Formal Methods and Functional Programming

Paul He

July 18, 2024

Contents

Part I

Functional Programming

1 *Basic Concepts in Functional Programming*

There are two main concepts of functions and values:

- Functions compute values.
- Functions are values: can compute and return them. This case is a lot less common in other programming languages but present in functional languages like Haskell.

Another really important concept is that functions do not have side effects: $f(x)$ will always return the same value. This is not the case in all programming languages. Consider the following code[1]

[1] y is a class variable: shared by all objects.

```
class Test {
    static int y = 0;
    static int f(int x) {
        y = y + 1;
        return y;
    }

    public static void main(String[] args) {
        System.out.println(f(0));
        System.out.println(f(0));
    }
}
```

Notice that $f(0) \neq f(0)$ in the above program. "This is a horrible thing" - Professor David Basin. In functional languages such as Haskell, we have no side effects, which allows us to reason as in mathematics. For example, if $f(0) = 2$, then in **every** possible context, $f(0) + f(0) = 2 + 2 = 4$. This property is called **referential transparency**: an expression that evaluates to the same value in every context. This means we have no assignments, and no global variables, allowing us to reason about programs without considering state. This also allows us to easily parallelize as computations cannot interfere.

We also use recursion instead of iteration, for example, instead of

```
public static int gcd (int x, int y) {
    while (x != y) {
        if (x > y) x = x - y;
        else y = y - x;
    }
}
```

we do recursion

```
gcd x y
 | x == y    = x
 | x > y     = gcd (x - y) y
 | otherwise = gcd x (y-x)
```

Functional programs also allow a flexible type system, which avoids many kinds of programming errors. For example, no runtime errors like `3 + True`. It also allows Polymorphism that supports reusability.

```
sort [5, 3, 4]
sort ["hello", "there", "world"]
```

1.1 Expression Evaluation

In mathematics, e.g., $f(x,y) = x - y$. We can compute $f(5,7)$ by substituting 5 for x and 7 for y and continue evaluation $f(5,7) = 5 - 7 = -2$. The same holds for Haskell:

```
gcd 10 15 = gcd 10 (15 - 10)
          = gcd 10 5
          = gcd (10 - 5) 5
          = gcd 5 5
          = 5
```

It turns out there are different evaluation strategies. Consider the program `diff x y = x - y` **Eager Evaluation**: Evaluate arguments first, also called by "call-by-value". Then, with this strategy we would get

```
diff (1 + 2) (3 + 4) = diff 3 (3 + 4)
```

```
                    = diff 3 (3 + 4)
                    = diff 3 7
                    = 3 - 7
                    = -4
```

However, this is not how Haskell evaluates its programs. **Lazy Evaluation:** This is used in Haskell, also called as "call by need" or "left-most/outermost". Certain functions force evaluation, e.g., arithmetic. Basically, we evaluate expressions only when needed.

```
diff (1 + 2) (3 + 4) = (1 + 2) - (3 + 4)
                     = (1 + 2) - 7
                     = 3 - 7
                     = -4
```

1.2 Syntax

1. functions and arguments start with lower-case letter
2. arguments written in sequence
3. and separated by whitespace

```
gcd x y      -- 1
 | x == y    = x
 | x > y     = gcd (x - y) y     -- 2
 | otherwise = gcd x (y-x)       -- 3
```

Functions consist of different cases:

```
functionName x1 ... xn
 | guard1 = expr1
   .
   .
 | guardm = exprm
```

Program consists of several definitions

```
myConstant = 5
aFunction y1 ... ym
 | guard1 = expr1
 | guard2 = expr2
```

```
anotherFunction z1 ... zk = ...
```

Notice that we use a space for indentation, not TABS. Patterns `y1 ... ym` are variables, constants, or built from data constructors (like tuples). Guards are boolean expressions. Pattern matching forces evaluation.

1.3 Types

Haskell is a strongly typed language, as mentioned before they avoid runtime errors. Either programmer provides types along with function definition.

```
gcd :: Int -> Int -> Int
```

or the system computes types itself. Function / argument types must "match" (formally later).

Type Tuple. A type constructor takes type and builds a type. For example, a student has name, ID number, starting year. Type `(String, Int, Int)` with element `("Bob A", 1234, 2015)`.

- If $T_1, \ldots, T_n$ are types, then $(T_1, \ldots, T_n)$ is a (tuple) type. e.g. `(Int, String, Bool)`.
- If $v_1 :: T_1, \ldots v_n :: T_n$ then $(v_1, \ldots v_n) :: (T_1, \ldots T_n)$. Essentially reads, if variables v_1 to v_n have type T_1 to T_n, then the tuple $(v_1, \ldots v_n)$ we construct has type $(T_1, \ldots, T_n)$. For example `(3, "hi", True) :: (Int, String, Bool)`.

Note that $n \geq 2$, so `("foo")` is not a tuple. We can also nest tuples:

- `(3, ("hi", True)) :: (Int, (String, Bool))`

Functions can take tuples as arguments or return tupled values

```
addPair :: (Int, Int) -> Int
addPair (x, y) = x + y
? addPair (3, 4)
7
```

Patterns can also be nested

```
shift :: ((Int, Int), Int) -> (Int, (Int, Int))
shift ((x, y), z) = (x, (y, z))
```

Pattern matching can be used to decompose tuples

```
name (s, id, x) = s
studentNumber (s, id, y) = id
year (s, id, y) = y
```

1.4 Scope

Global scope: A function can be called from any other

```
f x y = ...
g x = ... h ...
h z = ... f ... g ...
```

Local scope with `let` and `where`

```
let x1 = e1
    :
    xn = en
in e
```

`let` builds on expression from others

- `xi` can bind a variable or a local function
- local definitions may refer to each other

```
f p1 p2 ... pm
 | g1 = e1
 | g2 = e2
   :
 | gk = ek
where
 v1 a1 ... an = r1
 v2 = r2
 :
```

2 *Natural Deduction*

To carry out formal reasoning we need three essential parts:

1. Language
2. Semantics
3. Deductive system for carrying out proofs

Natural deduction is a method for proofs. It consists of a set of rules which are used to construct a derivation tree. Finally, a proof is a derivation tree whose root has no assumptions.

2.1 *Propositional Logic: Syntax*

Propositions are built from a collections of variables and closed under disjunctions, conjunction and implication, etc. More formally, let $\mathcal{V}$ be a given set of variables. $\mathcal{L}_P$, the **language of propositional logic**, 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 $B \in \mathcal{L}_P$

2.2 *Propositional Logic: Semantics*

- A **valuation** $\sigma : \mathcal{V} \to \{\mathsf{True}, \mathsf{False}\}$ is a function mapping variables to truth values (truth assignment). Valuations are simple kinds of models (interpretations). Let **Valuations** be a set of valuations.
- **Satisfiability** is the smallest relation $\models \subseteq \mathsf{Valuations} \times \mathcal{L}_P$ such that
  - $\sigma \models X$ if $\sigma(X) = \mathsf{True}$
  - $\sigma \models A \wedge B$ 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 $\sigma \models B$
  - $\sigma \not\models \bot$ for every $\sigma \in \mathsf{Valuations}$ (No valuation satisfies contradiction)
- A formula $A \in \mathcal{L}_P$ is **satisfiable** if $\sigma \models A$ for some valuation σ

- A formula $A \in \mathcal{L}_P$ is **valid** (a tautology) if $\sigma \models A$ for all valuations σ
- **Semantic Entailment**: $A_1, \ldots, A_n \models A$ if for all σ, if $\sigma \models A_1, \ldots, \sigma \models A_n$ then $\sigma \models A$

For example,

- $X \wedge Y$ is satisfiable, a solution simply sets X and Y to true: $\sigma \models X \wedge Y$ for $\sigma(X) = \sigma(Y) = \mathsf{True}$.
- $X \to X$ is valid (a tautology): $\sigma(X) = \mathsf{True}$ means $\mathsf{True} \to \mathsf{True}$, if $\sigma(X) = \mathsf{False}$ then the entire statement is vacuously true.
- $\neg X, X \vee Y \models Y$. Assume it holds as $\sigma \models \neg X$ and $\sigma \models X \vee Y$ constrain $\sigma(X) = \mathsf{False}$ and $\sigma(Y) = \mathsf{True}$, so $\sigma \models Y$.

2.3 Requirements for a Deductive System

Syntactic entailment $\vdash$ (derivation rules) and semantic entailment $\models$ (truth tables) should agree. This requirement has two parts:

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

for $H \equiv A_1, \ldots, A_n$ some collection of formulae. These are the key requirements for any logic. **Decidability** is also desirable. What is the complexity of determining:

- If a proposition A is satisfied by a valuation of σ? Linear.
- If A is satisfiable? NP.
- If A is a tautology? Exponential (Complement of NP).

2.4 Natural Deduction for Propositional Formulae

A **sequent** is an assertion (judgement) of the form $A_1, \ldots, A_n \vdash A$ where $A, A_1, \ldots, A_n$ are all propositional formulae. Intuitively, A follows from the A_is. If a deductive system is sound, this means that the A_is semantically entail A. An axiom is a starting point for building derivation trees.

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

A proof of A is a derivation tree with root $\vdash$ A. If deductive system is sound, then A is a tautology. Let's introduce some proof rules.

Conjunction: There are two kinds of rules, **introduce** and **eliminate** connectives.

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

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

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

Here is an example derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash X \wedge (Y \wedge Z)}\text{ axiom}}{\Gamma \vdash X}\wedge\text{-EL} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash X \wedge (Y \wedge Z)}\text{ axiom}}{\Gamma \vdash Y \wedge Z}\wedge\text{-ER}}{\Gamma \vdash Z}\wedge\text{-ER}}{\underbrace{X \wedge (Y \wedge Z)}_{\equiv\Gamma} \vdash X \wedge Z}\wedge\text{-I}$$

Each rule is sound in that is preserves semantic entailment. E.g., for $\wedge$-I if $\Gamma \models A$ and $\Gamma \models B$ then $\Gamma \models A \wedge B$. If all rules preserve semantic entailment, logic is sound.

Implication

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

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

Application of $\to$-I turns last derivation (the example derivation above) into a proof.

Disjunction

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

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

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

Elimination rule formalizes proof by cases. For example: - When it rains, then I wear my jacket. - When it snows, then I wear my jacket. - It is raining or snowing. - Therefore, I wear my jacket.

Falsity

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\text{-E}$$

Negation define $\neg A$ as $A \to \bot$.

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

which is derived by

$$\frac{\dfrac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot} \to\text{-E}}{\Gamma \vdash B} \bot\text{-E}$$

Syntax (Cont.) Form, the formulae of first-order logic, is the smallest set where 1. $\bot \in$ Form, 2. $p^n(t_1, \ldots, t_n) \in$ Form if $p^n \in \mathcal{P}$ and $t_j \in$ Term, for all $1 \leq j \leq n$, 3. $A \circ B \in$ Form if $A \in$ Form, $B \in$ Form, and $\circ \in \{\wedge, \vee, \to\}$, and 4. $Qx.A \in$ Form if $A \in$ Form, $x \in \mathcal{V}$, and $Q \in \{\forall, \exists\}$ Each occurrence of each variable in a formula is **bound** or **free**. A variable occurrence x in a formula A is **bound** is x occurs within a sub-formula B of A of the form $\exists.B$ or $\forall x.B$, which is analogous from mathematics $x^2 + \int_c^d x \cdot y dy$.

2.5 *Binding and α-conversion*

Names of bound variables are irrelevant, they just encode the binding structure. We can rename **bound** variables at any time (α-conversion). Note that we must preserve the binding structure. For example

$$\forall x.\exists y.p(x, y) = \forall y.\exists x.p(y, x)$$

However, you cannot do something from $\exists z.\forall y.p(z, f(y))$ to $\exists y.\forall y.p(y, f(y))$ since this changes the binding structure. Also converting $p(x) \to \forall x.p(x)$ into $p(y) \to \forall y.p(y)$ will not work since x is a free variable on the left hand side of the implication. Hence, we cannot rename free variables.

Semantics (Cont.) A **structure** is a pair $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$ where $U_\mathcal{S}$ is an nonempty set, the **universe**, and $I_\mathcal{S}$ is a mapping where: 1. $I_\mathcal{S}(p^n)$ is an n-ary relation on $U_\mathcal{S}$, for $p^n \in \mathcal{P}$, and 2. $I_\mathcal{S}(f^n)$ is an n-ary (total) function on $U_\mathcal{S}$ for $f^n \in \mathcal{F}$ **As a shorthand, we write $p^\mathcal{S}$ for $I_\mathcal{S}(p)$ and $f^\mathcal{S}$ for $I_\mathcal{S}(f)$.**

An **interpretation** is a pair $\mathcal{I} = \langle \mathcal{S}, v \rangle$, where $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$ is a structure and $v : \mathcal{V} \to U_\mathcal{S}$ a valuation. The **value** of a term t under the interpretation $I = \langle \mathcal{S}, v \rangle$ is written as $\mathcal{I}(t)$ and defined by 1. $\mathcal{I}(x) = v(x)$, for $x \in \mathcal{V}$, and 2. $\mathcal{I}(f(t_1, \ldots, t_n)) = f^\mathcal{S}(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$

Satisfiability $\models \subseteq$ Interpretations $\times$ Form is the smallest relation satisfying $\langle \mathcal{S}, v \rangle \models p(t_1, \ldots, t_n)$ if $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in p^\mathcal{S} \ldots \langle \mathcal{S}, v \rangle \models \forall x.A$ if $\langle \mathcal{S}, v[x \mapsto a] \rangle \models A$ for all $a \in U_\mathcal{S}$ and $\langle \mathcal{S}, v \rangle \models \exists x.A$ if $\langle \mathcal{S}, v[x \mapsto$

$a]\rangle \models A$. Here $v[x \mapsto a]$ is the valuation v' identical to v except that $v'(x) = a$. So when $\langle \mathcal{S}, v\rangle \models A$ we say A is satisfied with respect to $\langle \mathcal{S}, v\rangle$ OR $\langle \mathcal{S}, v\rangle$ is a model of A. Note that A does not have free variables, satisfaction does not depend on the valuation v. We write $\mathcal{S} \models A$. When every suitable interpretation is a model, we write $\models A$ and say A is valid. A is satisfiable if there is at least one model for A (and contradictory otherwise).

For example: $\forall x.p(x, s(x))$ then a model can be (for all x, x is less than $x+1$) - $U_{\mathcal{S}} = \mathcal{N}$ - $p^{\mathcal{S}} = \{(m,n)|m,n \in U_{\mathcal{S}} \wedge m < n\}$ - $s^{\mathcal{S}}$ = the successor function on $U_{\mathcal{S}}$ for example $s^{\mathcal{S}}(x) = x+1$

2.6 Substitution

Replaces all occurrences of a free variable x in A with some term t. We write $A[x/t]$ to indicate that we substitute x by t in A. For example, $A \equiv \exists y.y \times x = x \times z$ Then, if we want to substitute $2-1$ for x we would get $A[x/2-1] \equiv \exists y.y \times (2-1) = (2-1) \times z$. You can even substitute a free variable for x, for example $A[x/z] \equiv \exists y.y \times z = z \times z$.

2.7 Natural Deduction for First-order Logic

Universal Quantification

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ \forall\text{-I*}$$

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash B}\ \forall\text{-E}$$

Side condition * x is not free in any assumption in Γ.

Existential quantification

$$\frac{\Gamma \vdash A[x/t]}{\Gamma \vdash \exists x.A}\ \exists\text{-I*}$$

$$\frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ \exists\text{-E*}$$

Side condition *: x is neither free in B nor free in Γ.

2.8 Equality

Equality is an equivalence relation

$$\frac{}{\Gamma \vdash t = t}\ \text{ref}$$

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

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

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

$$\frac{\Gamma \vdash t_1 = s_1 \qquad \dots \qquad \Gamma \vdash t_n = s_n}{\Gamma \vdash p(s_1, \dots, s_n)}\ \text{cong-1}$$

$$\frac{\Gamma \vdash t_1 = s_1 \qquad \dots \qquad \Gamma \vdash t_n = s_n \qquad \Gamma \vdash p(t_1, \dots, t_n)}{\Gamma \vdash p(s_1, \dots, s_n)}\ \text{cong-2}$$

Soundness: Equality on $U_{\mathcal{S}}$ is a congruence.

3 Correctness

Correctness is important! It checks what your program should do. What does this means?

- Termination: Important for many, but not all programs
- Functional behavior: Function should return the "correct" value, it can be defined by another (mathematically defined) function or an input-output relation.

Correctness is rarely obvious, so it must be proven!

3.1 Termination

If f is defined in terms of functions $g_1, \ldots, d_k$ where $g_i \neq f$, and each g_i terminates, then so does f. For example:

```
g x = x * x + 15
f x = (g x + x + 2) / 13
```

The problem comes with recursive function, for example when $g_i = f$. An example of this is `fac n`. We need to introduce a sufficient condition for termination: Arguments are smaller along a well-founded order on function's domain.

Well-Founded Relation

An order $>$ on a set $\mathcal{S}$ is **well-founded** iff there is no infinite decreasing chain $x_1 > x_2 > x_3 > \ldots$ for $x_i \in \mathcal{S}$.

Examples are: $>_{\mathcal{N}}$, counter examples: $>_{\mathcal{Z}}, >_{\mathcal{R}}$ where $>_{\mathcal{S}}$ indicates the domain $\mathcal{S}$, for example $>_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$

Constructing Well-Founded Relations

We can construct new well-founded relations from existing ones. Let R_1 and R_2 be binary relations on a set S. The composition of R_1 and R_2 is defined as

$$R_1 \circ R_1 \equiv \{(a, c) \in S \times S | \exists b \in S. a R_1 b \wedge b R_2 c\}$$

Let $R \subseteq S \times S$, we define

$$R^1 \equiv R$$
$$R^{n_1} \equiv R \circ R^n \text{ for } n \geq 1$$
$$R^+ \equiv \bigcup_{n \geq 1} R^n$$

So aR^+b iff aR^ib for some $i \geq 1$

Lemma: Let $R \subseteq S \times S$. Let $s_0, s_i \in S$ and $i \geq 1$. Then $s_0 R^i s_i$ iff there are $s_1, \ldots, s_{i-1} \in S$ such that $s_0 R s_1 R \ldots R s_{i-1} R s_i$.

Theorem: If $>$ is a well-founded order on the set S, then $>^+$ is also well-founded on S.

Proof. For the sake of contradiction, assume that $a_1 >^+ a_2 >^+ a_3 >^+ \ldots$ is an infinite descending chain. Then, there exists $i_j \geq 1$ such that $a_j >^{i_j} a_{j+1}$, for all $j \geq 1$. By the above lemma, the sequence $a_1 >^{i_1} a_2 >^{i_2} a_3 >^{i_3} \ldots$ contradicts the well-foundedness of $>$. □

3.2 *Correctness*

Equational Reasoning Proofs based on a simple idea: Functions are equations! Consider the following simple Haskell Program:

```
swap :: (Int, Int) -> (Int, Int)
swap (a, b) = (b, a)
```

Meaning, for all possible values of a and b, $\texttt{swap}(a, b) = (b, a)$. More formally,

$$\forall a \in \mathbb{Z}, \forall b \in \mathbb{Z}, \texttt{swap}(a, b) = (b, a)$$

Some properties can be shown through equational reasoning. More generally, they become proofs in first-order logic with equality.

We can do a proof by cases. Consider the following example

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

Proof. We can prove that `maxi` n $m \geq n$. We have $n \geq m \vee \neg(n \geq m)$. Now we show `maxi` n $m \geq n$ for both cases

- Case when $n \geq m$: `maxi` n $m = n$ and $n \geq n$
- Case when $\neg(n \geq m)$: `maxi` n $m = m$, but $m > n$ so `maxi` n $m \geq n$

□

Proof by Induction How to prove a formula P (with free variable n), for all $n \in \mathbb{N}$? Proof by cases is not possible here! Base case: Prove $P[n/0]$, Step Case: For an arbitrary m not free in P, prove $P[n/m+1]$ under the assumption $P[n/m]$.

Well-founded Induction Also known as strong induction. To prove P for all natural numbers n: **Well-founded step**: For an arbitrary m (not free in P), prove $P[n/m]$ under the assumption that $P[n/l]$ holds, for all $l < m$ (where also l is not free in P). In general, we can use any well-founded ordering $<$. Here, the transitive closure of the relation on $\mathbb{N}$. Same principle applies to any set, not just $\mathbb{N}$.

4 Lists and Abstraction

4.1 List Type

List types are a new type constructor, if T is a type, then $[T]$ is a type. The elements of $[T]$ are:

- Empty List $[] :: [T]$
- Non-empty list $(x : xs) :: [T]$, if $x :: T$ and $xs :: [T]$

So $[1,2,3]$ is short hand for $1 : (2 : (3 : []))$. Lists are inductive data types.

Functions on List Note that when we want to write a function on lists, we will always need to define the case of the **empty list**.

- How to compute with the **empty list** $[]$
- How to compute with the non-empty list $(x : xs)$

For example: a function `sumList :: [Int] -> Int` must specify:

- Empty list $[] \mapsto 0$
- Non-empty list $(x : xs) \mapsto x$ + sum of lists xs

Translated to Haskell

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Patterns (lists and in general) Pattern matching has two purposes. It checks if an argument has the proper form, and it also binds values to variables. For example: $(x : xs)$ matches with $[2,3,4]$ $(2 : 3 : 4 : [] = \underbrace{2}_{x} : \underbrace{[3,4]}_{xs})$.

Patterns are inductively defined

- Constants: -2, '1', True, $[]$
- Variables: x, foo
- Wild Card: _
- Tuples: $(p_1, p_2, \ldots, p_k)$, where p_i are patterns
- Non-empty lists: $(p_1 : p_2)$, where p_i are patterns. It succeeds if a is a nonempty list $a_1 : a_2$ and p_1 matches a_1 and p_2 matches a_2

Patterns are required to be linear. This means that each variable can occur at most once! Hence, $[x, x]$ is not a valid pattern.

Zipper Function Extra elements in the longer list are discarded!

$$zip[2,3,4][4,5,78] = [(2,4),(3,5),(4,78)]$$

$$zip[2,3][1,2,3] = [(2,1),(3,2)]$$

In Haskell:

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

List Comprehension Notation for sequential processing of list elements, analogous to set comprehension in set theory $\{2 \cdot x | x \in X\}$. In Haskell

```
[2 * x | x <- xs]
```

Just like Python, they can be augmented with guards

```
[2 * x | x <- xs, pred1(x), ...]
```

Induction over lists To use induction, we use the following rule. **Proof by induction**: to prove P for all xs in $[T]$

- **Base case:** Prove $P[xs/[]]$
- **Step Case:** Prove $\forall y :: T, ys :: [T].P[xs/ys] \rightarrow P[xs/y : ys]$, i.e.,
  - **Fix arbitrary** $y :: T$ and $ys :: [T]$ (both not free in P)
  - **Induction Hypothesis**: $P[xs/ys]$
  - **Prove**: $P[xs/y : ys]$

4.2 Abstraction

Until now, we have only seen simple structuring techniques. We will now examine different ways of structuring, simplifying programs and improving their reusability.

Polymorphic Consider the following example

```
length []     = 0
length (x:xs) = 1 + length xs
```

What is the type? `[Int] -> Int, [String] -> Int, ...` The type is **polymorphic**: `[t] -> Int` **for all types** `t`. This is often called parametric polymorphism. A function is is typeable for all instances.

Definition A type w for f is a **most general** (also called **principle**) type iff for all types s for f, s is an instance of w.

Higher-order Functions **First order**: Arguments are base types or constructor types.

```
Int -> [Int]
```

Second order: Arguments are themselves functions

```
(Int -> Int) -> [Int]
```

Third Order: Arguments are functions, whose arguments are functions

```
((Int -> Int) -> Int) -> [Int]
```

Higher-order functions: Functions of arbitrary order
An example is the `map` function.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs)  = f x : map f xs
```

And lets consider a function `times2 x = 2 * x`, we can then do `map times2 [2, 3]` to multiply `2` to each element of the list. Consider another example:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Basically replace all of the : with a function f. So $fz[1,2,3] = f(1, f(2, f(3, z))$.

λ-expressions Anonymous functions in Haskell syntax: e.g. `\x -> 2 * x`.

Function Composition and Iteration

- Function Composition and Iteration

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

- Function Iteration

```
iter :: Int -> (a -> a) -> a -> a
iter 0 f x = x
iter n f x = f (iter (n-1) f x)
```

Functions as Values Functions can also be returned as values, for example:

```
twice :: (t -> t) -> (t -> t)
twice f = f . f
```

So if we run `twice times 3 1` its basically `times 3 (times 3 1) = 9`, similarly `(twice . twice) times3 1` = `times 3 (times 3 (times 3 (times 3, 1)))`. Now, we can also combine function iteration with function as values, such as[2]

[2] `f = iter 2 times2`, we want to compute `f 5`. Don't confuse the `f` inside the function definition of `iter` with it! In the function definition of `iter` it refers to `times2`!

```
let f = iter 2 times2 in f 5
 = (iter 2 times2) 5
 = (times 2 . (iter (2-1) times2)) 5
 = times 2 ((iter (2-1) times2) 5)
 = 2 * ((iter (2-1) times2) 5)
 = 2 * ((iter 1 times2) 5)
 = 2 * (times2 . (iter (1-1) times2) 5)
 = 2 * (2 * (iter (1-1) times2) 5)
 = 2 * (2 * (iter (0) times2) 5)
 = 2 * (2 * id 5)
 = 2 * (2 * 5)
 = 2 * 10
 = 20
```

Difference List : Concatenating lists naively takes $\mathcal{O}(n^2)$. This is bad! We can improve this to $\mathcal{O}(n)$ with difference lists. A difference list is a function `[a] -> [a]` **that prepends a list to its argument**. In func-

tional programming we can stick something to the end of the list in constant time.

```
type DList a = [a] -> [a]

empty :: DList a
empty = \xs -> xs

sngl :: a -> DList a
sngl x = \xs -> x : xs

-- concat (higher-order)
app :: DList a -> DList a -> DList a
ys 'app' zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]
toList ys = ys []
```

Partial Application Functions of multiple arguments can be partially applied! For example:

```
multiply :: Int -> Int -> Int
multiply a b = a * b
```

can be partially applied as `multiply 7` which has the type `Int -> Int`. We have the following lemma.

If $f :: t_1 \to t_2 \to \cdots \to t_n \to t$ and $e_1 :: t_1, \ldots, e_k :: t_k$ then $fe_1, \ldots e_k :: t_{k+1} \to \cdots \to t_n \to t$

So how many arguments do functions have? Each function only takes exactly one argument. Using the `multiply` example, `multiply 2 3` really means `(multiply 2) 3`. Partial application is consistent with the view (= illusion) that functions take multiple arguments. **Operator sections**: if $\oplus$ is an infix binary operator:

$$(a\oplus) \equiv \lambda x.a \oplus x$$

$$(\oplus a) \equiv \lambda x.x \oplus a$$

For example, running `map ((2*) . (3*)) [1, 2, 3]` returns `[6, 12, 18]`

Multiple Arguments versus Tupling Tuple arguments: no partial application. But with **currying** we can achieve something similar:

```
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
curry f   = f' where f' x1 x2 = f (x1, x2)
uncurry f'= f where f (x1,x2) = f' x1 x2
```

5 Higher-order Programming and Types

5.1 Review of higher-order functions

First-order functions take arguments that are base types. Higher-order functions have functions that are arguments. Consider the function `mystery x = x`. This function can be both first and higher-order. In addition to `map` and `fold`, we introduce a `filter` function.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
| p x          = x : filter p xs
| otherwise    = filter p xs
```

Map: Iteratively apply a function to each element

```
? map (2*) [1 .. 5]
[2, 4, 6, 8, 10] :: [Int]

? map (>2) [1 .. 5]
[False, False, True, True, True] :: [Bool]
```

Filter: Selection

```
? filter (>2) [1 .. 5]
[3, 4, 5] :: [Int]

? map (2>) [1 .. 5]
[1]:: [Int]
```

Fold: Use function to "combine" elements

```
? foldr (+) 0 [1 .. 5]
15 :: int
```

We can define new functions with filter, for example we can define `remove p = filter (not .  p)`. We can also partition lists as `part p`

`xs = (filter p xs, remove p xs)`. However the below for partition is more efficient (runtime)

```
partition p [] = ([], [])
partition p (x:xs)
 | p x          = (x:yesses, nos)
 | otherwise    = (yesses, x:nos)
 where (yesses, nos) = partition p xs
```

Map and filter can also be implemented using list comprehension.

```
map f xs    = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
```

Conversely, we can implement list comprehension with map and filter. `let fun p = expr in map fun s` defines `[expr | p <- s]`. **foldr** is right-associative fold

$$\textsc{foldr}(\oplus)e[l_1, l_2, \ldots, l_n] = l_1 \oplus (l_2 \oplus \cdots \oplus (l_n \oplus e))$$

```
foldr         :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

foldl is left-associative fold

$$\textsc{foldl}(\oplus)e[l_1, l_2, \ldots, l_n] = ((e \oplus l_1) \oplus l_2) \oplus \cdots \oplus l_n$$

```
foldl         :: (b -> a -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = foldl f (f e x) xs
```

For associative functions, and e is neutral element, there is no difference

```
? foldl (+) 0 [1, 2, 3]   -- ((0 + 1) + 2) + 3
6 :: Int
```

```
? foldr (+) 0 [1, 2, 3]   -- 1 + (2 + (3 + 0))
6 :: Int
```

However, not all (binary) functions are associative, it would not be equivalent for $-$. We can implement `length` using `foldr` as `length xs = foldr (_ y -> 1 + y) 0 xs`

Case Study: Operations on Vectors and Matrices

```
type Vector = [Int]

vecAdd :: Vector -> Vector -> Vector
vecAdd (x:xs) (y:ys) = (x + y) :: vecAdd xs ys
vecAdd _       _     = []
```

We can replace recursion with `map` and `zip`

```
vecAdd v1 v2 = map (uncurry (+)) (zip v1 v2)
```

A common pattern is combining `zip` and binary functions. `zipWith = map + zip`

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

We can represent matrices column-wise using lists. Hence, `type Matrix = [Vector]`. So we get

```
matAdd :: Matrix -> Matrix -> Matrix
matAdd = zipWith vecAdd
```

(each column is a list, add column with column). Now if we wanna do matrix multiplication, then we need to go through rows and columns. First, we define a few things. First with a constant vector of size n:

```
vconst :: Int -> Int -> Vector
vconst 0 _ = []
vconst n x = x : vconst (n - 1) x
```

and the unit matrix:

```
unit :: Int -> Matrix
unit 0 = []
unit n =
(1 : vconst (n - 1) 0) : map (0:) (unit (n - 1))
```

transposing a matrix

```
tr :: Matrix -> Matrix
tr []     = []
tr [v]    = map (\x -> [x]) v
tr (v:vs) = zipWith (:) v (tr vs)
```

dot product of two vectors

```
skProd :: Vector -> Vector -> Int
skProd (x:xs) (y:ys) = x*y + skProd xs ys
skProd _      _      = 0
```

OR

```
skProd :: Vector -> Vector -> Int
skProd v w = sum (zipWith (*) v w)
```

now we can define vector matrix multiplication as

```
vecMult :: Matrix -> Vector -> Vector
vecMult a b = map ('skProd' b) (tr a)
```

and matrix multiplication as

```
matMult :: Matrix -> Matrix -> Matrix
matMult a b = map (vecMult a) b
```

5.2 *Types*

Type checking should prevent "dangerous expressions". Dangerous expressions mean runtime error. Some objectives for a type check

- quick, decidable, static analysis
- permit as much generality / re-usability as possible
- prevent runtime errors: subject reduction

We say if $e \hookrightarrow e'$ and $\vdash e :: \tau$, then $e' :: \tau$. We examine a simplified language: 'Mini Haskell'.

Typing Types ($\mathcal{V}_\mathcal{T}$ is a set of type variables: $a, b, \dots$)

$$\tau ::= \mathcal{V}_\mathcal{T} \Big| \textsc{Bool} \Big| \textsc{Int}(\tau, \tau) \Big| (\tau \to \tau)$$

Type system notation based on typing judgement: $\Gamma \vdash t :: \tau$

- Γ is a set of bindings $x_i : \tau_i$ mapping variables to types. Intuitively, Γ represents a kind of typing "symbol table".
- t is a term.
- τ is a type.

The intuition is (not proof rules): given a symbol table Γ, then term t has type τ. For example $x : \textsc{Int} \vdash x + 2 :: \textsc{Int}$ but $x : \textsc{Int}, f : \textsc{Bool} \to \textsc{Bool} \nvdash fx :: \textsc{Bool}$.

Rules for core λ-calculus **Axiom**:

$$\frac{}{\dots, x : \tau, \cdots \vdash x :: \tau}\ \text{Var}$$

Abstraction ($x \notin \Gamma$)

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x.t) :: \sigma \to \tau}\ \text{Abs}$$

Application:

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau}\ \text{App}$$

Typing Rules for Mini-Haskell **Base Types**

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

Operations (**op** $\in \{+, \times\}$):

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

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

Tuples:

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

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

Type Inference Syntax-directed typing rules specify algorithm for computing type.

1. Start with judgement $\Gamma t :: \tau_0$ with type variable τ_0.
2. Build derivation tree bottom-up by applying rules. Introduce fresh type variables and collect constraints if needed.
3. Solve constraints (unification) to get possible types.

Here is a simple example:

$$\dfrac{\dfrac{\dfrac{}{z : \tau_1 \vdash z :: \tau_0}\ \text{Var}}{\vdash \lambda z.z :: \tau_1 \to \tau}\ \text{Abs} \qquad \dfrac{\dfrac{}{x :: \tau_2 \vdash x :: \tau_3}\ \text{Var}}{\vdash \lambda x.x :: \tau_1}\ \text{Abs}}{\vdash (\lambda z.z)(\lambda x.x) :: \tau_0}\ \text{App}$$

Here, we have to know that $\tau_2 = \tau_3$ to applying the variable rule on the right, and we also know that $\tau_1 = \tau_2 \to \tau_3$ to apply the abstraction rule. So we know $\tau_0 = \tau_3 \to \tau_3$ (the most general type). Sometimes, terms are untypeable. Type inference fails to build inference tree or solve constraints.

Self-Application Self application means "applying a function f to itself". Self application $\lambda f.ff$ is not typeable. We can show it is not typeable with the proof

$$\dfrac{\dfrac{\dfrac{}{f : \tau_1 \vdash f :: \tau_3 \to \tau_2}\ \text{Var} \qquad \dfrac{}{f : \tau_1 \vdash f :: \tau_3}\ \text{Var}}{f : \tau_1 \vdash ff :: \tau_2}\ \text{App}}{\vdash \lambda f.ff :: \tau_0}\ \text{Abs}$$

So, to apply the abstraction rule we must know that $\tau_0 = \tau_1 \to \tau_2$ and to apply the variable rule we must know $\tau_1 = \tau_3 \to \tau_2$ and that $\tau_1 = \tau_3$.

Problem! $\tau_3 = \tau_3 \to \tau_2$ requires infinite function type $((\ldots(\cdots \to \tau_2) \to \tau_2) \to \tau_2)$, but Haskell types are finite. This means there are no solution to the constraints. Compare it with the natural numbers: $x = 1 + x$ has the only solution ∞, but $\infty \notin \mathbb{N}$.

Curry-Howard Isomorphism (not exam relevant) Type construction $'' \to ''$ corresponds to propositional logic $'' \to ''$. Atomic types correspond to propositional variables.

5.3 Type Classes

Polymorphism is restricted using class constraints

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual x y z = ( x == y) && (y == z)
```

`Eq a` specifies a type class constraint. It means that a must be a type that belongs to the `Eq` type class. The `Eq` type class provides an interface for types that can be compared for equality.

`=>`: This symbol separates the type class constraints from the actual type of the function. It can be read as "given" or "subject to". So, you can read this as "given that a is an instance of `Eq`".

A class defines a set of types. E.g., `Eq` is the equality class

- `Int` $\in$ `Eq`

```
? allEqual 3 (2+1) (1+2)
True :: Bool
```

- `int` $\to$ `int` $\notin$ `Eq`

```
? allEqual (\x -> x + 1) (1+) (+1)
ERROR: a -> a is not an instance of class "Eq"
```

We define a type class as below

```
class Eq a where
 (==) :: a -> a -> Bool
 (/=) :: a -> a -> Bool

 x /= y = not (x==y)
```

The definition includes the class name `Eq`, signature and a default implementation (optional) which can be overwritten later. Elements of the class are called instances.

Classes allow restricted form of type generalization. The most general type with class constraint for `allEqual` is

```
allEqual :: Eq t => t -> t -> t -> Bool
```

For example, we can define elements of a list as

```
elem :: Eq t => t -> [t] -> Bool
elem _ []     = False
elem a (x:xs) = (a == x) || elem a xs
```

Instances Instances builds instance by "interpreting" signature functions

```
instance Eq Bool where
 True  == True  = True
 False == False = True
 _     == _     = False
```

We are defining the == symbol over the `Bool`.

Derived Classes Classes themselves can also depend on type conditions.

```
class Eq a => Ord a where
 (<), (>), (<=), (>=) :: a -> a -> Bool
 max, min :: a -> a -> a
 x < y  = x <= y && x /= y
 x >= y = y <= x
 x > y  = y <= x && x /= y

 max x y | x <= y    = y
         | otherwise = x
 max x y | x <= y    = x
```

```
                | otherwise = y
```

The above is saying, if `a` belongs to `Ord`, then `a` must also belong to `Eq`. Functions for `Eq` are inherited and some new ones must be given. This leads to **class hierarchies**, where classes can be hierarchically structured. Inheritance hierarchies like in Object-Oriented-Programming.

`Show` and `Read`

```
show :: Show a => a -> String
read :: Read a => String -> a
```

6 *Algebraic Data Types*

Until now, we've done data modeling with

- **Base types**: Ints, Bool, Char, Double, etc.
- **Compound types**: tuples, lists, functions, etc.
- **Type synonyms**: `type Complex = (Double, Double)`

Programming languages solve problems in the real world, so we better represent things in the real world. The solution is algebraic data types, which declare new types tailored to the objects being modeled. For example, we declare type `Months` with elements `January, February, ..., December`. These are new **data constructors**. For trees, declare type `Tree` with elements like

```
Node 1 (Node 10 Leaf Leaf) (Node 17 (Node (14 Leaf Leaf) (Node 20 Leaf Leaf))
```

Enumeration Types (Disjoint Unions)

```
data Season = Spring | Summer | Fall | Winter
data Month = January | February | March | April | May | June | July |
            August | September | October | November | December
```

Syntax: Starts with keyword `data`, names different (uniquely named) constructors, first letter of each constructor must be upper-case. It essentially defines a set: Season $=$ {Spring, Summer, Fall, Winter}. Functions can be written using pattern matching e.g. `whichSeason :: Month -> Season`.

Product Types

```
data People = Person Name Age
type Name = String
type Age = Int
```

An element of type `People` consists of a name n and an age a, e.g.,

```
Person "Uncle George" 85
Person "Levi Jeans" 501
```

Constructors are functions

```
? :type Person
Person :: Name -> Age -> People
```

Functions may be defined by pattern matching

```
showPerson :: People -> String
showPerson (Person n a) = n ++ " who is " ++ show a ++ " years old"
```

Product Types versus Tuples Alternative to products are tuples. Advantage of product types: Conceptual: new, self-contained type, objects are labeled hence types are unambiguous. Disadvantage include: Longer definitions, many polymorphic functions are no longer applicable `fst, zip, ...`

6.1 Enumeration and Product Types

These two types can be combined.

```
data Shape = Circle Double | Rectangle Double Double
```

We can write functions by pattern matching

```
area :: Shape -> Double
area (Circle r)      = pi * r * r
area (Rectangle h w) = h * w
```

Integration with classes No default functions like == or `show`

```
data Foo = D1 | D2 | D3
? D1 == D2
ERROR: No instance for (Eq Foo)
```

Class instances can be explicitly created

```
instance Eq Foo where
 D1 == D1 = True
 D2 == D2 = True
 D3 == D3 = True
 _  == _  = False

 ? D1 == D2
 False :: Bool
```

In some cases, they can be automatically derived.

```
data Foo = D1 | D2 | D3
          deriving (Eq, Ord, Enum, Show)
```

Recursive Types Sets of objects are often recursively define, below is a simple arithmetic grammar $\textsc{Expr} := \textsc{Int} | \textsc{Expr} + \textsc{Expr} | \textsc{Expr} - \textsc{Expr}$

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
            deriving (Show, Eq)
```

The program can be done via pattern matching

```
eval :: Expr -> Int
eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

A tree can be useful to describe many data structures. One can define a tree as an algebraic data structure in Haskell and define functions on it. Grammar:

$$\text{ITree} ::= \text{Leaf} \mid \text{Node Int ITree ITree}$$

Haskell data type

```
data ITree = Leaf | Node Int ITree ITree
           = deriving (Eq, Show)
```

Example tree t

```
Node 1 (Node 10 Leaf Leaf)
       (Node 17 (Node 14 Leaf Leaf)
       (Node 20 Leaf Leaf))
```

With that, we can define functions. Sum of values:

```
treeSum :: ITree -> Int
treeSum Leaf           = 0
treeSum (Node n t1 t2) = n + (treeSum t1) + (treeSum t2)
```

Depth:

```
treeSum :: Itree -> Int
depth Leaf           = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

Occurrences of an element

```
occurs :: ITree -> Int -> Int
occurs Leaf p = 0
occurs (Node n t1 t2) p
 | n == p = 1 + rest
 | otherwise = rest
 where rest = occurs t1 p + occurs t2 p
```

Note that this tree and the functions on it have monomorphic types → could use type parameters to get polymorphism: **Polymorphic Algebraic Types**.

Exercise! Have we seen this type before?

```
data L t = E | C t (L t)
          deriving (Eq, Ord, Show)
```

observe that

```
E :: L t
C :: t -> L t -> L t
```

What is the type of the following function?

```
f y E       = False
f y (C x 1) = x == y || f y 1
```

What is the result?

```
? f 3 (C 2 (C 3 (C4 E)))
```

Answers: 1. List Type! 2. `f :: Eq t => t -> L t -> Bool` 3. Check if element is inside a list `True :: Bool`

6.2 *Higher-order Programming with Data Types*

map to mapTree

```
mapTree :: (t -> u) -> Tree t -> Tree u
mapTree f Leaf            = Leaf
mapTree f (Node x t1 t2)  = Node (f x) (mapTree f t1) (mapTree f t2)

? mapTree (+2) (Node 7 (Node 20 Leaf Leaf) (Node 1 Leaf Leaf))
Node 9 (Node 22 Leaf Leaf) (Node 3 Leaf Leaf) :: Tree Int

? mapTree not (Node True (Node False Leaf Leaf) (Node True Leaf Leaf))
Node False (Node True Leaf Leaf) (Node False Leaf Leaf) :: Tree Bool
```

foldr to treeFold In a list l, : is replaced with f and E with e. In the tree t, a node N is replaced with f and a leaf L with e.

```
treeFold :: (a -> b -> b -> b) -> b -> Tree a -> b

treeFold f e Leaf         = e
treeFold f e (Node x l r) = f x (treeFold f e l) (treeFold f e r)
```

Tree Traversals

```
preorder t  = treeFold (\x l r -> [x] ++ l ++ r) [] t
postorder t = treeFold (\x l r -> l ++ r ++ [x]) [] t
inorder t   = treeFold (\x l r -> l ++ [x] ++ r) [] t
```

Algebraic types are "first class" citizens implying they are fully compatible with polymorphism and type classes.

6.3 Correctness of Algebraic Datatypes

Consider the algebraic datatype: `data Tree a = Leaf | Node a (Tree a ) Tree a)` again. A data type defines a set of terms for each type instance. E.g. `Tree Int` corresponds to $\{\text{NODE}, \text{NODE } 0 \text{ LEAF LEAF}, \dots\}$. Algebraic here means the smallest set S, where $\text{LEAF} \in S$ and $x \in a \wedge t_1 \in S \wedge t_2 \in S \implies (\text{NODE } x\ t_1\ t_2) \in S$. This is a recursive definition! A set S is built in steps: $\text{LEAF} \in S$ and $(\text{NODE } x\ t_1\ t_2) \in S$, where t_1 and t_2 in S in earlier steps.

7 *Lazy Evaluation*

Evaluation strategy until now has been unimportant. Haskell is **lazy**! Expressions are evaluated only when necessary. Subtle consequences such as data-driven computation. Evaluation is based on function application and substitution.

Evaluation based on function application and substitution. For example $f\ x\ y = x + y : f(9-3)(f\ 34\ 3) = (9-3) + (f\ 34\ 3)$. In Haskell, substitution occurs without argument evaluation. Evaluation of arguments is postponed.

$$\cdots = 6 + (f\ 34\ 3) = 6 + (34 + 3) = 6 + 37 = 43.$$

Some expressions may never be evaluated, this can save arbitrarily large amounts of time!
Potential Problem: Duplicated computation, e.g., `square x = x * x` and say `square (9-3) * (9-3) = 6 * (9-3) = 6 * 6 = 36`. The same expression $9 - 3$ is evaluated twice here. Duplication can be avoided by simultaneously reducing both occurrences. Implementation based on sharing: terms represented as directed graphs so functions arguments are evaluated only when needed and at most once.

Evaluation - pattern matching Arguments evaluated as far as needed to determine pattern match. Consider the following example

```
f [] _          = 0      -- f1
f _ []          = 0      -- f2
f (a:_) (b:_)   = a + b  -- f3
```

Then, `f [1 .. 3] [4 .. 6]` is executed as follows

```
f [1 .. 3] [4 .. 6]     -- does f1 match?
    = f (1 : [2 .. 3]) [4 .. 6]  -- No! does f2 match?
    = f (1 : [2 .. 3]) (4 : [5 .. 6]) -- No! does f3 match?
    = 1 + 4   -- Yes!
    = 5
```

Evaluation - Guards Execution proceeds sequentially, until success.

```
f a b c
 | a >= b && a >= c = a
 | b >= a && a >= c = b
 | otherwise        = c
```

Example is:

```
f (2+3) (4-1) (3+9)
```

```
 ?? (2+3) >= (4-1) && (2+3) >= (3+9)    -- try 1st guard
 ?? 5 >= 3 && 5 >= (3+9)  -- (2+3) is evaluated, but not (3+9)
 ?? True && 5 >= (3+9)
 ?? 5 >= (3+9)
 ?? 5 >= 12
 ?? False
 ?? 3 >= 5 && 5 >= 12  -- try 2nd guard, already partially evaluated
 ?? False && 5 >= 12
 ?? False   -- No need to evaluated second part
 ?? otherwise -- final guard = (True)
= 12  -- c already evaluated.
```

Local Definitions Local definitions (with `where`) are also lazily evaluated.

```
f a b
 | notNil l = front l
 | otherwise - b
 where
  l = [a .. b]

front (c:d:_) = c + d
front [c] = c

notNil [] = False
notNil _ = True
```

Then,

```
f 3 6
    ?? notNil l
           where l = [3 .. 6]
    ?? = notNil ([3 .. 6])
    ?? notNil (3:[4 .. 6]
    ?? True
= front l
  where
  l = 3:[4..6]
    = 3:4:[5..6]
= 3+4
= 7
```

Functions are evaluated top-down (outermost operator first) $f\ e_1(f\ e_2\ 17)$, and otherwise usually from left to right, depending on operator precedence $f\ e_1 + f\ e_2$, $f\ e_1 + f\ e_2 * f\ e_3$. This kind of evaluation is a natural as **eager evaluation**. But the consequences and possibilities are sur-

prising.

Application 1: Data-Driven Programming Data can be generated lazily (on demand), the result is improved runtime complexity. An example is computing the minimal element of a list of elements. A Data-Driven solution would be sorting the list and taking the sorted lists head. The resulting program is just `lmin = head . sort`. The complexity is just $\mathcal{O}(n)$, even though normally a sorting algorithm runs $\mathcal{O}(n \log n)$, with lazy evaluation, we do not need to sort all the elements.

Application 2: Infinite Data Lazy evaluation enables finite representation of infinite data. For example: infinite lists (streams). Can compute with infinite data in finite time: describe an infinite stream and compute with finite prefixes of it.

Part II

Formal Methods

8 IMP Language

I will omit the concrete syntax for IMP, refer to lecture slides.

8.1 Meta-variables

Meta-variables denote an arbitrary element of a syntactic category, e.g., an arbitrary statement. In this section of the course, we will use the following naming conventions for meta-variables[3]:

- n for numerals
- x, y, z for variables
- 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)

We use the naming conventions to avoid the need for explicit types, for example when we write $\forall x.P(x)$ we mean $\forall x \in \textsc{Var}.P(x)$.

[3] Meta-variables are written in math font, while program variables are written in `typewriter` font.

8.2 Meta-variables vs. Program Variables

Meta-variables x and y stand for arbitrary program variables, program variables `x` and `y` are concrete variables in a program. We write $\equiv$ for **syntactic equality** on variables, statements, etc. `x=y` might evaluate to true in some states, but `x`$\equiv$`y` is always false, as they are not syntactically equal. However, $x \equiv y$ might be true, as they could denote the same program variable.

8.3 Semantics of IMP expressions

A semantic function maps elements of syntactic categories (e.g. Numerals) to elements of semantic categories (e.g. Values)

Semantics of Numerals The semantic function $\mathcal{N} : \mathsf{Numeral} \to \mathsf{Val}$ maps a numeral n to an integer value $\mathcal{N}[\![n]\!]$. For example, $\mathcal{N}[\![9]\!] = 9$, with double digits we have $\mathcal{N}[\![n\ 8]\!] = \mathcal{N}[\![n]\!] \times 10 + 8$.

States The meaning of an expression depends on the values bound to the variables that occur in it. A state is a total function $\mathsf{State} : \mathsf{Var} \to \mathsf{Val}$. We use σ as a meta-variable for states.

- We define a designated (constant) state σ_{zero}, in which all variables have the value 0: $\sigma_{\mathsf{zero}}(x) = 0$ for all x.
- Updating states: $\sigma[y \mapsto v]$ is the function that overrides the association of y in σ by $y \mapsto v$.

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

- Two states σ_1 and σ_2 are equal if they are equal as functions:

$$\sigma_1 = \sigma_2 \iff \forall x.(\sigma_1(x) = \sigma_2(x))$$

Semantics of Arithmetic Expressions The sematic function $\mathcal{A} : \mathsf{Aexp} \to \mathsf{State} \to \mathsf{Val}$ maps an arithmetic expression e and a state σ to a value $\mathcal{A}[\![e]\!]\sigma$.

- $\mathcal{A}[\![x]\!]\sigma = \sigma(x)$
- $\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$
- $\mathcal{A}[\![e_1 \ \mathsf{op} \ e_2]\!] = \mathcal{A}[\![e_1]\!]\sigma \ \overline{\mathsf{op}} \ \mathcal{A}[\![e_2]\!]\sigma$ for $\mathsf{op} \in \mathsf{Op}$

where $\overline{\mathsf{op}}$ is the operation $\mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$ corresponding to op.

Semantics of Boolean Expressions The semantic function $\mathcal{B} : \mathsf{Bexp} \to \mathsf{State} \to \mathsf{Bool}$ maps a boolean expression b and a state σ to a truth value $\mathcal{B}[\![b]\!]\sigma$:

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

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

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

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

8.4 Properties of the Expression Semantics

The semantics is given by **recursive definitions** of functions $\mathcal{A}$ and $\mathcal{B}$. The values for composite elements are defined inductively in terms of the immediate constituents. Since the decomposition of elements is unique, the definition gives a well-defined functions. Inductive defintions suggest proofs by **structural induction**.

9 Operational Semantics

Big-Step semantics describe how the **overall** results of the executions are obtained. The system in this course is also called **Natural Semantics**. Small-step semantics describe how the individual steps of the computations take place. The system in this course is called **structural operational semantics**. Alternative approaches exists, for example abstract state machines.

9.1 Big-Step Semantics

Natural Semantics of IMP

Transition System
A transition system is a tuple $(\Gamma, T, \to)$ where

- Γ is a set of configurations
- T is a set of terminal configurations, $T \subseteq \Gamma$
- $\to$ is a transition relation, $\to \subseteq \Gamma \times \Gamma$

Operational semantics include two types of configurations

- $\langle s, \sigma \rangle$ which represents the statement s is to be executed in state σ.
- σ, which represents a final state (terminal configuration)

The transition relation $\to$ describes how executions take place. Big-step transitions are in the form $\langle s, \sigma \rangle \to \sigma'$.

$$\Gamma = \{\langle s, \sigma \rangle \mid s \in \textsc{Stm}, \sigma \in \textsc{State}\} \cup \textsc{State}$$

$$T = \textsc{State}$$

$$\to \subseteq \{\langle s\sigma \rangle \mid s \in \textsc{Stm}, \sigma \in \textsc{State}\} \times \textsc{State}$$

Big-Step Semantics of IMP

- `skip` does not modify the state
$$\frac{}{\langle \texttt{skip}, \sigma \rangle \to \sigma} \ (\textsc{Skip}_{\textsc{ns}})$$
- $x := e$ assigns the value of e to the variable x
$$\frac{}{\langle x := e, \sigma \rangle \to \sigma[x \to \mathcal{A}[\![e]\!]\sigma]} \ (\textsc{Ass}_{\textsc{ns}})$$
- Sequential composition $s : s'$. First s is executed in state σ, leading to σ', then s' is executed in state σ', leading to σ''.
$$\frac{\langle s, \sigma \to \sigma' \qquad \langle s', \sigma' \rangle \to \sigma''}{\langle s; s', \sigma \rangle \to \sigma''} \ (\textsc{Seq}_{\textsc{ns}})$$
- Conditional statement `if` b `then` s `else` s' `end`
$$\frac{\langle s, \sigma \rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}, \sigma \rangle \to \sigma} \ (\textsc{IfT}_{\textsc{ns}})$$

$$\frac{\langle s', \sigma\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to \sigma} \text{ (IfF}_{\text{NS}}\text{)}$$

- Loop statement `while` b `do` s `end`.

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

$$\frac{}{\langle \texttt{while } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma''} \text{ (WhF}_{\text{NS}}\text{)}$$

Inference rule definitions are actually **rule schemes**. Meta-variables in rule definitions are placeholders for statements, states, etc. A rule scheme describes infinitely many **rule instances**. A rule is instantiated when all meta-variables are replaced with syntactic elements. Below is an **instance** of the assignment rule of natural semantics:

$$\frac{}{\langle v := v + 1, \sigma_{\text{ZERO}}\rangle \to \sigma_{\text{ZERO}}[v \mapsto \mathcal{A}[\![v + 1]\!]\sigma_{\text{ZERO}}]} \text{ (Ass}_{\text{NS}}\text{)}$$

Rule instances can be combined to derive a transition $\langle s, \sigma\rangle \to \sigma'$. The result is a **derivation tree** T. The root of T would be $\langle s, \sigma\rangle \to \sigma'$, denoted as $\mathsf{root}(T) \equiv \langle s, \sigma\rangle \to \sigma'$. The leaves of T are axiom rule instances, and the internal nodes of T are conclusions of rule instances and have the corresponding premises as immediate children. The side-conditions of all instantiated rules must be satisfied.

$$\Gamma\langle s, \sigma\rangle \to \sigma' \iff \exists T.\textsc{root}(T) \equiv \langle s, \sigma\rangle \to \sigma'$$

Termination For **an IMP statement** s, we define termination in the context of big-step semantics as follows. The execution of a statement s in state σ **terminates successfully** iff there exists a state σ' such that $\vdash \langle s, \sigma\rangle \to \sigma'$, **fails to terminate** iff there is no state σ' such that $\vdash \langle s, \sigma\rangle \to \sigma'$. For example, `while true do skip end` fails to terminate.

Semantic Equivalence

Two statements s_1 and s_2 are **semantically equivalent** (written $s_1 \simeq s_2$) iff:

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

An example is `while false do s end` $\simeq$ `skip`

The big-step semantics of IMP is deterministic. Formally,

$$\forall s, \sigma, \sigma', \sigma''.(\vdash \langle s, \sigma\rangle \to \sigma' \wedge \vdash \langle s, \sigma\rangle \vdash \sigma'' \implies \sigma' = \sigma''$$

9.2 *Small-Step Semantics*

Small-step semantics focuses attention on the **individual steps** of an execution (execution of assignments, executions of if-conditions, while-iterations, etc). Describing small steps of the execution allows one to express the **order of execution** of individual steps. Can be used to express interleaving computations or evaluation order for expressions. Always describing the **next small step** allows one to express **properties of non-terminating programs**.

Transitions in SOS The configuration are the same for natural semantics ($\langle s, \sigma\rangle$ or σ). We use γ as a meta-variable for (terminal or non-terminal) configurations. The transition relation $\to_1$ can have two forms.

- $\langle s, \sigma \to_1 \langle s', \sigma'\rangle$: the execution of s from σ is **not completed** and the remaining computation is expressed by the intermediate configuration $\langle s', \sigma'\rangle$.
- $\langle s, \sigma\rangle \to_1 \sigma'$: the execution of s from σ **has terminated** and the final state is σ'

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

Formally, the **transition system** is denoted as

$$\Gamma = \{\langle s, \sigma\rangle \mid s \in \textsc{Stm}, \sigma \in \textsc{State}\} \cup \textsc{State}$$

$$T = \textsc{State}$$

$$\to_1 \subseteq \{\langle s, \sigma\rangle \mid s \in \textsc{Stm}, \sigma \in \textsc{State}\} \times \Gamma$$

We say that a non-terminal configuration $\langle s, \sigma\rangle$ is **stuck** if there does not exist a configuration γ such that $\langle s, \sigma\rangle \to_1 \gamma$ (terminal configurations are never stuck). We will again define the transition relation $\to_1$ using a derivation system, and write $\vdash \langle s, \sigma \to_1 \gamma$ to mean there exists a **finite** derivation tree ending in $\langle s, \sigma\rangle \to_1 \gamma$

Transitive Closure

$$\vdash \langle s, \sigma \to_1 \gamma \iff \exists T.\textsc{root}(T) \equiv \langle s, \sigma\rangle \to_1 \gamma$$

SOS of IMP

- `skip` does not modify the state

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to_1 \sigma} \ (\textsc{Skip}_{\text{SOS}})$$

- $x := e$ assigns the value of e to the variable x

$$\frac{}{\langle x := e, \sigma\rangle \to_1 \sigma[x \to \mathcal{A}[\![e]\!]\sigma]} \text{(Ass}_{\text{SOS}}\text{)}$$

- `skip` and assignment require only one step to reach a final state
- Sequential composition. Either s executes completely in one step, or s is not executed completed after one step

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s; s', \sigma\rangle \to_1 \langle s', \sigma'\rangle} \text{(Seq1}_{\text{SOS}}\text{)}$$

$$\frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s; s', \sigma\rangle \to_1 \langle s''; s', \sigma'\rangle} \text{(Seq2}_{\text{SOS}}\text{)}$$

- Conditional statement `if` b `then` s_1 `else` s_2 `end` is to determine the outcome of the test b, and thereby which branch to select.

$$\frac{}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \langle s, \sigma\rangle} \text{(IfT}_{\text{SOS}}\text{)}$$

$$\frac{}{\langle \texttt{if } b \texttt{ then } s \texttt{ else } s' \texttt{ end}, \sigma\rangle \to_1 \langle s', \sigma\rangle} \text{(IfF}_{\text{SOS}}\text{)}$$

- Loop statement `while` b `do` s `end`.

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

Multi=Step Executions A k-step execution, written $\gamma \to_1^k \gamma'$ is an execution from γ to γ' in exactly k steps where $k \in \mathbb{N}$. Formally,

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

Derivation Sequences A **derivation sequence** is a (non-empty, finite or infinite) sequence of configurations $\gamma_0, \gamma_1, \gamma_2, \ldots$, 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 derivation sequence is **finite**, then the last configuration in the sequence is either a terminal configuration of a stuck configuration

A derivation sequence shows a sequence of transitions which cannot be extended with further transitions.

Termination The execution of a statement s in state σ **terminates** iff there is a finite derivation sequence starting with $\langle s, \sigma\rangle$, **runs forever** iff there is an infinite derivation sequence starting with $\langle s, \sigma\rangle$. **Terminates successfully** iff $\exists\sigma'.\langle s, \sigma\rangle \to_1^* \sigma'$. Note these are properties of **configurations** and not statements alone. For example, some `while` loops terminate successfully in sometimes, and runs forever in others.

Proving Properties of Derivation Sequences A finite derivation sequence has a length which is a natural number. When reasoning about finite derivation sequences, we usually use strong induction on the length of a derivation sequence. More generally, we reason about a multi-step execution $\gamma \to_1^k \gamma'$ by **strong induction on the number of steps** k [4]

- Define $P(k) \equiv$ "for all executions of length k, our property holds".
- Prove $P(k)$ for arbitrary k, with the induction hypothesis $\forall k' < k.\ P(k')$

[4] This is unlike Big-Step semantics, where we do induction on the shape of the derivation tree.

10 Axiomatic Semantics

10.1 Motivation

Formal semantics can be used to prove the correctness of a program. **Partial correctness** expresses that if a program terminates then there will be a certain relationship between the initial and the final state. **Total correctness** express that a program will terminate and there will be a certain relationship between the initial and the final state. **Total correctness = partial correctness + termination**.

Consider the factorial statement

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

Specification: The final value of `y` is the factorial of the initial value of `x`. The statement is partially correct, it does not terminate for $x < 1$. We can express the specification formally based ona formal semantics

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

This specification expresses partial correctness using big-step semantics. We could have used small-step semantics to formulate the property instead. We prove partial correctness in three steps.

- The body of the loop satisfies

  $$\forall \sigma, \sigma''. \vdash \langle \texttt{y:= y *x; x:= x-1}, \sigma \rangle \to \sigma'' \wedge \sigma''(x) > 0 \implies \sigma(\texttt{y}) \times \sigma(\texttt{x})! = \sigma''(\texttt{y}) \times \sigma''(\texttt{x})! \wedge \sigma(x) > 0$$

- The loop satisfies

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

- The whole statement is partially correct

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

However, proofs with formal semantics are too detailed to be practical. Axiomatic semantics provides a way of constructing these proofs conveniently. Proofs can focus on essential properties of interest, decomposing the program into small parts happens naturally. The induction for reasoning about loops is built into the semantic rule for loops.

10.2 Hoare Logic

Hoare Triples and Assertions Properties of programs are specified as **Hoare Triples**

$$\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$$

where s is a statement and **P**, **Q** are assertions (about the state). Some terminology:

- The assertion **P** is called the **precondition** of a triple $\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$
- The assertion **Q** is called the **postcondition** of a triple $\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$
- Assertions are boolean expressions, with some additional features. We use **P, Q, R** as meta-variables over assertions.

The informal meaning of $\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$ is:

> If **P** evaluates to true in an initial state σ and if the execution of s from σ terminates in a state σ' then **Q** will evaluate to true in σ'

This meaning describes **partial correctness**, that is, termination is not an essential property. It is also possible to assign different meanings to Hoare Triples.

We allow assertions to contain **logical variables**, they may only occur in assertions. Logical variables are **not** program variables, and may, thus not be accessed in programs; in particular, they are never assigned to. Logical variables can be used to "save" values in the initial state, so that they can be referred to later. This is the triple specifies the factorial example. Note that we cannot use the program variable x to specify **Q** since it refers to the final value of x, which is obviously wrong.

$$\{\mathsf{x} = \mathsf{N}\}$$

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

$$\{\mathsf{y} = \mathsf{N}! \wedge \mathsf{N} > 0\}$$

States map logical variables (and program variables) to their values.

Assertion Language Pre- and postconditions are assertions, that is, boolean expressions plus logical variables. In particular, we will use program boolean expressions b as assertions. It is common practice to use a richer set of expressions for assertions, for instance, to include quantification. We will use additional expressions when it is convenient. We assume that the subsitution lemma from the exercises still holds when we use an extended assertion language:

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

We will use the following convenient notations

- "$P_1 \wedge P_2$" for "P_1 `and` P_2"
- "$P_1 \vee P_2$" for "P_1 `or` P_2"
- "$\neg P$" for "`not` P"

Derivation System We formalize an axiomatic semantics of IMP by describing the valid Hoare triples. This is done by a derivation system, the derivation rules specify which triples can be derived for each statement. The premises and conclusions of the derivation rules are Hoare triples. Derivation trees are as defined before, using the rules presented later. Similary to the other derivation systems we have studied, we write $\vdash \{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$ if and only if there exists a (finite) derivation tree ending in $\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$.

$$\vdash \{\mathbf{P}\}\ s\ \{\mathbf{Q}\} \iff \exists.\textsc{Root}(T) \equiv \{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$$

Here are the axiomatic semantics of IMP

- `skip` does not modify the state
$$\frac{}{\{\mathbf{P}\}\ \texttt{skip}\ \{\mathbf{Q}\}}\ (\textsc{Skip}_{\textsc{Ax}})$$
- $x := e$ assigns the value of e to the variable x
$$\frac{}{\{\mathbf{P}[x \mapsto e]\}\ \texttt{x := e}\ \{\mathbf{P}\}}\ (\textsc{Ass}_{\textsc{Ax}})$$
Let σ be the initial state, precondition: $\mathcal{B}[\![\mathbf{P}[x \mapsto e]]\!]\sigma$, which is equivalent to $\mathcal{B}[\![\mathbf{P}]\!]\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$ (substitution lemma). The final state is $\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$. Consequently, $\mathcal{B}[\![\mathbf{P}]\!]$ holds in the final state.
- Sequential composition
$$\frac{\{\mathbf{P}\}\ s\ \{\mathbf{Q}\} \qquad \{\mathbf{Q}\}\ s'\ \{\mathbf{R}\}}{\{\mathbf{P}\}\ s; s'\ \{\mathbf{R}\}}\ (\textsc{Seq}_{\textsc{Ax}})$$
- Conditional statement `if` b `then` s_1 `else` s_2 `end`
$$\frac{\{b \wedge \mathbf{P}\}\ \texttt{s}\ \{\mathbf{Q}\} \qquad \{\neg b \wedge \mathbf{P}\} s' \{\mathbf{Q}\}}{\{\mathbf{P}\}\ \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{end}\ \{\mathbf{Q}\}}\ (\textsc{If}_{\textsc{Ax}})$$
- Loop statement `while` b `do` s `end`.
$$\frac{\{b \wedge \mathbf{P}\}\ s\ \{\mathbf{P}\}}{\{\mathbf{P}\}\ \ \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}\ \{\neg b \wedge \mathbf{P}\}}\ (\textsc{While}_{\textsc{SOS}})$$
The assertion **P** is the **loop invariant**

The rules so far manipulate assertions syntactically, for example so far we cannot derive the triple

$$\{\texttt{x = 4} \wedge \texttt{y = 5}\}\ \texttt{skip}\ \{\texttt{y = 5} \wedge \texttt{x = 4}\}$$

So we introduce **semantic entailment**, which expresses these reasoning steps: we write $\mathbf{P} \models \mathbf{Q}$ iff "for all states σ, $\mathcal{B}[\![\mathbf{P}]\!]\sigma = tt$ implies $\mathcal{B}[\![\mathbf{P}]\!]\sigma = tt$. The **rule of consequence** allows semantic entailments in derivations

$$\frac{\{\mathbf{P}'\}\ s\ \{\mathbf{Q}'\}}{\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}}\ \textsc{Cons}_{\textsc{Ax}}$$

Of course, with the side conditions if $\mathbf{P} \models \mathbf{P}'$ and $\mathbf{Q}' \models \mathbf{Q}$.

Total Correctness We introduce an alternative form of Hoare Triple $\{\mathbf{P}\}\ s\ \{\Downarrow \mathbf{Q}\}$. The informal meaning is "If $\mathbf{P}$ evaluates to true in the initial state σ then the execution of s from σ terminates and $\mathbf{Q}$ will evaluate to true in the final state. This meaning describes total correctness, that is, termination is required. We do not mix these triples with those of partial correctness! The two form two separate axiomatic semantics (and corresponding derivation systems). However, all total correctness derivation rules are analogous to those for partial correctness, except for the rule for loops.

Termination is proved using **loop variants**. A loop variant is an expression that evaluates to a value in a well-founded set (for instance, $\mathbb{N}$) before each iteration. Each loop iteration must decrease the value of the loop variant. The loop has to terminate when a minimal value of the well-founded set is reached (or earlier than this).

For simplicity, we consider loop variants that evaluate to values in $\mathbb{N}$. We use arithmetic expressions e of IMP to represent loop variants. We prove explicitly that the value of e will be non-negative before each loop iteration. The intuition is: a correct loop variant provides an upper bound on the number of loop iterations.

Total correctness derivation rule for loops:

$$\frac{\{b \wedge \mathbf{P} \wedge e = Z\}\ s\ \{\Downarrow \mathbf{P} \wedge e < Z\}}{\{\mathbf{P}\}\ \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}\ \{\Downarrow \neg b \wedge \mathbf{P}\}}\ \textsc{WhTot}_{\textsc{Ax}}$$

Of course, with the side condition if $b \wedge \mathbf{P} \models 0 < e$, where Z is a fresh logical variable not used in $\mathbf{P}$.

10.3 Soundness and Completeness

- **Soundness** If a property can be proved then it does indeed hold. An unsound derivation system is useless.

$$\vdash \{\mathbf{P}\}\ s\ \{\mathbf{Q}\} \implies \models \{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$$

- **Completeness** If a property does hold then it can be proved. With an incomplete derivation system, a program might be correct, but we cannot prove it.

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

11 Modeling

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:

- Explicit state model checking: represent state explicitly through concrete values.
- Symbolic model checking: represent state through boolean formulas.

We will focus on explicit state model checking.

Model Checking Process

- **Modeling phase**: Model the system under considering using the description language of your model checker (possibly a programming language). Formalize the properties to be checked.
- **Running phase**: Run the model checker to check the validity of the property in the system model.
- **Analysis phase**: If property is satisfied, celebrate and move on to next property, if violated analyze and counterexample, if out of memory reduce model and try again.

We model systems as **communicating sequential processes** (agents), there are a finite number of processes, and they execute in an interleaved way. Processes can communicate via shared variables, synchronous message passing, asynchronous message passing.

11.1 Protocol Meta Language Promela

Input language of the Spin model checker. Main objects are processes, channels, and variables. C-like syntax. Spin can "execute" (simulate models).

Constant declarations

```
#define N 5
mtype = { ack, req };
```

Structure declarations

```
typedef vector { int x; int y};
```

Global channel declarations

```
chan buf = [2] of { int };
```

Global variable declarations

```
byte counter;
```

Process declarations

```
proctype myProc(int p) { ... }
```

Promela Process Declarations Simple form `proctype myProc(int p) ...` The body consists of a sequence of variable declarations, channel declarations, and statements. No arrays as parameters.

Active processes (start `N` instances of `myProc` in the initial state.

```
active [N] proctype myProc(...) {...}
```

`init` process is started in the initial state.

Type	Value Range
`bit` or `bool`	0...1
`byte`	0...255
`short`	$-2^{15}...2^{15}-1$
`int`	$-2^{31}...2^{31}-1$

Table 1. Primitive Types (No floats or mathematical (unbounded) integers

Promela Types User-defined types

- Arrays: `int name[4]`
- Structures
- Type of symbolic constants: `mtype`

Channel type: `chan`

Initial State
Global Variables are specified initial value or default value. Global Channels are empty. `init` and processes declared `active`.

State Transitions A statement can be **executable** or **blocked**.

- Send is blocked if channel is full
- $s_1; s_2$ is blocked if s_1 is blocked
- `timeout` is executable if all other statements are blocked

A transition is made in three steps:

- Determine **all executable statements** of all active processes. If no executable statement exists, transition system gets stuck
- Choose **non-deterministically** one of the executable statements. Non-determinism models concurrency through interleaving.

- Change the state according to the chosen statement.

11.2 State Space of Programs

Number of states for **Sequential Programs**

$$\#\text{program locations} \times \prod_{\text{variable}\, x} |\textsc{dom}(x)|$$

where $|\textsc{dom}(x)|$ denotes the number of possible values of variable x.

The number of states grows exponentially in the number of variables. State space explosion!

Number of states for **Concurrent Programs**. The number of states $P \equiv P_1 \| \ldots \| P_N$ is at most

$$\#\textsc{states of}\ P_1 \times \cdots \times \#\textsc{states of}\ P_N = \prod_{i=1}^{N} (\#\textsc{program locations}_i \prod_{\text{variable}\, x_i} |\textsc{dom}(x_i)|)$$

where $|\textsc{dom}(x)|$ denotes the number of possible values of variable x.

The number of states grows exponentially in the number of variables. State space explosion!

Number of states for **Promela Programs**. The number of states of a system with N processes and K channels is at most

$$\#\text{program locations} \times \prod_{\text{variable}\, x_i} |\textsc{dom}(x_i)| \times \prod_{j=1}^{K} |\textsc{dom}(c_j)|^{\textsc{cap}(c_j)}$$

where $|\textsc{dom}(c_j)|$ denotes the number of possible messages of channel c_j, and $\textsc{cap}(c_j)$ is the capacity (buffer size) of channel c_j. Number of states grows **exponentially** in the number of capacity of channels, **state space explosion**!

11.3 Promela Statements

- `skip` does not change state (except location counter), always executable.
- `timeout` does not change the state (except location counter), executable if all other statements in the system are blocked.

- `assert(E)` aborts execution if expression `E` evaluates to zero; otherwise equivalent to `skip`, always executable.
- Assignment
  - `x = E` assigns the value of `E` to variable `x`.
  - `a[n] = E` assigns the value of `E` to an array element `a[n]`
  - Always executable.
- Sequential composition `s1;s2` is executable if `s1` is executable.
- Expression statement
  - Evaluates expression `E`
  - Executable if `E` evaluates to value different from zero
  - `E` must not change state (no side effects)
  - Examples:

```
run myProcess;
x > 0;
```

- Selection

```
if
:: s1 /* option 1 */
:: ...
:: sn /* option n */
fi
```

  - Executable if at least one of its options is executable.
  - Chooses an option non-deterministically and executes.
  - Statement `else` is executable if no other option is executable (may occur at most in one option).
- Repetition is executable if at least one of its options is executable. Choose repeatedly an option non-deterministically and executes it. Terminates when a `break` or `goto` is executed.

```
do
:: s1 /* option 1 */
:: ...
:: sn /* option n */
od
```

- **Atomic**. Basic statements are executed atomically (not interleaving during execution `skip, timeout, assert, assignment` expression statements are atomic. For complex statements, `atomic s` executes `s` atomically. Executable if the first statement of `s` is executable. If any other statement within `s` blocks once the execution of `s` has started, atomicity is lost. Consider the example with Binary semaphores (lock). The first block is a wrong example

```
bit locked; /* global */
```

```
/* lock */
locked == 0;
locked = 1;

/* critical section */
locked = 0; /* unlock */
```

It is wrong as it allows interleaving executions between checking if the lock is available and setting it to 1. The correct approach would be.

```
bit locked; /* global */

/* lock */
atomic {
    locked == 0;
    locked = 1;
}
/* critical section */
locked = 0; /* unlock */
```

- Macros. Promela does not contain procedures. Effect can be achieved using macros.

```
inline lock() {
    atomic {
        locked == 0;
        locked = 1
    }
}
```

A macro just defines a replacement text for a symbolic name, possibly with parameters. However, no new variable scope, no recursion and no return values, as it is just a semantic representation for text. Define macro globally before its first use.

- Channels. `chan ch = [d] of t1, ... , tn` declares a channel. Channel can buffer up to `d` messages. If $d > 0$ then we have a buffered channel (FIFO). If $d = 0$ then we have an unbuffered channel (rendex-vous). Each message is a tuple whose elements have types `t1, ..., tn`

12 Linear Temporal Logic

12.1 Linear-Time Properties

Transition Systems Revisited We use a slightly different definition here (than earlier in the courses).

A finite transition system is a tuple $(\Gamma, \sigma_I, \rightarrow)$

- Γ: a finite set of configurations
- σ_I: an initial configuration, $\sigma_I \in \Gamma$
- $\rightarrow$: a transition relation, $\rightarrow \subseteq \Gamma \times \Gamma$

The difference is that we fixed the initial configurations, (transition systems model only one program/system, not all programs of a programming language), and also we omit terminal configurations from the definition. Termination can be modelled by transition to a special extra sink state (which allows further transitions only back to itself).

Transition System of a Promela Model

- Configurations: states (see previous section) - states that include global variables and channels and for every active process the local variables, channels and the location counter.
- Initial configuration: initial state (see previous section).
- Transition relation: defined by operational semantics of statements.
- A Promela model has a finite number of states.

Computations

- S^ω is the set of **infinite sequences** of elements of set S, $s_{[i]}$ denotes the i-th element of the sequence $s \in S^\omega$.
- $\gamma \in \Gamma^\omega$ is a **computation** of a transition system if:
  - $\gamma_{[0]} = \sigma_I$
  - $\gamma_{[i]} \rightarrow \gamma_{[i+1]}$ (for all $i \geq 0$)
  - Note: we use σ to range over the states Γ of a transition system.
  - Note (notation above): if $\gamma = \sigma_0\sigma_1\sigma_2\sigma_3\ldots$ then $\gamma_{[i]} = \sigma_i$
- $\mathcal{C}(TS)$ is the set of all computations of a transition system TS.

Linear-Time Properties 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. P specifies a particular set of infinite sequences of configurations. TS satisfies LT-property P (over Γ):

$$TS \models P \text{ if and only if } \mathcal{C}(TS) \subseteq P$$

All computations of TS belong to the set P.

By contrast: branching-time properties (not in this course) can also express the **existence of a computation**.

LT-Properties: Example
All opened files must be closed eventually:

$$P = \{\gamma \in \Gamma^\omega \mid \forall i \geq 0 \ : \ \gamma_{[i]}(o) = 1 \implies \exists n > 0 \ : \ \gamma_{[i+n]}(o) = 0\}$$

However, the explicit representation above (defining the set of sequences) is not convenient. Logical formalism needed to simplify specification of LT-properties.

From Configurations to (Sets of) Propositions For a transition system TS, we additionally specify a set AP of atomic propositions (of our choice)

- An atomic proposition is a proposition containing no logical connectives.
- Example: $AP = \{\text{OPENED}, \text{CLOSED}\}$ (for files)
- Example: $AP = \{x > 0, y \leq x\}$

We must provide a **labeling function** that maps configurations to sets of atomic propositions from AP

- $L : \Gamma \to \mathcal{P}(AP)$
- Example: $L(\sigma) = \begin{cases} \{\text{OPEN}\} & \text{if } \sigma(o) = 1 \\ \{\text{CLOSED}\} & \text{if } \sigma(o) = 0 \\ \{\} & \text{otherwise} \end{cases}$

We call $L(\sigma)$ an abstract state. From now on, we consider AP and L to be a part of the transition system.

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

- $t \in \mathcal{P}(AP)^\omega$ is a **trace of a transition system** TS if $t = L(\gamma_{[0]})L(\gamma_{[1]})L(\gamma_{[2]})\ldots$ and γ is a computation of TS.
- $\mathcal{T}(TS)$ is the set of all traces of a transition system TS.
- LT-properties are typically specified over infinite sequences of abstract states, rather than over sequences of configurations.

$$P = \{t \in \mathcal{P}(AP)^\omega \mid \forall i \geq 0 \ : \ \gamma_{[i]}(o) = 1 \implies \exists n > 0 \ : \ \gamma_{[i+n]}(o) = 0\}$$

Safety Properties The intuition is "something bad is never allowed to happen (and can't be fixed)"

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 $\hat{t}$ of t such that for every infinite sequence t' with prefix $\hat{t}$, $t' \notin P$.

$\hat{t}$ is called a **bad prefix**; essentially, this finite sequence of steps already violates the property (whatever happens afterwards). Safety properties are violated in finite time and cannot be repaired. For example:

- State properties, for instance, invariance

$$P = \{t \in P(AP)^\omega \mid \forall i \geq 0 \ : \ \text{OPEN} \ \in t_{[i]} \vee \ \text{CLOSED} \ \in t_{[i]}\}$$

- "Money can be withdrawn only after correct PIN has been entered"

Liveness Properties The intuition is "something good will happen eventually", and "if the good thing has not happened yet, it could happen in the future.

An LT-property P is a liveness property if for every finite sequences $\hat{t} \in \mathcal{P}(AP)^*$ is a prefix on an infinite sequence $t \in P$. A liveness property does not rule out any prefix. Every finite prefix can be extended to an infinite sequence that is in P.

Liveness properties are violated in infinite time. Some examples:

- All opened files must be closed eventually

$$P = \{t \in \mathcal{P}(AP)^\omega \mid \forall i \geq 0 \ : \ \gamma_{[i]}(o) = 1 \implies \exists n > 0 \ : \ \gamma_{[i+n]}(o) = 0\}$$

- "The program terminates eventually"

12.2 *Linear Temporal Logic*

Linear Temporal Logic (LTL) allows us to formalize LT-properties of traces in a convenient and succinct way. We will see syntax and semantics for LTL (no inference rules, etc.) Whether or not the traces of a finite transition system satisfy an LTL formula is decidable.

Basic Operators Syntax:

$$\phi = p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathsf{U} \phi \mid \ \bigcirc \phi$$

where p is a proposition from a chosen set of atomic propositions $AP \neq \varnothing$, $\phi \mathsf{U} \psi$ means "ϕ until ψ", and $\bigcirc\phi$ means next.

LTL Semantics
$t \models \phi$ expresses that trace $t \in \mathcal{P}(AP)^\omega$ satisfies LTL formula ϕ

$t \models p$	iff	$p \in t_{[0]}$
$t \models \neg\phi$	iff	not $t \models \phi$
$t \models \phi \wedge \psi$	iff	$t \models \phi$ and $t \models \psi$
$t \models \phi \mathsf{U} \psi$	iff	there is a $k \geq 0$ with $t_{\geq k} \models \psi$
		and $t_{\geq j} \models \phi$ for all j such that $0 \leq j < k$
$t \models \bigcirc\phi$	iff	$t_{\geq 1} \models \phi$

where $t_{\geq i}$ is the suffix of t starting at t_i.

Derived Operators

- TRUE, FALSE, $\vee$, $\implies$, $\impliedby$, $\iff$ defined as usual.
- Eventually: $\Diamond\phi \equiv (\text{TRUE } \mathsf{U}\phi)$
- Always (from now): $\Box\phi \equiv \neg\Diamond\neg\phi$

Precedence: unary operators always have the highest precedence, usually we use parentheses to explicitly clarify other ambiguities.

Useful Specification Patterns

Strong Invariant: $\Box\psi$

- ψ always holds
- A file is always opened or closed: $\Box(\text{OPENED} \vee \text{CLOSED})$
- Safety property

Monotone Invariant: $\Box(\psi \implies \Box\psi)$

- Once ψ is true, then ψ is always true
- For example, once information is public, it can never become secret again (but it may always stay secret): $(\text{PUBLIC} \implies \Box\text{PUBLIC})$
- Safety property

Establishing an Invariant: $\Diamond\Box\psi$

- Eventually ψ will hold
- Example: system initialization starts server: $\Diamond\Box\text{SERVERRUNNING}$
- Liveness property

Responsiveness: $\square(\psi \implies \lozenge\phi)$

- Every time that ψ holds, ϕ will eventually hold.
- For example, all opened files must be closed eventually:

$$\square(\textsc{open} \implies \lozenge\textsc{closed})$$

- Liveness property

Fairness: $\square\lozenge\psi$

- ψ holds infinitely often
- For example, producer does not wait infinitely long before entering the critical section:

$$\square\lozenge\textsc{critical}$$

- Liveness property

LTL Model Checking Problem

Given a finite transition system TS and an LTL formula ϕ decide whether $t \models \phi$ for all $t \in \mathcal{T}(TS)$

We need to check inclusion of traces: LTL formula ϕ describes a set of traces $P(\phi)$, we need to determine whether or not $\mathcal{T}(TS) \subseteq P(\phi)$. **Naively searching all traces is not an option** (infinite length).

12.3 Checking Safety Properties

Automatic checking of LTL formulas is non-trivial because traces are infinite. For **safety properties**, recall that any violation can be observed after finite prefix. The idea is to characterize all finite prefixes of the traces of a transition system using a **finite automation** and check whether any of them violates the safety property. For **liveness properties**, we need other strategies.

Finite Automaton for Finite Prefixes
Given a transition system $TS = (\Gamma, \sigma_I, \to)$, we define an NFA $\mathcal{FA}_{TS}$ characterizing all finite prefixes $\mathcal{T}_{\text{fin}}(TS)$ of the traces of TS. The automaton $\mathcal{FA}_{TS} = (Q, \Sigma, \delta, Q_0, F)$

- $Q = \Gamma \cup \{_0\}$ where $\sigma_0 \notin \Gamma$
- $\Sigma = \mathcal{P}(AP)$
- $\delta = \{(\sigma, p, \sigma') \mid \sigma \to \sigma' \wedge p = L(\sigma')\} \cup \{(\sigma_0, p, \sigma_I) \mid p = L(\sigma_I)\}$
- $Q_0 = \{\sigma_0\}$

- $F = Q$ (accept any prefix of a trace)

Regular Safety Properties
A safety property is regular if its **bad prefixes** are described by a **regular language** over the alphabet $\mathcal{P}(AP)$. Every invariant over AP is a regular safety property. For the property defined by $\Box p$, all bad prefixes start with S^*T where S describes any subset of $\mathcal{P}(AP)$ that contains p, and T any subset that does not contain p. For example, bad prefixes for $\Box$ OPEN are described by

$$(\{\text{OPEN}\}|\{\text{OPEN, CLOSED}\})^*(\{\}|\{\text{CLOSED}\})$$

Non-regular safety properties also exist. For example: Vending machine - at least as many coins inserted as drinks dispensed. Bad prefixes: regular languages "cannot count".

To **check for regular safety properties**, we follow the steps:

1. Describe finite prefixes $\mathcal{T}_{\text{fin}}(TS)$ by **finite automaton** $\mathcal{FA}_{TS}$.
2. Describe bad prefixes of regular safety property P by finite automaton $\mathcal{FA}_{\bar{P}}$.
3. Construct finite automaton for product for $\mathcal{FA}_{TS}$ and $\mathcal{FA}_{\bar{P}}$.
4. Check if the resulting automaton has any reachable accepting states
   - If not, the property P is never violated in traces of TS.
   - If yes, the property P is violated.
   - Moreover, each word in the accepted language of the product automaton is a counter example.

ω-Regular Languages Regular expressions denote languages of **finite** words, ω-regular expressions denote languages of **infinite** words. An ω-regular expression G has the form

$$G = E_1F_1^\omega + \cdots + E_nF_n^\omega \quad (1 \leq n)$$

where E_i and F_i are regular expressions and $\epsilon \notin \mathcal{L}(F_i)$.

$$\mathcal{L}(F^\omega) = \{w_1w_2w_3\ldots \mid \forall i.w_i \in \mathcal{L}(F)\}$$

Büchi Automata
Büchi automata are similar to finite automata, but accept **infinite words**. The class of languages accepted by non-deterministic Büchi automata agrees with the class of ω-regular languages. A non-deterministic Büchi automaton (NBA) is a tuple $(Q, \Sigma, \delta Q_0, F)$

- Q is a finite set of states
- Σ is a finite alphabet

- δ is a transition relation, $\delta \subseteq Q \times \Sigma \times Q$
- $Q_0 \subseteq Q$ is a set of initial states
- $F \subseteq Q$ is a set of accepting states

A run of an NBA accepts its input if it **passes infinitely often through an accepting state**. It also enjoys many of the properties of finite automata. We can construct the product of two NBA, emptiness is decidable.

12.4 LTL Model Checking

1. Describe traces $\mathcal{T}(TS)$ by NBA $\mathcal{BA}_{TS}$.
2. For an LTL formula ϕ, construct NBA $\mathcal{BA}_{\neg\phi}$ that accepts the traces characterized by $\neg\phi$ (bad traces).
3. Construct NBA for product of $\mathcal{BA}_{TS}$ and $\mathcal{BA}_{\neg\phi}$
4. Check whether the language accepted by product NBA is empty. If the language is non-empty, property ϕ is violated. Each word in the language is a counter-example.

For a finite transition system TS and an LTL formula ϕ, the model checking problem $TS \models \phi$ is solvable in $\mathcal{O}(|TS| \times 2^{|\phi|})$.

Where $|TS|$ is the size of the transition system, which grows exponentially in the number of variables, processes, and channels. $|\phi|$ is the size of ϕ; exponential complexity comes from the construction of $\mathcal{BA}_{\neg\phi}$