

Formal Methods & Functional Programming

Janis Hutz
<https://janishutz.com>

June 7, 2026



“A funny quote by a professor”

- Prof. Dr. Professor Name, YEAR

FS2026, ETHZ
Summary of the Lectures

Contents

1	Haskell	4
1.1	The bad news: The syntax	4
2	Formal Reasoning	5
2.1	Formal proofs	5
2.2	Natural deduction	5
2.3	Propositional logic	5
2.3.1	Syntax	5
2.3.2	Semantics	5
2.3.3	Requirements for a deductive system	6
2.3.4	Natural deduction for propositional formulas	6
2.3.5	Derivation rules for propositional logic	6
2.4	First-Order Logic	7
2.4.1	Syntax	7
2.4.2	Semantics	7
2.4.3	Quantifiers	8
2.5	Equality	8
2.6	Correctness	9
2.6.1	Termination	9
2.6.2	Correctness - Behaviour	9
2.6.3	Induction	10
3	Typing	11
3.1	Mini-Haskell	11
3.1.1	Syntax	11
3.1.2	Lambda calculus	11
3.1.3	Further rules for mini-Haskell	11
3.1.4	Type inference	11
3.2	Natural Number Proofs	12
3.2.1	Induction over the natural numbers	12
3.2.2	Lists	12
3.2.3	Trees	12
3.2.4	Structural Induction	13
3.3	Interpreters	14
3.3.1	Read step	14
3.4	Evaluation	17
3.4.1	Lazy Evaluation	17
4	Language Semantics	18
4.1	IMP Language	18
4.1.1	The syntax	18
4.1.2	The semantics	18
4.1.3	Properties of expression semantics	19
4.2	Operational Semantics	21
4.2.1	Big-Step Semantics	21
4.2.2	Small-Step semantics	24
4.2.3	Equivalence	26
4.2.4	Operational Semantics Rules Overview	27
4.3	Axiomatic Semantics	28
4.3.1	Program Correctness	28
4.3.2	Hoare Logic	29
4.3.3	Soundness and Completeness	30
5	Modelling	31
5.1	Model Checking	31
5.1.1	The Model Checking Process	31

5.2	Promela	31
5.2.1	Expressions	33
5.2.2	Statements	33
5.2.3	Macros	33

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. statements 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 Formal Reasoning

2.1 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:

$$\frac{\frac{+}{\otimes} \alpha \quad \frac{+}{\times} \beta}{\frac{\otimes \quad \times}{\oplus} \gamma} \quad \frac{-}{+} \delta$$

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

$$\frac{\frac{-}{+} \delta \quad \frac{-}{+} \delta}{\frac{\frac{\otimes}{\otimes} \alpha \quad \frac{\times}{\times} \beta}{\oplus} \gamma}$$

2.2 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.

2.3 Propositional logic

2.3.1 Syntax

Definition 2.3.1 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 $X \in \mathcal{V}$
- $A \vee B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$
- $\perp \in \mathcal{L}_P$
- $A \rightarrow B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$
- $A \wedge B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$

2.3.2 Semantics

Definition 2.3.2 (*Valuation*) $\sigma : \mathcal{V} \rightarrow \{\text{True}, \text{False}\}$ maps variables to truth values. They are the simple models (i.e. *interpretations*). **Valuations** is the set of valuations.

Definition 2.3.3 (*Satisfiability*) smallest relation $\models \subseteq \text{Valuations} \times \mathcal{L}_P$ such that

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

Definition 2.3.4 (*satisfiable formula*) A formula $A \in \mathcal{L}_P$ is **satisfiable** if $\sigma \models A$ for **some** valuation σ

Definition 2.3.5 (*tautology, valid formula*) A formula $A \in \mathcal{L}_P$ is **valid** (a **tautology**) if $\sigma \models A$ for **all** valuations σ

Definition 2.3.6 (*Semantic entailment*) $A_1, \dots, A_n \models A$ if $\forall \sigma$ we have $\sigma \models A_1, \dots, \sigma \models A_n$, then $\sigma \models A$

2.3.3 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.

2.3.4 Natural deduction for propositional formulas

Definition 2.3.7 (*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 2.3.8 (*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 2.3.9 (*Proof*) of A is a derivation tree with 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.

2.3.5 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)

2.3.5.1 Conjunction

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

2.3.5.2 Disjunction

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

2.3.5.3 Implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow\text{-I} \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow\text{-E}$$

2.3.5.4 Others

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

2.4 First-Order Logic

2.4.1 Syntax

Definition 2.4.1 (*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 2.4.2 (*Term*) is the terms of first-order logic is smallest set, where (with \mathcal{V} again a set of variables)

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

Definition 2.4.3 (*Form*) is the formulas of first-order logic, is the smallest set where

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

2.4.1.1 Binding

Definition 2.4.4 (*Bound variable*) A variable that occurs in a quantifier in scope (blue in example below)

Definition 2.4.5 (*Free variable*) A variable that is not bound by a quantifier in scope (red in example below)

Example 2.4.6 $(q(x) \vee \exists x. \forall y. p(f(x), z) \wedge q(a)) \vee \forall x. r(x, z, g(x))$

Definition 2.4.7 (α -conversion) A **bound** variable can be renamed at any time, but must preserve binding structure.

2.4.1.2 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.

2.4.2 Semantics

Definition 2.4.8 (*Structure*) is a pair $\mathcal{S} = \langle U_{\mathcal{S}}, I_{\mathcal{S}} \rangle$, where $U_{\mathcal{S}}$ is the universe and it is a non-empty set and $I_{\mathcal{S}}$ is a mapping with

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 2.4.9 (*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 2.4.10 (*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 2.4.11 (*Satisfiability*) $\models \subseteq \text{Interpretations} \times \text{Form}$ is the smallest relation satisfying

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

Definition 2.4.12 (*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 2.4.13 (*Validity*) When every interpretation is a model, we write $\models A$, and we say that A is **valid**

Definition 2.4.14 (*Satisfiability*) A is **satisfiable**, if there exists at least one model for A .

Example 2.4.15 Given $\forall x.p(x, s(x))$, we a model would be

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

2.4.2.1 Substitution

Definition 2.4.16 (*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.

2.4.3 Quantifiers

2.4.3.1 Universal quantification

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

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \forall\text{-I}^* \quad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \rightarrow t]} \forall\text{-E}$$

Again here be mindful not to capture free variables.

2.4.3.2 Existential quantification

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

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

Again here be mindful not to capture free variables.

2.5 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 $t_1 = t_2 \in \text{Form}$ if $t_1, t_2 \in \text{Term}$, 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):

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

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

$$\frac{\Gamma \vdash t_1 = s_1 \quad \dots \quad \Gamma \vdash t_n = s_n}{\Gamma \vdash f(t_1, \dots, t_n) = f(s_1, \dots, s_n)} \text{cong}_1$$

$$\frac{\Gamma \vdash t_1 = s_1 \quad \dots \quad \Gamma \vdash t_n = s_n \quad \Gamma \vdash p(t_1, \dots, t_n)}{\Gamma \vdash p(s_1, \dots, s_n)} \text{cong}_2$$

2.6 Correctness

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

2.6.1 Termination

Theorem 2.6.1 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 2.6.2 Sufficient condition for termination: The arguments are smaller along a **well-founded** order on the function's domain

Definition 2.6.3 (*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 2.6.4 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 $s_0 R s_1 R \dots R s_{i-1} R s_i$

Definition 2.6.5 $R^+ \equiv \bigcup_{n \geq 1} R^n$, where $R^n \equiv R \circ R^{n-1}$ for $n \geq 2$ and $R^1 \equiv R$

Theorem 2.6.6 If $>$ is a well-founded order on S , then $>^+$ is also well-founded on S

2.6.2 Correctness - Behaviour

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

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

The two functions are given as follows:

<pre> 1 fac :: Int -> Int 2 fac 0 = 1 3 fac n = n * fac (n - 1)</pre>	<pre> 1 fac2 :: (Int, Int) -> Int 2 fac2 (0, a) = a 3 fac2 (n, a) = fac2 (n - 1, n * a)</pre>
--	--

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 2.6.7 Consider the Haskell function below

```

1 maxi :: Int -> Int -> Int
2 maxi n m
3   | n >= m    = n
4   | 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.

So 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

2.6.3 Induction

To prove recursive formulas, or more precisely formulated, a formula P (with free variable n) for all $n \in \mathbb{N}$, we can't really do a proof by cases, as there are infinitely many cases (one for each input). Thus: We can use induction to prove recursive formulas or functions.

2.6.3.1 The schema

To prove $\forall n \in \mathbb{N}.P$ (with n free in P), we do the following:

Base case We show that $P[n \mapsto 0]$ is correct

Step case For an arbitrary m not free in P , we show that $P[n \mapsto m + 1]$ is correct under the assumption that $P[n \mapsto m]$

For **well-founded** domains, we have to adjust the induction hypothesis slightly: We assume $\forall l \in \mathbb{N}.l < m \rightarrow P[n \mapsto l]$ and then prove $P[n \mapsto m]$ under our assumption.

2.6.3.2 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 $\forall y :: T, ys :: [T].P[xs \mapsto ys] \rightarrow P[xs \mapsto y : ys]$, 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 $P[xs \mapsto y : ys]$

3 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.

3.1 Mini-Haskell

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

3.1.1 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:

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

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

The types are given by $\tau ::= \mathcal{V}_T \mid \text{Bool} \mid \text{Int} \mid (\tau, \tau) \mid (\tau \rightarrow \tau)$, 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.

3.1.2 Lambda calculus

To prove that types are correct, the lambda calculus comes in handy. It is based on the same concept as natural deduction trees

3.1.2.1 Core rules for Lambda-Calculus

$$\frac{}{\Gamma, x : \tau \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x.t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}$$

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

3.1.3 Further rules for mini-Haskell

3.1.3.1 Base types

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

3.1.3.2 Operations

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

$$\frac{\Gamma \vdash t :: \text{Int}}{\Gamma \vdash (\text{iszero } t) :: \text{Bool}} \text{iszero} \quad \frac{\Gamma \vdash t_1 :: \text{Int} \quad \Gamma \vdash t_2 :: \text{Int}}{\Gamma \vdash (t_1 \text{ op } t_2) :: \text{Int}} \text{BinOp} \\ \frac{\Gamma \vdash t_0 :: \text{Bool} \quad \Gamma \vdash t_1 :: \tau \quad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\text{if } t_0 \text{ then } t_1 \text{ else } t_2) :: \tau} \text{if}$$

3.1.3.3 Tuples

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

3.1.4 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.

3.1.4.1 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**

3.1.4.2 Curry-Howard isomorphism

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

$$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \rightarrow \tau} \rightarrow\text{-I} \quad \frac{\Gamma \vdash \sigma \rightarrow \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau} \rightarrow\text{-E}$$

3.2 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:

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

3.2.1 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

$$\text{Nat} = \{\text{Zero}, \text{Succ Zero}, \text{Succ (Succ Zero)}, \dots\}$$

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

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

The induction stated as a natural deduction rule:

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

3.2.2 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:

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

3.2.3 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:

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

3.2.4 Structural Induction

Induction is based on the structure of terms

data $\mathbf{T} \ t = \mathbf{Leaf} \ t \mid \mathbf{Node1} \ (\mathbf{T} \ t) \mid \mathbf{Node2} \ t \ (\mathbf{T} \ t) \ (\mathbf{T} \ t)$

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

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

A natural deduction rule structural induction is:

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

(*) a, l, r, s not free in Γ, P

3.3 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.

3.3.1 Read step

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

3.3.1.1 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 <code>position</code> | 5. Identifier <code>rate</code> |
| 2. Assignment symbol <code>:=</code> | 6. Addition symbol <code>+</code> |
| 3. Identifier <code>initial</code> | 7. Number <code>60</code> |
| 4. Addition symbol <code>+</code> | |

It also removes whitespaces and comments

3.3.1.2 Parsing

The tokens are then turned into an abstract syntax tree.

The syntax is specified by a grammar such as:

$$\begin{aligned} \text{Expr} &::= \text{Identifier} \mid \text{Number} \mid \text{Expr} \text{ '+' } \text{Expr} \\ \text{Assign} &::= \text{Identifier} \text{ ' := ' } \text{Expr} \end{aligned}$$

This can also be represented as a haskell type:

```

1 data Expr = Identifier Ident | Number Num | Plus Expr Expr
2 data Assign = Assignment Ident Expr
3 type Ident = String
4 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:

```

1  {-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
2
3  {-# HLINT ignore "Use lambda-case" #-}
4  {-# HLINT ignore "Use newtype instead of data" #-}
5
6  import Prelude hiding (return, (>>), (>>=))
7
8  data Parser a = Prs (String -> [(a, String)])
9
10 -- Main parser function
11 parse :: Parser a -> String -> [(a, String)]
12 parse (Prs p) = p
13
14 -----
15 -- Basic parsers --
16 -----
17 -- Trivial failure ([] signifies parse failed)
18 failure :: Parser a
19 failure = Prs (const [])
20
21 -- Trivial success without progress
22 return :: a -> Parser a
23 return x = Prs (\inp -> [(x, inp)])
24
25 -- Trivial success with progress
26 item :: Parser Char
27 item =
28     Prs
29     ( \inp -> case inp of
30         "" -> []
31         (x : xs) -> [(x, xs)]
32     )
33
34 -----
35 -- Glue --
36 -----
37 -- Apply both parsers
38 (|||) :: Parser a -> Parser a -> Parser a
39 p ||| q = Prs (\s -> parse p s ++ parse q s)
40
41 -- If first parser fails, apply second parser
42 (+++) :: Parser a -> Parser a -> Parser a
43 p +++ q =
44     Prs
45     ( \s -> case parse p s of
46         [] -> parse q s
47         res -> res
48     )
49
50 -- Sequencing (first parser p, then parser q)
51 (>>=) :: Parser a -> (a -> Parser b) -> Parser b
52 p >>= g = Prs (\s -> [(u, s'') | (t, s') <- parse p s, (u, s'') <- parse (g t) s'])
53
54 -- Simple version of the above

```

```
55 (>>) :: Parser a -> Parser b -> Parser b
56 p >> q = p >>= const q
57
58 -- Parse single character with property p
59 sat :: (Char -> Bool) -> Parser Char
60 sat p = item >>= \x -> if p x then return x else failure
61
62 char :: Char -> Parser Char
63 char x = sat (== x)
64
65 string :: String -> Parser String
66 string "" = return ""
67 string (x : xs) = char x >> string xs >> return (x : xs)
68
69 -- 0 or more repetitions of p
70 many :: Parser a -> Parser [a]
71 many p = many1 p ||| return []
72
73 -- 1 or more repetitions of p
74 many1 :: Parser a -> Parser [a]
75 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 this that was the case at least)

3.4 Evaluation

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

3.4.1 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.

3.4.1.1 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)

3.4.1.2 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.

3.4.1.3 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

4 Language Semantics

4.1 IMP Language

4.1.1 The syntax

The allowed characters:

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

And the allowed tokens:

```
1 Ident = Letter { Letter | Digit }*
2 Numeral = Digit | Numeral Digit
3 Var = Ident
```

Arithmetic and Boolean statements could be defined in Haskell as follows

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

with the abbreviations for Eq (=), Neq (#), Le (<), Leq (≤), Ge (>) and 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

4.1.2 The semantics

4.1.2.1 Numerals

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

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

4.1.2.2 States

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

$$\sigma : \text{Var} \rightarrow \text{Val}$$

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: $\sigma_1 = \sigma_2 \Leftrightarrow \forall x. (\sigma_1(x) = \sigma_2(x))$

4.1.2.3 Arithmetic Expressions

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

$$\begin{aligned}\mathcal{A}[\![x]\!]\sigma &= \sigma(x) \\ \mathcal{A}[\![n]\!]\sigma &= \mathcal{N}[\![x]\!] \\ \mathcal{A}[\![e_1 \text{ op } e_2]\!]\sigma &= \mathcal{A}[\![e_1]\!]\sigma \text{ op } \mathcal{A}[\![e_2]\!]\sigma\end{aligned}$$

For $\text{op} \in \text{Op}$, op is the corresponding operation $\text{Val} \times \text{Val} \rightarrow \text{Val}$

4.1.2.4 Boolean Expressions

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

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

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

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

not is defined as follows:

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

4.1.3 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

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

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

4.1.3.1 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 $-e$

4.1.3.2 Free Variables

For Arithmetic Expressions

$$FV(e_1 \text{ op } e_2) = FV(e_1) \cup FV(e_2)$$

$$FV(n) = \emptyset$$

$$FV(x) = \{x\}$$

For Boolean Expressions

$$FV(b_1 \text{ op } b_2) = FV(b_1) \cup FV(b_2)$$

$$FV(\text{not } b) = FV(b)$$

$$FV(b_1 \text{ or } b_2) = FV(b_1) \cup FV(b_2)$$

$$FV(b_1 \text{ and } b_2) = FV(b_1) \cup FV(b_2)$$

And finally for Statements:

$$FV(\text{skip}) = \emptyset$$

$$FV(x := e) = \{x\} \cup FV(e)$$

$$FV(s_1; s_2) = FV(s_1) \cup FV(s_2)$$

$$FV(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}) = FV(b) \cup FV(s_1) \cup FV(s_2)$$

$$FV(\text{while } b \text{ do } s \text{ end}) = FV(b) \cup FV(s)$$

4.1.3.3 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:

$$(e_1 \text{ op } e_2)[x \mapsto e] \equiv (e_1[x \mapsto e] \text{ op } e_2[x \mapsto e])$$

$$n[x \mapsto e] \equiv n$$

$$y[x \mapsto e] \equiv \begin{cases} e & \text{if } x \equiv y \\ y & \text{otherwise} \end{cases}$$

The same for boolean expressions:

$$(e_1 \text{ op } e_2)[x \mapsto e] \equiv (e_1[x \mapsto e] \text{ op } e_2[x \mapsto e])$$

$$(\text{not } b)[x \mapsto e] \equiv \text{not } (b[x \mapsto e])$$

$$(b_1 \text{ or } b_2)[x \mapsto e] \equiv (b_1[x \mapsto e] \text{ or } b_2[x \mapsto e])$$

$$(b_1 \text{ and } b_2)[x \mapsto e] \equiv (b_1[x \mapsto e] \text{ and } b_2[x \mapsto e])$$

Substitution Lemma

Lemma 4.1.1

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

4.2 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)

4.2.1 Big-Step Semantics

4.2.1.1 Transition Systems

Definition 4.2.1 (*Transition System*) is a tuple (Γ, T, \rightarrow) , where Γ is a set of **configurations**, T is a set of terminal configurations, with $T \subseteq \Gamma$ and \rightarrow is a transition relation, with $\rightarrow \subseteq \Gamma \times \Gamma$, which describes how executions take place. Big-step transitions are of form $\langle s, \sigma \rangle \rightarrow \sigma'$, e.g. $\langle \text{skip}, \sigma \rangle \rightarrow \sigma$

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

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

or spelled out, "If $\varphi_1, \dots, \varphi_n$ are transitions (and the *side-condition* is true), then ψ is a transition".

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

4.2.1.2 Big-Step Semantics of IMP

$$\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma} \text{SKIP}_{NS} \quad \frac{}{\langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[[e]]\sigma]} \text{ASS}_{NS}$$

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

$$\frac{\langle s, \sigma \rangle \rightarrow \sigma' \quad \langle s', \sigma' \rangle \rightarrow \sigma''}{\langle s; s', \sigma \rangle \rightarrow \sigma''} \text{SEQ}_{NS}$$

Conditional Statements if b then s else s' end (If b holds, execute s , otherwise execute s')

$$\frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow \sigma'} \text{IFT}_{NS} \quad \frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow \sigma'} \text{IFF}_{NS}$$

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

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

$$\frac{\langle s, \sigma \rangle \rightarrow \sigma' \quad \langle \text{while } b \text{ do } s \text{ end}, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma''} \text{WHT}_{NS} \quad \text{if } \mathcal{B}[[b]]\sigma = \text{tt}$$

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

$$\frac{}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma} \text{WHF}_{NS} \quad \text{if } \mathcal{B}[[b]]\sigma = \text{ff}$$

4.2.1.3 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

$$\frac{}{\langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[[e]]\sigma]} \text{ASS}_{NS} \quad \frac{}{\langle v := v + 1, \sigma_{\text{zero}} \rangle \rightarrow \sigma[v \mapsto \mathcal{A}[[v + 1]]\sigma_{\text{zero}}]} \text{ASS}_{NS}$$

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

4.2.1.4 Termination

Termination

Definition 4.2.2

The execution of a statement s in a state σ

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

4.2.1.5 Semantic equivalence

Definition 4.2.3 Two statements s_1 and s_2 are **semantically equivalent**, denoted $s_1 \simeq s_2$, if and only if

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

4.2.1.6 Unfolding loops

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

```

1  int i = 0;
2  while ( i < 2 ) {
3
4      while ( i < 1 )
5          if ( i == 0 ) break;
6
7      i++;
8  }
9  printf( "i = %d", i );
10
```

Prints **i = 2**

```

1  int i = 0;
2  while ( i < 2 ) {
3      if ( i == 0 ) {
4          if ( i == 0 ) break;
5          while ( i < 1 )
6              if ( i == 0 ) break;
7      }
8      i++;
9  }
10 printf( "i = %d", i );
```

Prints **i = 0**

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

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

So what we have to prove is this:

$$\forall b, s, \sigma, \sigma'. (\vdash \langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \iff \vdash \langle \text{if } b \text{ then } s; \text{ while } b \text{ do } s \text{ end end}, \sigma \rangle \rightarrow \sigma')$$

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)

4.2.1.7 Deterministic Semantics

Lemma 4.2.4 The big-step semantics of IMP is deterministic

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

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.

4.2.1.8 Extensions of IMP

4.2.1.8.1 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:

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

4.2.1.8.2 Procedure Declaration and Calls

procedure `p(x1, ..., xn; y1, ..., ym) begin s end`

Here, the `xi` are the **value** parameters and the `yj` 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 `xi` and `yj` must be pairwise distinct and the only free variables in `s`.

For the **procedure call** `p(e1, ..., en; z1, ..., zm)`, the **actual** variable parameters must be pairwise distinct.

The corresponding Big-step semantics rule is:

$$\frac{\langle s, \sigma_{\text{zero}}[\vec{x}_i \mapsto \mathcal{A}[\![e_i]\!]\sigma][\vec{y}_j \mapsto \overrightarrow{\sigma(z_j)}] \rangle \rightarrow \sigma'}{\langle p(\vec{e}_i; \vec{z}_j), \sigma \rangle \rightarrow \sigma[\vec{z}_j \mapsto \overrightarrow{\sigma'(y_j)}]} \text{CALL}_{NS}$$

where the notation $\sigma[\vec{x}_i \mapsto \vec{v}_i]$ is an abbreviation of $\sigma[x_1 \mapsto v_1] \dots [x_n \mapsto v_n]$

4.2.1.8.3 Non-determinism

For a statement `s □ s'`, either `s` or `s'` is non-deterministically chosen to be executed.

Example 4.2.5 In the statement `x := 1 □ (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:

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

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

4.2.1.8.4 Parallelism

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

Example 4.2.6 `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.

4.2.2 Small-Step semantics

4.2.2.1 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 \rightarrow_1 , which can have two forms:

- $\langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle$ is for any non-complete computation, where the next computation is expressed by the intermediate configuration $\langle s', \sigma' \rangle$
- $\langle s, \sigma \rangle \rightarrow_1 \sigma'$ is the final execution that reaches a terminal state.

Finally, a transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$ 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 $\langle s, \sigma \rangle \rightarrow_1 \gamma$.

4.2.2.1.1 The rules

$$\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow_1 \sigma} \text{SKIP}_{SOS} \quad \frac{}{\langle x := e, \sigma \rangle \rightarrow_1 \sigma[x \mapsto \mathcal{A}[e]\sigma]} \text{ASS}_{SOS}$$

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 (SEQ1), or does not (SEQ2).

$$\frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle s; s', \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{SEQ1}_{SOS} \quad \frac{\langle s', \sigma' \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s; s', \sigma \rangle \rightarrow_1 \langle s''; s', \sigma' \rangle} \text{SEQ2}_{SOS}$$

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

$$\frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{IFT}_{SOS} \quad \frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{IFF}_{SOS}$$

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

$$\frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \sigma'} \text{IFT1}_{SOS} \quad \frac{\langle s, \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s''', \sigma' \rangle} \text{IFT2}_{SOS}$$

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

$$\frac{}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow_1 \langle \text{if } b \text{ then } s; \text{ while } b \text{ do } s \text{ end else skip end}, \sigma \rangle} \text{WHILE}_{SOS}$$

4.2.2.2 Multi-Step Execution

We use the definitions we have to define a **k -step execution**, denoted $\gamma \rightarrow_1^k \gamma'$. 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 :

- $\gamma \rightarrow_1^0 \gamma'$ if and only if $\gamma = \gamma'$
- $\gamma \rightarrow_1^k \gamma'$ if and only if there exists γ'' such that both $\gamma \rightarrow_1$ and $\gamma'' \rightarrow_1^{k-1} \gamma'$

Resulting from this, $\gamma \rightarrow_1^{k_1+k_2} \gamma'$ if and only if $\exists \gamma'' . \gamma \rightarrow_1^{k_1} \gamma'' \wedge \gamma'' \rightarrow_1^{k_2} \gamma'$

We write $\gamma \rightarrow_1^* \gamma'$ to signify that there is an execution from γ to γ' in some finite number of steps, or more formally:

$$\exists k . \gamma \rightarrow_1^k \gamma'$$

4.2.2.3 Derivation Sequences

Definition 4.2.7 A **derivation sequence** is a non-empty sequence of configurations γ_0, \dots , for which $\gamma_i \rightarrow_1 \gamma_{i+1}$ 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!)

4.2.2.4 Termination

Theorem 4.2.8 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 4.2.9 The execution of statement s in state σ **terminates** successfully, if and only if $\exists \sigma'. \langle s, \sigma \rangle \rightarrow_1^* \sigma'$

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

4.2.2.5 Proving properties of Derivation Sequences

For reasoning about finite derivation sequences, we commonly reason about a multi-step execution $\gamma \rightarrow_1^k \gamma'$ 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

4.2.2.6 Semantic Equivalence and Determinism

Definition 4.2.10 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 $\langle s_1, \sigma \rangle \rightarrow_1^* \gamma$ if and only if $\langle s_2, \sigma \rangle \rightarrow_1^* \gamma$, and
- there is an 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 4.2.11 The small-step semantics of IMP is **deterministic**. That is:

$$\forall s, \sigma, \gamma, \gamma'. \vdash \langle s, \sigma \rangle \rightarrow_1 \gamma \wedge \vdash \langle s, \sigma \rangle \rightarrow_1 \gamma' \implies \gamma = \gamma'$$

Corollary 4.2.12 There is exactly one derivation sequence starting in configuration $\langle s, \sigma \rangle$

4.2.2.7 Extensions of IMP

4.2.2.7.1 Local Variable Declarations

As already partially established in Section 4.2.1.8.1, 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 the 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:

$$\frac{}{\langle \text{var } x := e \text{ in } s \text{ end}, \sigma \rangle \rightarrow_1 \langle s; \text{restore } (x, \sigma(x)), \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma] \rangle} \text{LOC}_{SOS}$$

$$\frac{}{\langle \text{restore } (x, \sigma(x)), \sigma \rangle \rightarrow_1 \sigma[x \mapsto v]} \text{RET}_{SOS}$$

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

4.2.2.7.2 Non-determinism

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

$$\frac{}{\langle s \sqcap s', \sigma \rangle \rightarrow_1 \langle s, \sigma \rangle} \text{ND1}_{SOS} \quad \frac{}{\langle s \sqcap s', \sigma \rangle \rightarrow_1 \langle s, \sigma' \rangle} \text{ND2}_{SOS}$$

4.2.2.7.3 Parallelism

$$\frac{\langle s, \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s'' \text{ par } s', \sigma' \rangle} \text{PAR1}_{SOS} \quad \frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{PAR2}_{SOS}$$

$$\frac{\langle s', \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s \text{ par } s'', \sigma' \rangle} \text{PAR3}_{SOS} \quad \frac{\langle s', \sigma \rangle \rightarrow_1 \sigma'}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s, \sigma' \rangle} \text{PAR4}_{SOS}$$

4.2.3 Equivalence

Equivalence Theorem

Theorem 4.2.13

For every statement s of IMP, we have

$$\vdash \langle s, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle s, \sigma \rangle \rightarrow_1^* \sigma'$$

If the execution of s from some state terminates successfully in one of the semantics, then 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 4.2.14 (*Equivalence Lemma 1*) For every statement s of IMP and states σ and σ' we have:

$$\vdash \langle s, \sigma \rangle \rightarrow \sigma' \implies \langle s, \sigma \rangle \rightarrow_1^* \sigma'$$

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

$$\langle s, \sigma \rangle \rightarrow_1^k \sigma' \implies \vdash \langle s, \sigma \rangle \rightarrow \sigma'$$

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!)

4.2.4 Operational Semantics Rules Overview

4.2.4.1 Big-Step Semantics

$$\begin{array}{c}
\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma} \text{SKIP}_{NS} \quad \frac{}{\langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[e]\sigma]} \text{ASS}_{NS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow \sigma' \quad \langle s', \sigma' \rangle \rightarrow \sigma''}{\langle s; s', \sigma \rangle \rightarrow \sigma''} \text{SEQ}_{NS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow \sigma'} \text{IFT}_{NS} \quad \frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow \sigma'} \text{IFF}_{NS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow \sigma' \quad \langle \text{while } b \text{ do } s \text{ end}, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma''} \text{WHT}_{NS} \quad \frac{}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma} \text{WHF}_{NS} \\
\text{if } \mathcal{B}[b]\sigma = \text{tt} \quad \text{if } \mathcal{B}[b]\sigma = \text{ff}
\end{array}$$

$$\frac{}{\langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[e]\sigma]} \text{ASS}_{NS} \quad \frac{}{\langle v := v + 1, \sigma_{\text{zero}} \rangle \rightarrow \sigma[v \mapsto \mathcal{A}[v + 1]\sigma_{\text{zero}}]} \text{ASS}_{NS}$$

4.2.4.2 Small-Step Semantics

$$\begin{array}{c}
\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow_1 \sigma} \text{SKIP}_{SOS} \quad \frac{}{\langle x := e, \sigma \rangle \rightarrow_1 \sigma[x \mapsto \mathcal{A}[e]\sigma]} \text{ASS}_{SOS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle s; s', \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{SEQ1}_{SOS} \quad \frac{\langle s', \sigma' \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s; s', \sigma \rangle \rightarrow_1 \langle s''; s', \sigma' \rangle} \text{SEQ2}_{SOS} \\
\\
\frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{IFT}_{SOS} \quad \frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{IFF}_{SOS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \sigma'} \text{IFT1}_{SOS} \quad \frac{\langle s, \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma \rangle \rightarrow_1 \langle s''', \sigma' \rangle} \text{IFT2}_{SOS} \\
\\
\frac{}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow_1 \langle \text{if } b \text{ then } s; \text{ while } b \text{ do } s \text{ end else skip end}, \sigma \rangle} \text{WHILE}_{SOS}
\end{array}$$

4.2.4.3 Extensions

4.2.4.3.1 Big-Step Semantics

$$\begin{array}{c}
\frac{\langle s, \sigma[x \mapsto \mathcal{A}[e]\sigma] \rangle \rightarrow \sigma'}{\langle \text{var } x := e \text{ in } s \text{ end}, \sigma \rangle \rightarrow \sigma'[x \mapsto \sigma(x)]} \text{LOC}_{NS} \quad \frac{\langle s, \sigma_{\text{zero}}[\vec{x}_i \mapsto \mathcal{A}[e_i]\sigma][\vec{y}_j \mapsto \sigma(\vec{z}_j)] \rangle \rightarrow \sigma'}{\langle p(\vec{e}_i; \vec{z}_j), \sigma \rangle \rightarrow \sigma[\vec{z}_j \mapsto \sigma'(\vec{y}_j)]} \text{CALL}_{NS} \\
\\
\frac{\langle s, \sigma \rangle \rightarrow \sigma'}{\langle s \square s', \sigma \rangle \rightarrow \sigma'} \text{ND1}_{NS} \quad \frac{\langle s', \sigma \rangle \rightarrow \sigma'}{\langle s \square s', \sigma \rangle \rightarrow \sigma'} \text{ND2}_{NS}
\end{array}$$

4.2.4.3.2 Small-Step Semantics

$$\begin{array}{c}
\frac{}{\langle \text{var } x := e \text{ in } s \text{ end}, \sigma \rangle \rightarrow_1 \langle s; \text{restore } (x, \sigma(x)), \sigma[x \mapsto \mathcal{A}[e]\sigma] \rangle} \text{LOC}_{SOS} \\
\frac{}{\langle \text{restore } (x, \sigma(x)), \sigma \rangle \rightarrow_1 \sigma[x \mapsto v]} \text{RET}_{SOS} \\
\frac{}{\langle s \square s', \sigma \rangle \rightarrow_1 \langle s, \sigma \rangle} \text{ND1}_{SOS} \quad \frac{}{\langle s \square s', \sigma \rangle \rightarrow_1 \langle s, \sigma' \rangle} \text{ND2}_{SOS} \\
\frac{\langle s, \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s'' \text{ par } s', \sigma' \rangle} \text{PAR1}_{SOS} \quad \frac{\langle s, \sigma \rangle \rightarrow_1 \sigma'}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle} \text{PAR2}_{SOS} \\
\frac{\langle s', \sigma \rangle \rightarrow_1 \langle s'', \sigma' \rangle}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s \text{ par } s'', \sigma' \rangle} \text{PAR3}_{SOS} \quad \frac{\langle s', \sigma \rangle \rightarrow_1 \sigma'}{\langle s \text{ par } s', \sigma \rangle \rightarrow_1 \langle s, \sigma' \rangle} \text{PAR4}_{SOS}
\end{array}$$

4.3 Axiomatic Semantics

4.3.1 Program Correctness

Definition 4.3.1 (*Partial Correctness*) if a program terminates *then* there will be a certain relationship between the initial and final state

Definition 4.3.2 (*Total Correctness*) Program terminates *and* is partially correct, i.e.
total correctness = partial correctness + termination

Definition 4.3.3 (*Formal Specification*) is used to express 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 4.3.4 Consider the factorial statement

```

1 y := 1;
2 while not x = 1 do
3     y := y * x;
4     x := x - 1
5 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:

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

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.

4.3.2 Hoare Logic

4.3.2.1 Hoare Triples

Definition 4.3.5 Properties are specified as **Hoare Triples** $\{P\} s \{Q\}$, with statement s , P the precondition and Q the postcondition. Here, P, Q are assertions and are, together with R , meta-variables over assertions.

Definition 4.3.6 (*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:

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

We use $P_1 \wedge P_2$ instead of $P_1 \text{ and } P_2$, $P_1 \vee P_2$ instead of $P_1 \text{ or } P_2$ and $\neg P$ instead of **not** P

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

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

4.3.2.2.1 The rules

$$\frac{}{\{P\} \text{ skip } \{P\}} \text{SKIP}_{Ax} \quad \frac{}{\{P[x \mapsto e]\} x := e \{P\}} \text{ASS}_{Ax}$$

Sequential Composition $s; s'$, loop (**while** b **do** s **end**) and conditional statements (**if** b **then** s_1 **else** s_2 **end**)

$$\frac{\{P\} s \{Q\} \quad \{Q\} s' \{R\}}{\{P\} s; s' \{P\}} \text{SEQ}_{Ax} \quad \frac{\{b \wedge P\} s \{Q\} \quad \{\neg b \wedge P\} s' \{Q\}}{\{P\} \text{ if } b \text{ then } s \text{ else } s' \text{ end } \{Q\}} \text{IF}_{Ax}$$

$$\frac{\{b \wedge P\} s \{P\}}{\{P\} \text{ while } b \text{ do } s \text{ end } \{\neg b \wedge P\}} \text{WH}_{Ax}$$

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

Definition 4.3.7 (*Semantic entailment*) expresses the reasoning steps “We write $P \models Q$ if and only if for all states σ , $\mathcal{B}[\![P]] \sigma = \text{tt}$ implies $\mathcal{B}[\![Q]] \sigma = \text{tt}$ ”

This leads to the **Rule of Consequence**

$$\frac{\{P'\} s \{Q'\}}{\{P\} s \{Q\}} \text{CONS}_{Ax} \quad \text{if } P \models P' \text{ and } Q' \models Q$$

a rule, where we can **strengthen preconditions** (P cannot be weaker than P') and **weaken postconditions** (Q cannot be stronger than 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 SKIP_{Ax} , instead of writing $\vdash \{P\} \text{ skip } \{P\}$, we can write. More examples on slides 167 - 170 (pages 30 - 33 in Slide Deck 4)

$$\begin{array}{c} \{P\} \\ \text{skip} \\ \{P\} \end{array}$$

Forward Assignment

$$\frac{}{\{P\} x := e \{ \exists V. P[x \mapsto V] \wedge x = e[x \mapsto V] \}} \text{ASSF}_{Ax}$$

4.3.2.3 Total Correctness

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

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

$$\frac{\{b \wedge P \wedge e = Z\} s \Downarrow P \wedge e < Z}{\{P\} \text{ while } b \text{ do } s \text{ end } \Downarrow \neg b \wedge P} \text{WHTOT}_{Ax} \quad \text{if } b \wedge P \models 0 \leq e$$

4.3.3 Soundness and Completeness

Definition 4.3.8 (*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 4.3.9 (*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 4.3.10 The partial correctness triple $\{P\} s \{Q\}$ is valid, written $\models \{P\} s \{Q\}$ if and only if

$$\forall \sigma, \sigma'. \mathcal{B}[\![P]\!]\sigma = \text{tt} \wedge \vdash \langle s, \sigma \rangle \rightarrow \sigma' \implies \mathcal{B}[\![Q]\!]\sigma' = \text{tt}$$

More generally:

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

Soundness, Completeness

Theorem 4.3.11

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

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

5 Modelling

5.1 Model Checking

Model Checking

Definition 5.1.1

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.

5.1.1 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.

5.2 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

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

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

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

```

1 // --- Constants -----
2 // As C preprocessor macros
3 #define N 5
4
5 // Symbolic constant
6 mtype = { ack, req };
7
8
9 // --- Typedef -----
10 // Structure declarations
11 typedef vector { int x; int y };
12
13
14 // --- Channels -----
15 // Channels are used to exchange messages between processes
16 chan buf = [2] of { int }; // can store up to 2 integers
17 chan buf2 = [2] of { mtype, bit, chan }; // Messages are Triples
18 chan channel = [0] of { int }; // No buffer
19 chan unassigned_chan; // This channel is unassigned
20
21
22 // --- Variables -----
23 // Note that there are no floats and unbounded integers.
24 // Variables are initialized to 0 (ish) values
25 // If defined outside a process, they are global, if inside,
26 // they are scoped to said process
27 bit bit_val; // 1 bit
28 bool bool_val; // Equivalent to bit, also 1 bit
29 byte counter; // 1 byte (0...255)
30 short short_val; // 2 bytes ( $-2^{15} \dots 2^{15} - 1$ )
31 int val; // 4 bytes
32 int arr[4]; // 16 bytes, array of four integers
33 vector v; // using the custom vector type
34 mtype msg = ack; // Using the symbolic constant
35
36
37 // --- Processes -----
38 // Process declarations
39 proctype myProc(int p) {
40     // Like the init body, any promela statements go in here
41     // This includes any variable or channel declarations
42     printf("My process");
43 }
44
45 active [N] proctype myActiveProc(int p) {
46     // Body goes here
47 }
48
49
50 // --- Initial state -----
51 init {
52     printf("Hello World");
53 }

```

5.2.1 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 $E1 \rightarrow E2 : E3$

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

- | | | | | |
|------------------------|-------------------------|---------------------------------|--------------------------|--------------------------|
| ▪ <code>len()</code> | ▪ <code>nempty()</code> | ▪ <code>nfull()</code> | ▪ <code>eval()</code> | ▪ <code>pcvalue()</code> |
| ▪ <code>empty()</code> | ▪ <code>full()</code> | ▪ <code>run <proc></code> | ▪ <code>enabled()</code> | |

5.2.2 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.

5.2.3 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.