set, such as $\{A, B\}$ instead of the s_1.
- We can also create statements, such as never as $\square\neg\Phi$ (always not Φ), etc.
- Often, an `if` in the description means we should use a $\Rightarrow$.
- A typical task is "something will do something (denoted A here) N times" (N in that case known an low), an LTL formula in the $N = 2$ case is then $\diamond(A \wedge \bigcirc\diamond A)$. Note that $\diamond(A \wedge \diamond A)$ is ***NOT*** correct (because that is true also for A happening only once).
- If property A has to be true infinitely many times, the LTL formula is $\square\diamond A$
- Be careful with parenthesis (e.g. with $\neg$)!

7.4.5.2 Liveness and Safety Properties

Proofs here run using the definitions directly, either by showing a counter example (for disproving) or showing that, in fact, the definition holds. Indirect proofs may also come in handy, because it is typically easier to show that something is not a safety property (or liveness property) than to show that it is.

These two properties are *mutually exclusive* (with one exception, `true`), to the extent that an LTL formula can't be both at the same time, but be a conjunct of both. In fact, every LTL formula is a either one of the two, or a conjunct of both.

Finally, as a reminder, an LTL formula is for example $\square\diamond A$, with A an atomic proposition.

7.5 Checklist

7.5.1 Learning

These are (some of) the things that we need to learn by heart for the exam

- ☐ Haskell prelude functions (see summary of prelude on ComSol)
- ☐ Definition of fold functions
- ☐ Definition of canonical function
- ☐ LTL symbols
- ☐ Induction scheme (especially structural induction and shape of derivation tree)
- ☐ Ideally for shortness the CYP syntax (roughly, they said they'd not deduct points for incorrect CYP syntax)
- ☐ Promela syntax
- ☐ Read through PVW script (it's VERY good and has some handy tips and tricks for all types of exercises)

7.5.2 What to bring to the exam

- ☐ Multiple colours of highlighters (for eval tasks)
- ☐ Enough pencils
- ☐ Probably won't be needed / allowed, but some spare paper (for notes)