Formal Methods and Functional Programming Summary FS22

Danny Camenisch

June, 2022

Contents

Functional Programming **3**
1. Introduction . . . 3
1.1 Basic Concepts of Functional Programming . . . 3
1.2 Introduction to Functional Programming . . . 3
2. Natural Deduction . . . 5
2.1 Propositional logic . . . 5
2.2 First-order logic . . . 7
2.4 Equality . . . 8
3. Correctness . . . 9
3.1 Termination . . . 9
3.2 Reasoning . . . 9
4. List and Abstraction . . . 10
4.1 List Type . . . 10
4.2 Patterns . . . 10
4.3 Intermezzo - Advice on Recursion . . . 10
4.4 List Comprehension . . . 11
4.5 Induction over Lists . . . 11
5. Abstraction . . . 11
5.1 Polymorphic Types and Reusability . . . 11
5.2 Higher-Order Functions . . . 11
5.3 λ - Expressions . . . 12
5.4 Functions as Values . . . 12
6. Typing . . . 12
6.1 Type Checking . . . 13
6.2 Mini-Haskell - Syntax . . . 13
6.3 Typing . . . 13
6.4 Rules for Core λ-Calculus . . . 13
6.5 Type Inference . . . 14
6.6 Type Classes . . . 14
7. Algebraic Data Types . . . 15
7.1 Enumeration Types (disjoint unions) . . . 15
7.2 Product Types . . . 15
7.3 General Definition . . . 16
7.4 Recursive Types . . . 16
7.5 Correctness for Algebraic Data Types . . . 16
8. Lazy Evaluation . . . 16
8.1 Evaluation Strategy . . . 16
8.2 Correctness of lazy Programs . . . 17

Formal Methods **18**
1. Introduction . . . 18
2. Introduction to Language Semantics . . . 18
2.1 The Language IMP . . . 18
2.2 Semantics of IMP Expressions . . . 19
2.3 Properties of the Expression Semantics . . . 20
Substitution . . . 20
3. Operational Semantics . . . 20
3.1 Big-Step Semantics . . . 21

3.2 Small-Step Semantics 24
3.3 Equivalence 26
4. Axiomatic Semantics 27
4.1 Motivation 27
4.2 Hoare Logic 27
4.3 Soundness and Completeness 28
5. Modeling 29
5.1 Protocol Meta Language Promela 29
5.2 Motivation 31
5.3 More on Promela 32
6. Linear Temporal Logic 33
6.1 Linear-Time Properties 33
6.2 Linear Temporal Logic 34
7. Model Checking 35

Functional Programming

1. Introduction

1.1 Basic Concepts of Functional Programming

One important notion in functional programming is that functions have no side effect, i.e. $f(x)$ always returns the same value (for the same x). This allows us to reason as in mathematics, i.e. if $f(0) = 2$ then $f(0) + f(0) = 2 + 2$.

The above mentioned property is called `referential transparency`, i.e. *an expression evaluates to the same value in every context.*

Another basic concept is that we use `recursion instead of iteration`. This can be shown with an easy example of `gcd`. In Java we might use an iterative function:

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

whereas in functional programming we strictly use recursion:

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

In this example we can already see the main property of functional programming - it specifies **what** should be computes, rather than **how**.

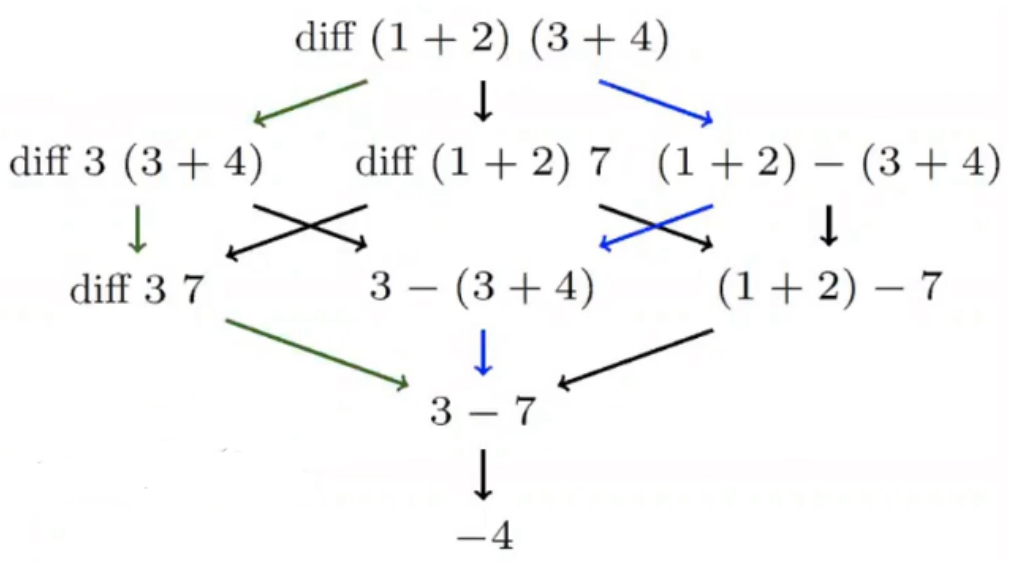
1.2 Introduction to Functional Programming

Expression Evaluation

In general, expression evaluation in Haskell works the same way as in mathematics, e.g. for $f(x, y) = x - y$, if we want to compute $f(5, 7)$ we substitute 5 for x and 7 for y.

We differ between two types of `evaluation strategies`:

- `Eager evaluation`, where we evaluate arguments first (corresponds to the *green path* in the picture below)
- `Lazy evaluation`, which is used in Haskell, where we evaluate expression from the left and *only when needed* (corresponds to the blue path in the picture below)



Syntax and Types

The basic syntax of Haskell consists of the following two rules: - Functions consist of different cases:

```
functionName x1 ... x2
    | guard1 = expr1
    | guard2 = expr2
    ...
    | guardm = exprm
```

- Programs consist of several definitions:

```
myConstant = 5

aFunction y1 ... ym
    | guard1 = expr1
    | guard2 = expr2

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

In Haskell, **indentation determines separation of definitions**:

- All function definitions must start at the same indentation level.
- If a definition requires $n > 1$ lines, we indent the lines 2 to n further.

Spaces are therefore very important, one should ***not use tabs!***

Haskell is a strongly typed language, i.e. we can only assign values to variables of a certain type. This helps us to avoid runtime errors like `3 + True`. In Haskell we can use the following `types`:

- `Int` for integers with at least the range of $\{-2^{29}, ..., 2^{29} - 1\}$ and the functions `+, *, ^, -, div, mod, abs`
- `Integer` for unbounded integers and the same functions as `Int`
- `Bool` with the values `True, False` and the binary operators `&&, ||` and the unary operator `not`
- `Char` for single characters as expected, i.e. `'a', 'b', 'c', ...`
- `String` for strings as expected, i.e. `"hello", "world", ...`
- `Doubles` for double precision numbers and functions like `+, -, *, /, abs, acos, ...`

We furthermore examine the type `tuple` more in detail. Tuples are represented in `()` brackets. We can put as many elements in a tuple as we want, we can even put tuples in tuples.

```
(Type1, Type2, ..., TypeM)
```

Functions can take tuples as arguments and / or return tupled values as shown in the example below:

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

Pattern and Function Definition

Function definitions build from both patterns xi and guards gi.

```
functionName x1 ... x2
    | g1 = expr1
    | g2 = expr2
    ...
    | gm = exprm
```

Patterns are variables, constants or built from data constructors, while guards are boolean expressions. We need to note that pattern matching forces evaluation.

Function Scope

Functions have a `global scope`, i.e. a function can be called from any other function.

We can force a `local scope` with the keywords `let` and `where` as shown in the example below:

```
f x = let sq y = y * y
      in sq x + sq x
```

The keyword `where` comes directly after a function definition and is used to define bindings over all guards:

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

2. Natural Deduction

To carry out `formal reasoning` about systems 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

The formal definition is given by: - Let a set $\mathcal{V}$ of variables be given. Then $\mathcal{L}_P$, 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 $B \in \mathcal{L}_P$.

Semantics

A `valuation` $\sigma : \mathcal{V} \to \{\text{True, False}\}$ is a function mapping variables to truth values. We furthermore let `Valuations` be the set of valuations.

`Satisfiability` describes the smallest relation $\vDash \subseteq \text{Valuations} \times \mathcal{L}_P$ such that:

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

We note here that $\sigma \nvDash \bot$, for every $\sigma \in \text{Valuations}$.

A formula $A \in \mathcal{L}_P$ is `satisfiable` if

$$\sigma \vDash A, \text{ for some valuation } \sigma$$

A formula $A \in \mathcal{L}_P$ is `valid` (a `tautology`) if

$$\sigma \vDash A, \text{ for all valuations } \sigma$$

We furthermore respect `semantic entailment`, that is, $A_1, ..., A_n \vDash A$ if for all σ, if $\sigma \vdash A_1, ..., \sigma \vDash A_n$, then $\sigma \vDash A$.

Requirement for a Deductive System

For a deductive system we require that *syntactic entailment* $\vdash$ (derivation rules) and *semantic entailment* $\vDash$ (truth tables) agree. This requirement has two parts. For $H \equiv A_1, ..., A_n$ some collection of formulae:

1. `Soundness`: If $H \vdash A$ can be derived, then $H \vDash A$
2. `Completeness`: If $H \vDash A$, then $H \vdash A$ can be derived

Further `Decidability` is desirable.

Natural Deduction for Propositional Formulae

We define three keywords for natural deduction:

- `Sequent`: An assertion of the form $A_1, ..., A_n \vdash A$ where all A, A_1, $A_2, ..., A_n$ are propositional formulae
- `Axiom`: A starting point for building derivation trees of the form

$$\frac{}{..., A, ... \vdash A}\ axiom$$

- `Proof` (of A): A derivation tree with root $\vdash A$

Rules

We distinguish between two kinds of `rules`:

- `introduce`, denoted by $-I$, which introduces a connective
- `eliminate`, denoted by $-E$, which eliminates a connective

Conjunction Rules

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

Implication Rules

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

Disjunction Rules

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

Falsity Rule

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

Negation Rule

$$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash B} \neg - E$$

2.2 First-order logic

Syntax

There are two syntactic categories: `terms` and `formulae`.

Furthermore, a `signature` consists of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$ and we also denote the set of variables as $\mathcal{V}$.

Term, the `terms of first-order logic`, is the smalles set where:

1. $x \in$ Term if $x \in \mathcal{V}$, and
2. $f^n(t_1, ..., t_n) \in$ Term if $f^n \in \mathcal{F}$ and $t_j \in$ Term for all $1 \leq j \leq n$

Form, the `formulae of first-order logic`, is the smallest set where:

1. $\bot \in$ Form
2. $p^n(t_1, ..., t_n) \in$ 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 $\circ \in \{\wedge, \vee, \rightarrow\}$
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 either `bound` or `free`. A variable occurrence x in a formula A is `bound` if x occurs within a subformula B of A of the form $\exists x.B$ or $\forall x.B$ and is said to be `free` otherwise.

α - Conversion Names of bound variables are irrelevant, the just encode the binding structure. Therefore we can rename *bound* variables at any time (called α `-conversion`).

Example:

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

Omitting Parentheses For binary operators we have the following binding strengths:

- $\wedge$ binds stronger than $\vee$ binds stronger than $\rightarrow$
- $\rightarrow$ associates to the right, $\wedge$ abd $\vee$ bind to the left
- $\neg$ binds stronger than any binary operator
- Quantifiers extend to the right as far as possible, that is, the end of the line or ")"

Semantics

A `structure` is a pair $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$ where $U_\mathcal{S}$ is a non-empty 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 shorthand, we may also 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 $v : \mathcal{V} \rightarrow U_\mathcal{S}$ a valuation.

The `value` of a term t under the interpretation $\mathcal{I} = \langle \mathcal{S}, v \rangle$ is written as $\mathcal{I}(t)$ and defined by:

1. $\mathcal{I}(x) = v(x)$, for $x \in \mathcal{V}$, and
2. $\mathcal{I}(f(t_1, ..., t_n)) = f^\mathcal{S}(\mathcal{I}(t_1), ..., \mathcal{I}(t_n))$

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.

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.

Following an example of a suitable model for

$$\forall x.p(x,\, s(x))$$

- $U_{\mathcal{S}} = \mathcal{N}$
- $p^{\mathcal{S}} = \{(m,\, n) \mid m, n \in U_{\mathcal{S}} \text{ and } m < n\}$
- $s^{\mathcal{S}}(x) = x + 1$

Substitution

`Substitution` describes replacing in A all occurrences of a *free variable* x with some term t. We write $A[x \mapsto t]$ to indicate that we substitute x by t in A.

Example:

$$A \equiv \exists y.y * x = x * z \to A[x \mapsto 2 - 1] \equiv \exists y.y * (2 - 1) = (2 - 1 * z)$$

Rules

We will continue to use rules from propositional logic, but now for first-order formulae.

Universal Quantification

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\,\forall - I^*, \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x/t]}\,\forall - E$$

For the insertion rule, the side condition * denotes that x cannot be free in any assumption Γ.

Existential Quantification

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

For the elimination rule, the side condition * denotes that x is neither free in B nor free in Γ.

2.4 Equality

First order logic with equality

Equality is logical symbol with associated proof rules. We define it with:

- Extended language: $t_1 = t_2 \in Form$ if $t_1, t_2 \in Term$
- Extended definition of $\vDash$: $\mathcal{I} \vDash t_1 = t_2$ if $\mathcal{I}(t_1) = \mathcal{I}(t_2)$

Equality

`Equality` is an equivalence rule, shown by the following three rules:

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

Equality is also a `congruence` on terms and all definable relations:

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

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

3. Correctness

Correctness is important for programs. But what does `correctness` mean? What properties should hold?

- `Termination`: Important for many, but not all programs
- `Functional behavior`: Function should return "correct" values

Correctness is rarely obvious, it must be proven!

3.1 Termination

If f is defined in terms of functions $g_1, ..., g_k$, and each g_j terminates, then so does f. If we work with recursion, a sufficient condition for termination is: $>$ Arguments are smaller along a well-founded order of function's domain. $>>$ - An order $>$ on a set $\mathcal{S}$ is `well-founded` iff. there is no infinite decreasing chain chain $x_1 > x_2 > x_3 > \cdots$, for $x_i \in \mathcal{S}$ $>$ - We write $>_{\mathcal{S}}$ to indicate the domain $\mathcal{S}$, i.e. $>_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$

Well-Founded Relations

We can construct new well-founded relations from existing ones the following way:

Let R_1 and R_2 be binary relations on a set S. The composition of R_1 and R_2 is defined as:

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

Remark: For a binary relation R, we write $a\, R\, b$ for $(a, b) \in R$.

Let $R \subseteq S \times S$. Define:

- $R^1 \equiv R$
- $R^{n+1} \equiv R \circ R^n$, for $n \geq 1$
- $R^+ \equiv \bigcup_{n \geq 1} R^n$

So $a\, R^+\, b$ iff. $a\, R^i\, b$ 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 $s_1, .., s_{i-1} \in S$ such that $s_0\, R\, s_1\, R \ldots R\, s_{i-1}\, R\, s_i$.
>
> Theorem: If $>$ is a well-founded order on a set S, then $>^+$ is also well-founded on S.

3.2 Reasoning

Equational Reasoning

`Equational reasoning` include proofs based on the simple idea, that functions are equations.

For example we can have the following simple Haskell program:

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

More formally:

$$\forall a \in \mathcal{Z}, \forall b \in \mathcal{Z}, \text{swap}(a, b) = (b, a)$$

Reasoning by Cases

Consider the following Haskell snippet:

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

Can we prove that maxi $n\, m \geq n$?

- We have that $n \geq m \vee \neg(n \geq m)$
- We now show maxi $n\,m \geq n$ for both cases:
  - Case 1: $n \geq m$, then maxi $n\,m = n$ and $n \geq n$
  - Case 2: $\neg(n \geq m)$, then maxi $n\,m = m$. But $m > n$, so maxi $n\,m \geq n$

Proof by Induction

We use a domino principle formulated by induction proof rule.

Example: To prove $\forall n \in \mathcal{N}.P$

- Base case: Prove $P[n \mapsto 0]$
- Step case: For an arbitrary m not free in P, prove $P[n \mapsto m+1]$ under the assumption $P[n \mapsto m]$.

Well-Founded Induction

The induction schema for well-founded induction is given by:

- To prove P for all natural numbers n
- Well-founded step: For an arbitrary m (not free in P), prove $P[n \mapsto m]$ under the assumption that $P[n \mapsto l]$ holds, for all $l < m$ (where also l is not free in P).

4. List and Abstraction

4.1 List Type

Lists require a new type constructor:

- If T is a type, then $[T]$ is a type.

The elements of $[T]$ are given by:

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

In Haskell we can use the following shorthand: $1 : (2 : (3 : []))$ written as [1, 2, 3].

When writing functions on lists we always have to consider the case of the empty list!

4.2 Patterns

Pattern matching has two purposes:

1. checks if an argument has the proper form
2. binds values to variables

Example: $(x : xs)$ matches with [2, 3, 4] $\rightarrow x = 2$, $xs = [3, 4]$.

Patterns are inductively defined with:

- Constants: -2, $'1'$, True, $[], ...$
- Variables: x, $foo, ...$
- Wild card: _
- Tuples: $(p_1, ..., p_k)$, where p_i are patterns
- Non-empty lists: $(p_1 : p_2)$, where p_i are patterns

Moreover, patterns require to be `linear`, this means that each variable can occur at most once.

4.3 Intermezzo - Advice on Recursion

Defining recursive functions can be difficult. We show a 5-step "tool" which should make the process easier. Following an example of defining a function `drop` which removes the first n elements of a list:

1. Step: Define the type

```
drop :: Int -> [Int] -> [Int]
```

2. Step: Enumerate the cases

```
drop 0 []     = ...
drop 0 (x:xs) = ...
drop n []     = ...
drop n (x:xs) = ...
```

3. Step: Define the simple cases

```
drop 0 []     = []
drop 0 (x:xs) = x:xs
drop n []     = []
drop n (x:xs) = ...
```

4. Step: Define the other cases

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

5. Step: Generalize and simplify

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

4.4 List Comprehension

We often need an analogous notation to set comprehension in set theory, i.e. an expression of the form $\{2 \cdot x \mid x \in X\}$. In Haskell we can use list comprehension to achieve this:

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

4.5 Induction over Lists

If we want to apply induction proofs to a function on lists, we need the following schema;

- Proof by Induction: to prove P for all xs in $[T]$
- Base case: prove $P[xs \mapsto []]$
- Step case: prove $\forall y :: T, ys :: [T]. \quad P[xs \mapsto ys] \implies P[xs \mapsto y : ys]$
  - Fix arbitrary $y :: T$ and $ys :: [T]$ (both not free in P)
  - Induction hypothesis: $P[xs \mapsto ys]$
  - To prove: $P[xs \mapsto y : ys]$

5. Abstraction

5.1 Polymorphic Types and Reusability

Until now we have only seen monomorphic types. For example:

```
[Int] -> Int, [String] -> Int, ...
```

In Haskell we can define polymorphic types that contain type variables (start with lower case letter).

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

5.2 Higher-Order Functions

`First-order functions` take base types or constructor types as arguments. While `second-order functions` take `first-order functions` as arguments and so on.

One example of such a higher-order functions are `map` and `foldr`.

```
map :: (a -> b) -> [a] -> [b]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

The advantages of functions as arguments is that we can increase reusability, definitions get easier to understand and correctness is simpler to show.

5.3 λ - Expressions

λ-`expressions` describes the convention to use or λ instead of $f(x)$, i.e., $f(x) = x + 3$ in conventional notation becomes $\lambda x.x + 3$ in λ-calculus.

We will look at the following two simple functions:

```
times2 x = 2 * x
atEnd x xs = xs ++ [x]
```

Haskell provides a notation to write functions like `times2` and `atEnd` **in-line**:

```
? map (\x -> 2 * x) [2, 3, 4]
[4, 6, 8]

? foldr (\x xs -> xs ++ [x]) [] [1, 2, 3, 4]
[4, 3, 2, 1]
```

5.4 Functions as Values

Functions can be returned as values. Example (`f . f` denotes the composition, i.e. $f \circ f$):

```
twice :: (t -> t) -> (t -> t)
twice f = f . f
-----------------------------
? > twice times3 1
9 :: Int
```

6. Typing

Again a friendly reminder that there are always hidden gems in the lecture slides / notes from when lecturers try their best to be funny.

6.1 Type Checking

`Type checking` should prevent "dangerous" expressions such as $2 + True$. The objectives for a type checker are as follows:

- quick, decidable, static analysis
- permit as much generality as possible
- prevents runtime errors

We examine here a simplified language: `Mini-Haskell`

6.2 Mini-Haskell - Syntax

Programs are terms t (we assume variables $\mathcal{V}$ and integers $\mathcal{Z}$ to be given):

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

We often employ syntactic sugar, like omitted parentheses, to make the syntax more readable.

6.3 Typing

Types are terms τ (we assume that $\mathcal{V}_T$ is a set of types variables: a, b, ...):

$$\tau ::= \mathcal{V}_T \mid \text{Bool} \mid \text{Int} \mid (\tau,\, \tau) \mid \tau \to \tau$$

We define a type system `notation` based on typing judgement: $\Gamma \vdash t :: \tau$ where:

- Γ is a set of bindings.
- t is a term
- τ is a type

Intuitively, one might read $x : \text{Int} \vdash x + 2 :: \text{Int}$ as "Given symbol table x : Int (Γ), then term $x + 2$ (t) has type Int (τ)".

6.4 Rules for Core λ-Calculus

We introduce the first three core rules:

Axiom

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

Abstraction

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

Application

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

Furthermore, there are some additional typing rules for Mini-Haskell:

Base Types

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

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

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

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

Tuples

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

6.5 Type Inference

Syntax-directed typing rules specify the algorithm for computing a type:

1. Start with the judgment $\vdash t :: \tau_0$ with the type variable τ_0.
2. Build the derivation tree bottom-up by applying rules.
3. Solve constraint (unification) to get possible types.

Example:

Type inference example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x :: ((\tau_3 \to Int), \tau_4)}\,Var}{\Gamma \vdash \textbf{fst}\ x :: \tau_3 \to Int}\,fst \qquad \dfrac{\dfrac{}{\Gamma \vdash 2 :: \tau_5}\,Int \qquad \dfrac{}{\Gamma \vdash True :: \tau_6}\,True}{\Gamma \vdash (2, True) :: \tau_3}\,Tuple}{\Gamma \vdash (\textbf{fst}\ x)\ (2, True) :: Int}\,App}{\overbrace{x : \tau_1}^{\Gamma=} \vdash \textbf{iszero}\ ((\textbf{fst}\ x)\ (2, True)) :: \tau_2}\,iszero}{\vdash \lambda x.\, \textbf{iszero}\ ((\textbf{fst}\ x)\ (2, True)) :: \tau_0}\,Abs$$

Constraints:

$\tau_0 = \tau_1 \to \tau_2$
$\tau_2 = Bool$
$\tau_1 = ((\tau_3 \to Int), \tau_4)$
$\tau_3 = (\tau_5, \tau_6)$
$\tau_5 = Int$
$\tau_6 = Bool$

Most general type:

$\tau_0 = (((Int, Bool) \to Int), a) \to Bool$

Exercise:

Infer the type of
$\lambda x.\, \lambda y.\, (\textbf{iszero}\ (x\ y),\ x\ 3)$

6.6 Type Classes

Monomorphic versus Polymorphic

Some functions are `monomorphic`. For example, the function XOR, which can only be applied to Bool . In contrast, some functions are `polymorphic`, for example a function which can be applied to lists of any type.

Definition of the Eq Class

The `equality class` is defined as:

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

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

The definition includes the *class name*, the *signature* (the list of function names and types), and the default implementations. Elements of the class are called `instances`.

Classes allow restricted form of type generalization with class constraints. For example the function `elem` checks if something is an element of a list.

```
elem :: Eq t => t -> [t] -> Bool
```

```
elem _ []     = False
elem a (x:xs) = (a == x) || elem a xs
```

Instances

We can create an instance the following way:

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

Now the type `Bool` is an instance of the `Eq` class and we can use the functions of the type class.

Class Hierarchies

Classes can be hierarchically structured. For example, the class `Ord` is a subclass of the class `Eq`.

```
class (Eq a) => Ord a where ...
```

Classes can even "inherit" from multiple classes.

```
class (Num a, Ord a) => Real a where ...
```

7. Algebraic Data Types

With `algebraic data types` we can declare new types tailored to the objects being modeled. For months for example, we declare the type *Month* with elements *January, February,..., December*. These are new `data constructors`.

7.1 Enumeration Types (disjoint unions)

For example:

```
data Season = Spring | Summer | Fall | Winter
data Month  = January | February | ... | December
```

For the syntax, the following holds:

- Starts with the keyword `data`
- Names different constructors
- First letter of each constructor must be upper-case

7.2 Product Types

A `product type` looks as follows:

```
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 "John" 30
```

Enumeration and product types can be `combined`:

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

i.e., we define a `Shape` as either a circle with a radius or a rectangle with two sides. Functions are definable by *pattern matching*:

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

These new types can again be used with type classes to gain some functionality. In some cases, class instances can be automatically derived.

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

7.3 General Definition

The general definition looks as follows:

```
data T = Constr_1 T_11 ... T_1k
       | Constr_2 T_21 ... T_2k'
         ...
       | Constr_n T_n1 ... T_nk''
```

where T_{ij} are types, possibly also containing T.

7.4 Recursive Types

Set of objects are often recursively defined:

$$Expr ::= Int \,|\, Expr + Expr \,|\, Expr - Expr$$

Formalized as a recursive data type:

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

An example of recursive functions over data types would be an `interpreter for arithmetic expressions`, which could look as follows:

```
eval :: Expr -> Int

eval (Lit n)     = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

7.5 Correctness for Algebraic Data Types

Structural Inducation

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

We want to perform induction over the structure of terms.

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

Proof: We show $\forall x \in \text{Tree } t.P$ by induction.

- Base case: Show $P[x \mapsto \text{Leaf}]$
- Step case: Let $a \in t$ and $l, r \in \text{Tree } t$ be arbitrary.
  - Assume $P[x \mapsto l]$ and $P[x \mapsto r]$.
  - Show $P[x \mapsto \text{Node } a\, l\, r]$

8. Lazy Evaluation

8.1 Evaluation Strategy

Lazy Evaluation

Haskell has a `lazy evaluation` strategy, which means that expressions are evaluated only when necessary. This happens top-down (outermost operator first) and from left to right, depending on operator precedence.

Example:

$$f(9-3)(f\ 34\ 3) = (9-3) + (f\ 34\ 3) = 6 + (f\ 34\ 3) = 6 + (34+3) = 6 + 37 = 43$$

In Haskell, substitution occurs without argument evaluation. A potential problem with lazy evaluation is that computations may be duplicated. However, function arguments are evaluated only when needed and *at most once*, using `sharing`, i.e. sharing common sub-terms via pointers.

Evaluation - Pattern Matching

Arguments are evaluated as far as needed to determine a pattern match. Example:

```
f []     _      = 0          -- (f.1)
f _      []     = 0          -- (f.2)
f (a:_) (b:_) = a + b       -- (f.3)
```

On execution of `f [1 .. 3] [4 .. 6]`, the following evaluation happens:

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

8.2 Correctness of lazy Programs

Induction is only sound for finite, everywhere defined data. Thus we will only consider these type of programs for correctness proofs and not programs that lazyly generate infinite data.

Formal Methods

`Formal methods` are mathematical approaches to software and system development which support the rigorous specification, design, and verification of computer systems.

1. Introduction

We need to set `specifications` for programs. We need to use mathematical notations to describe:

- System design to accomplish these requirements, e.g. program code
- Requirements for the system, i.e. desired properties, e.g. deadlock freedom
- Assumptions about the environment, e.g. intruder model

We need to use a formal logic for `verification` to:

- Validate specifications by checking consistency
- Prove that the design satisfies requirements under given assumptions
- Deductive via proof system or algorithmic via model checking

The following `limitations` we see with formal methods:

- Incorrect specifications
- Technical limitations (undecidable properties, computing resources)
- Many applications of formal methods require specialist users
- Application of formal methods is expensive

2. Introduction to Language Semantics

2.1 The Language IMP

The core language `IMP` is very simple. It has:

- Boolean and arithmetic `expressions`
- `Variables` which range over integers and are initialized
- IMP does not include:
  - Heap allocation and pointers
  - Variable declarations
  - Procedures
  - Concurrency

Concrete Syntax of IMP

`Characters`:

```
Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
Digit  = '0' | '1' | ... | '9'
```

`Tokens`:

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

`Arithmetic Expressions`:

```
Aexp = '(' Aexp Op Aexp ')'
     | Var
     | Numeral

Op   = '+' | '-' | '*'
```

`Boolean Expressions`:

```
Bexp = '(' Bexp 'or' Bexp ')'
     | '(' Bexp 'and' Bexp ')'
     | 'not' Bexp
     | Aexp Bexp Aexp

Rop  = '=' | '#' | '<' | '<=' | '>' | '>='
```

Note that `#` is not equal.

`Statements`:

```
STM = 'skip'
    | Var ':=' Aexp
    | '(' STM ';' Stm ')'
    | 'if' Bexp 'then' Stm 'else' Stm 'end'
    | 'while' Bexp 'do' Stm 'end'
```

Meta-Variables

`Meta-variables` denote an arbitrary element of a syntactic category, e.g. an arbitrary statement.

We follow the naming conventions below:

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

2.2 Semantics of IMP Expressions

`Semantic functions` map elements of syntactic categories to elements of semantic categories. For example, we need a function that maps from Numerals to Val.

Example:

- 101 to 5 (Binary to Decimal)
- 101 to 101 (Straight forward, no conversion)

Semantic of Numerals

The semantic function

$$\mathcal{N} : \text{Numeral} \to \text{Val} = \mathbb{Z}$$

maps a numeral n to an integer value $\mathcal{N}[[n]]$.

Examples:

$$\mathcal{N}[[8]] = 8$$
$$\mathcal{N}[[n\ 9]] = \mathcal{N}[[n]] \times 10 + 9$$

A `state` is a total function, associating a value with each program variable:

$$\text{State} : \text{Var} \to \text{Val}$$

We use σ as a meta-variable for states. We define a designated state σ_{zero}, in which all variables have the value 0. States can be updated with a function $\sigma[y \to v]$:

$$(\sigma[y \to v])(x) = v \text{ if } x \equiv y, \quad \sigma(x) \text{ else}$$

Two states σ_1 and σ_2 are said to be `equal` if they are equal functions:

$$\sigma_1 = \sigma_2 \iff \text{if } \forall x.\sigma_1(x) = \sigma_2(x)$$

Semantics of Arithmetic Expressions

The semantic function

$$\mathcal{A} : \text{Aexp} \to \text{State} \to \text{Val}$$

maps an arithmetic expression e and a state σ to a value $\mathcal{A}[[e]]\sigma$. Example:

$$\mathcal{A}[[x]]\sigma = \sigma(x)\mathcal{A}[[e_1 \text{ op } e_2]]\sigma = \mathcal{A}[[e_1]]\sigma \ \overline{\text{op}} \ \mathcal{A}[[e_2]]\sigma \quad \text{for op} \in \text{Op}$$

Note: $\overline{op}$ is the operation $\text{Val} \times \text{Val} \to \text{Val}$ corresponding to op.

Semantics of Boolean Expressions

The semantic function

$$\mathcal{B} : \text{Bexp} \to \text{State} \to \text{Bool}$$

maps a boolean expression b and a state σ to a truth value $\mathcal{B}[[b]]\sigma$. Example:

$$\mathcal{B}[[e_1 \text{ op } e_2]]\sigma = tt \text{ if } \mathcal{A}[[e_1]]\sigma \ \overline{\text{op}} \ \mathcal{A}[[e_2]]\sigma, \quad ff \text{ otherwise}$$

with op $\in$ Rop.

2.3 Properties of the Expression Semantics

Inductive Definitions

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.

Example: Assume a new arithmetic expression: $-e$. A inductive definition of $\mathcal{A}[[-e]]\sigma$ is given by:

$$\mathcal{A}[[-e]]\sigma = 0 - \mathcal{A}[[e]]\sigma$$

since e is a `subterm` of $-e$. The definition $\mathcal{A}[[-e]]\sigma = \mathcal{A}[[0 - e]]\sigma$ is not inductive since $0 - e$ is not a subterm of $-e$.

Lemma: The equations for $\mathcal{N}$ define a `total function` $\mathcal{N}$: Numeral $\to$ Val.

Substitution

Substitution $[x \mapsto e]$ replaces each free occurrence of variable x by e. We use the following substitution lemma.

Lemma: $\mathcal{B}[[b[x \mapsto e]]]\sigma \iff \mathcal{B}[[b]]\sigma[x \mapsto \mathcal{A}[[e]]\sigma]$

3. Operational Semantics

Operational semantics describe *how* the state is modified during the execution of a statement.

3.1 Big-Step Semantics

Natural Semantics of IMP

A `transition system` is a tuple $(\Gamma,\ T,\ \rightarrow)$, where

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

`Operational semantics` includes two types of configurations:

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

The `transition relation` $\rightarrow$ describes how executions take place. We specify transition relations by rules of the form

$$\frac{\phi_1 \cdots \phi_n}{\psi}(\text{Name})^*$$

where $\phi_1 \cdots \phi_n$ and ψ are transitions. The meaning of the rules is:

If $\phi_1 \cdots \phi_n$ are transitions, then ψ is a transition.

We use the following terminology:

- $\phi_1 \cdots \phi_n$ are called the `premises` of the rule.
- ψ is called the `conclusion` of the rule.

Big-Step Semantics of IMP

Skip

$$\frac{}{\langle \text{skip},\ \sigma\rangle \rightarrow \sigma}(\text{SKIP}_{NS})$$

Assign

$$\frac{}{\langle x := e,\ \sigma\rangle \rightarrow \sigma[x \rightarrow \mathcal{A}[[e]]\sigma]}(\text{ASS}_{NS})$$

Sequential Composition

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

Conditional Statement

$$\frac{\langle s,\ \sigma\rangle \rightarrow \sigma'}{\langle \text{ if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \rightarrow \sigma'}(\text{IFT}_{NS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = tt$$

$$\frac{\langle s',\ \sigma\rangle \rightarrow \sigma'}{\langle \text{ if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \rightarrow \sigma'}(\text{IFF}_{NS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = ff$$

Loop Statement

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

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

Rule Schemes and Instantiations Inference rule definitions are actually `rule schemes`. A rule is `instantiated` when all meta-variables are replaced with syntactic elements. For example, consider the assignment rule *scheme*

$$\frac{}{\langle x := e,\, \sigma\rangle \to \sigma[x \to \mathcal{A}[[e]]\sigma]}(\text{ASS}_{NS})$$

and a corresponding assignment rule instance:

$$\frac{}{\langle v := v + 1,\, \sigma_{\text{zero}}\rangle \to \sigma_{\text{zero}}[v \to 1]}(\text{ASS}_{NS})$$

Derivation Trees Rule instances can be combined to derive a transition $\langle s,\, \sigma\rangle \to \sigma'$. The result of the derivation is a `derivation tree` T.

- The root of T is $\langle s,\, \sigma\rangle \to \sigma'$. written as $\text{root}(T) \equiv \langle s,\, \sigma\rangle \to \sigma'$.
- The leaves of T are axiom rule instances.

The transition system permits a transition $\langle s,\, \sigma\rangle \to \sigma'$, written as $\vdash \langle s,\, \sigma\rangle \to \sigma'$, if and only if there exists a `finite derivation tree` ending in $\langle s,\, \sigma\rangle \to \sigma'$.

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 $\exists\sigma'$ such that $\vdash \langle s,\, \sigma\rangle \to \sigma'$
- `fails to terminate` iff there is no state σ' such that $\vdash \langle s,\, \sigma\rangle \to \sigma'$

3.1.2 Proving Properties of the Semantics

Semantic Equivalence

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

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

Unfolding Loops in IMP We have the following two lemmas:

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

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

Deterministic Semantics

Lemma: The big-step semantics of IMP is `deterministic`.

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

Induction on Derivation Trees

We introduce a new proof technique: `induction on the shape of derivation trees`.

To prove a property $P(T)$ for all derivation trees T, prove that $P(T)$ holds for an arbitrary derivation tree T under the assumption (*I.H.*) that $P(T')$ holds for all *sub-trees* T' of T.

Such a proof typically proceeds by case distinction on the rule applied at the root of the arbitrary derivation tree.

3.1.3 Extensions of IMP

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

The following semantics hold:

1. Expression e is evaluated in the initial state
2. Statement s is executed in a state in which x has the value of e
3. After the execution of s, the initial value of x is restored

Big-step semantics rule:

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

Procedure Declarations and Calls We look at statements of the form:

$$\text{procedure}(x_1, \cdots, x_n;\, y_1, \cdots, y_m) \text{ begin } s \text{ end}$$

We have the following formal parameters:

- $x_1, \cdots, x_n$ are value parameters
- $y_1, \cdots, y_m$ are variable parameters (used to assign values back to the procedure caller)

Procedures can be declared as part of a source program, and called. We apply the following restrictions:

- In a procedure declaration, the formal parameter names $x_1, \cdots, x_n$, $y_1, \cdots, y_m$ must be distinct from each other
- $x_1, \cdots, x_n$, $y_1, \cdots, y_m$ are the only free variables in s

The big-step semantics rule is given by:

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

Abort The idea is that the statement `abort` stops the execution of the complete program. Aborting is modeled in the operational semantics by ensuring that the configurations $\langle \text{abort},\, \sigma\rangle$ are stuck, that is, that there is no state σ' such that $\langle \text{abort},\, \sigma\rangle \to \sigma'$.

abort and skip are not semantically equivalent since there is a derivation tree for $\langle \text{skip},\, \sigma\rangle \to \sigma$, but not for $\langle \text{abort},\, \sigma\rangle \to \sigma'$.

abort and while true do skip end are semantically equivalent however!

Non-determinism The idea is that for the statement $s[]s'$ either s or s' is non-deterministically chosen to be executed.

Example: The statement

$$x := 1[](x := 2;\, x := x + 2)$$

will result in a state in which x either has the value 1 or 4.

The following rules apply:

$$\frac{\langle s,\, \sigma\rangle \to \sigma'}{\langle s[]s',\, \sigma\rangle \to \sigma'}(\text{ND1}_{\text{NS}}) \qquad \frac{\langle s',\, \sigma\rangle \to \sigma'}{\langle s[]s',\, \sigma\rangle \to \sigma'}(\text{ND2}_{\text{NS}})$$

Parallelism The idea is that for the statement s par s' both statements s and s' are executed, but execution can be interleaved.

Example: The statement

$$x := 1 \text{ par } (x := 2;\ x := x + 2)$$

could result in a state in which x has the value 4, 1, or 3.

3.2 Small-Step Semantics

3.2.1 Structural Operational Semantics of IMP

Structural Operational Semantics (SOS) `Small-step semantics` focuses attention on the individual steps of an execution. For example:

- Execution of assignments
- Execution of if-conditions, while-iterations, etc.

Describing small steps of the execution allows one to express the *order of execution* of individual steps.

The configurations are the same as for natural semantics ($\langle s, \sigma\rangle$ or σ). We may use γ as a meta-variable for terminal or non-terminal configurations.

The transition relation $\to_1$ can have two forms:

- $\langle s, \sigma\rangle \to_1 \langle s'. \sigma'\rangle$: 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 of the form $\langle s, \sigma\rangle \to_1 \gamma$ describes the first step of the execution of s in state σ.

Transition System

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

$$T = \text{State}$$

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

We say that a non-terminal configuration $\langle s, \sigma\rangle$ is `stuck` if there does not exist a configuration γ such that $\langle s, \sigma\rangle \to_1 \gamma$. We note here that terminal configurations (i.e- final states) σ are never stuck.

SOS of IMP We have the following small-step semantics rules:

Skip

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

Assign

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

These rules are analogous to natural semantics.

Sequential Composition

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s; s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}(\text{SEQ1}_{SOS}) \qquad \frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s; s', \sigma\rangle \to_1 \langle s''; s', \sigma'\rangle}(\text{SEQ2}_{SOS})$$

Conditional Statement

The first step of executing if b then s_1 else s_2 end is to determine the outcome of the test b, and thereby, which branch to select:

$$\frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \to_1 \langle s, \sigma\rangle}(\text{IFT}_{SOS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = tt$$

$$\frac{}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \to_1 \langle s', \sigma\rangle}(\text{IFF}_{SOS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = ff$$

However, there are alternative rules for conditional statements, where the first step of executing if b then s_1 else s_2 end is the first step of the branch determined by the outcome of the test b:

$$\frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \to_1 \sigma'}(\text{IFT1}_{SOS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = tt$$

$$\frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle \text{if } b \text{ then } s \text{ else } s' \text{ end}, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}(\text{IFT2}_{SOS}) \qquad \text{if } \mathcal{B}[[b]]\sigma = tt$$

Loop Statement

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

Multi-Step Executions A relation is a k-step execution, written as $\gamma \to_1^k \gamma'$. The intuitive meaning of this is that there is an execution from γ to γ' in exactly k steps. We define the relation $\gamma \to_1^k \gamma'$ as follows:

- $\gamma \to_1^0 \gamma'$ 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 $\gamma'' \to_1^{k-1} \gamma'$

We may use the notation $\gamma \to_1^* \gamma'$ to denote that there is an execution from γ to γ' in some finite number of steps.

Derivation Sequences A `derivation sequence` is a non-empty, finite or infinite, sequence of configurations $\gamma_0, \gamma_1, \gamma_2, ...$ 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 or a stuck configuration

Note that if $\gamma_0, \gamma_1, \gamma_2, ...$ is a derivation sequence, then, for all i in the range of the sequence, $\gamma_0 \to_1^i \gamma_i$.

Termination The execution of a statement s in state σ

- `terminates` iff there is a finite derivation sequence starting with $\langle s, \sigma\rangle$
- `terminates successfully` iff $\exists \sigma'. \langle s, \sigma\rangle \to_1^* \sigma'$
- `runs forever` iff there is an infinite derivation sequence starting with $\langle s, \sigma\rangle$

3.2.2 Proving Properties of the Semantics

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 :

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

Semantic Equivalence The following lemma holds:

> Under the small-step semantics, two statements s_1 and s_2 are `semantically equivalent` if for all states σ, both:
>
> - for all stuck or terminal configurations γ, we have $\langle s_1, \sigma\rangle \to_1^* \gamma$ if and only if $\langle s_2, \sigma\rangle \to_1^* \gamma$, and
> - there is an infinite derivation sequence starting in $\langle s_1, \sigma\rangle$ if and only if there is one starting in $\langle s_2, \sigma\rangle$

Determinism The following lemma holds:

> The small-step semantics of IMP is `deterministic`. That is, for all s, σ, γ, and γ' we have that
>
> $$\vdash \langle s, \sigma\rangle \to_1 \gamma \wedge \vdash \langle s, \sigma\rangle \to_1 \gamma' \implies \gamma = \gamma'$$

3.2.3 Extensions of IMP

Local Variable Declarations A `local variable declaration` is of the form var $x :=$ e in s end.

The steps are:

1. Assign e to x
2. Execute s
3. Restore the initial value of x

The first small step could be easily defined as:

$$\frac{}{\langle \text{var } x := e \text{ in } s \text{ end}, \sigma\rangle \to_1 \langle s, \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]\rangle}$$

We extend the syntactic category Stm with a restore statement:

```
STM = ... | 'restore' (Var, Val)
```

Now we can use the restore statement to mark the end of the scope of a local variable and remember its original value:

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

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

Abort The `abort` statement stops the execution of the complete program. Aborting is modeled by ensuring that the configurations $\langle$abort, $\sigma\rangle$ are stuck.

Non-Determinism For the statement $s[]s'$ either s or s' is non-deterministically chosen to be executed. We have the following rules:

$$\frac{}{\langle s[]s', \sigma\rangle \to_1 \langle s, \sigma\rangle}(\text{ND1}_{SOS}) \qquad \frac{}{\langle s[]s', \sigma\rangle \to_1 \langle s', \sigma\rangle}(\text{ND2}_{SOS})$$

Parallelism For the statement s par s' both statements s and s' are executed, but the execution may be interleaved:

$$\frac{\langle s, \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \text{ par } s', \sigma\rangle \to_1 \langle s'' \text{ par } s', \sigma'\rangle}(\text{PAR1}_{SOS}) \quad \frac{\langle s, \sigma\rangle \to_1 \sigma'}{\langle s \text{ par } s', \sigma\rangle \to_1 \langle s', \sigma'\rangle}(\text{PAR2}_{SOS})$$

$$\frac{\langle s', \sigma\rangle \to_1 \langle s'', \sigma'\rangle}{\langle s \text{ par } s', \sigma\rangle \to_1 \langle s \text{ par } s'', \sigma'\rangle}(\text{PAR2}_{SOS}) \quad \frac{\langle s', \sigma\rangle \to_1 \sigma'}{\langle s \text{ par } s', \sigma\rangle \to_1 \langle s, \sigma'\rangle}(\text{PAR2}_{SOS})$$

3.3 Equivalence

The following theorem holds: > For every statement s of IMP, we have: >

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

Furthermore, we introduce the following two lemmas: > For every statement s of IMP and states σ and σ' we have: >

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

> For every statement s of IMP, states σ and σ', and natural number k we have that:
>
> $$\langle s, \sigma\rangle \to_1^k \sigma' \implies \vdash \langle s, \sigma\rangle \to \sigma'$$

4. Axiomatic Semantics

4.1 Motivation

We can prove correctness of programs based on formal semantics. The proof would also be possible with small-step semantics, but even more complicated. Such a proof is too detailed to be practical. Axiomatic semantics provides a way of constructing these proofs conveniently.

4.2 Hoare Logic

4.2.1 Hoare Triples and Assertions

The properties of programs are specified as `Hoare Triples` of the form

$$\{P\}\, s\, \{Q\}$$

where s is a statement and P and Q are assertions about the state.

Terminology:

- The assertion P is called the `precondition`
- The assertion Q is called the `postcondition`

This means that 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'.

Logical Variables We allow assertions to contain `logical variables`:

- Logical variables may occur only in assertions
- Logical variables are not program variables and may, thus, not be accessed in programs

For example:

```
{ x = N }
    y := 1; while not x = 1 do y := y*x; x := x-1 end
{ y = N! AND N > 0 }
```

4.2.2 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 from each statement
- The premises and conclusions of the derivation rules are Hoare triples

Similarly to the other derivation systems we have studied, we write $\vdash \{P\}\, s\, \{Q\}$ if and only if there exists a finite derivation tree ending in $\{P\}\, s\, \{Q\}$

Axiomatic Semantics of IMP Skip

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}(\text{SKIP}_{Ax})$$

Assign

$$\frac{}{\{P[x \to e]\}\, x := e\, \{P\}}$$

Sequential Composition

$$\frac{\{P\}\, s\, \{Q\} \quad \{Q\}\, s'\, \{R\}}{\{P\}\, s;\, s'\, \{R\}}(\text{SEQ}_{Ax})$$

Conditional Statement

$$\frac{\{b \wedge P\}\, s\, \{Q\} \quad \{\neg b \wedge P\}\, s'\, \{Q\}}{\{P\}\ \text{if } b \text{ then } s \text{ else } s' \text{ end}\, \{Q\}}(\text{IF}_{Ax})$$

Loop Statement

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

The rules so far manipulate assertions *syntactically*. `Semantic entailment` expresses semantic reasoning steps:

- We write $P \vDash Q$ iff "for all states σ, $\mathcal{B}[[P]]\sigma = tt$ implies $\mathcal{B}[[Q]]\sigma = tt$.

The `rule of consequence` allows semantic entailments in derivations:

$$\frac{\{P'\}\, s\, \{Q'\}}{\{P\}\, s\, \{Q\}} (\text{CONS}_{Ax}) \quad \text{if } P \vDash P' \text{ and } Q' \vDash Q$$

4.2.3 Proving Properties of the Semantics

Induction on the Shape of Derivation Trees Properties of the axiomatic semantics are typically proved by *induction on the shape of the derivation tree*. Those proofs typically proceed by case distinction on the rule applied to the root of the arbitrary derivation tree T.

Semantic Equivalence Two statements s_1 and s_2 are `provably equivalent` if:

$$\forall P, Q\,.\, \vdash \{P\}\, s_1\, \{Q\} \iff \{P\}\, s_2\, \{Q\}$$

4.2.4 Total Correctness (Termination)

We introduce an alternative form of Hoare triple:

$$\{P\}\, s\, \{\Downarrow Q\}$$

The informal meaning of the above triple is:

> If P evaluates to true in the initial state σ then the execution of s from σ terminates and Q will evaluate to true in the final state.

We do not mix these triples with those of partial correctness. However, all total correctness derivation rules are analogous to those for partial correctness, except for the rule for loops.

Loop Variants Termination is proved using `loop variants`, which is an expression that evaluates to a value in a well-founded set before each iteration.

While Rule for Total Correctness The total correctness derivation rule for loops is given by:

$$\frac{\{b \wedge P \wedge e = Z\}\, s\, \{\Downarrow P \wedge e < Z\}}{\{P\}\ \text{while } b \text{ do } s \text{ end}\, \{\Downarrow \neg b \wedge P\}} (\text{WHTOT}_{Ax}) \qquad \text{if } b \wedge P \vDash 0 < e$$

4.3 Soundness and Completeness

We define the following two terms:

- `Soundness`: If a property can be proved then it does indeed hold.
- `Completeness`: If a property does hold then it can be proved.

Soundness and completeness can be proved w.r.t. an operational semantics (here, big-step semantics):

The partial correctness tripel $\{P\}\, s\, \{Q\}$ is valid, written as $\vDash \{P\}\, s\, \{Q\}$, iff:

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

- *Soundness*: $\vdash \{P\}\, s\, \{Q\} \implies \vDash \{P\}\, s\, \{Q\}$
- *Completeness*: $\vDash \{P\}\, s\, \{Q\} \implies \vdash \{P\}\, s\, \{Q\}$

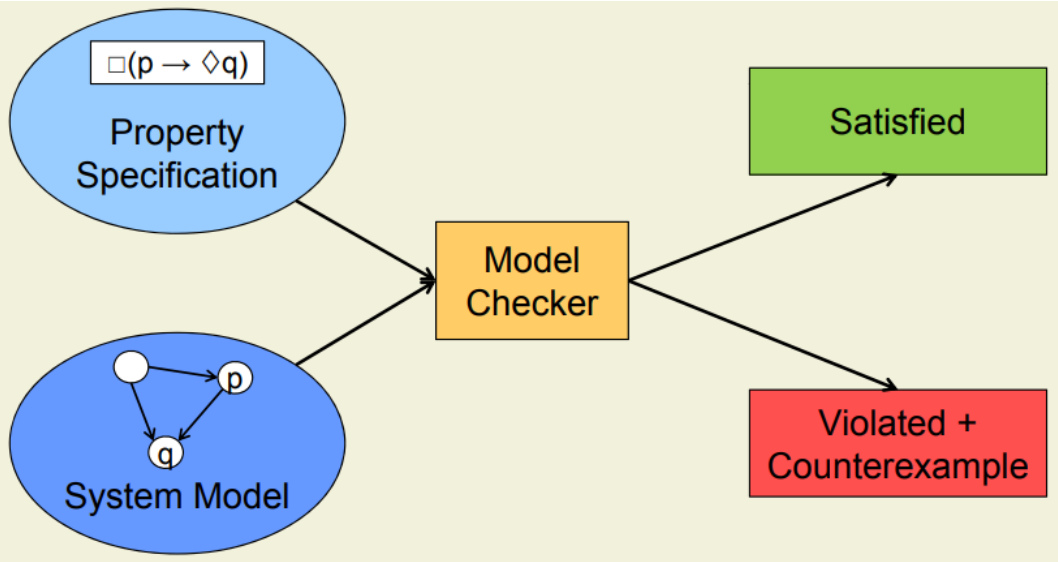
The soundness and completeness theorem of partial correctness is given by:

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

$$\vdash \{P\}\, s\, \{Q\} \iff \vDash \{P\}\, s\, \{Q\}$$

5. 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 Checking Process

The model checking process consists of three different phases:

1. Modeling phase: Model the system under consideration using the description language of your model checker and formalize the properties to be checked.
2. Running phase: Run the model checker to check the validity of the property in the system model.
3. Analysis phase: If property is satisfied, celebrate. If property is violated, analyze counterexample, reduce, and try again.

5.1 Protocol Meta Language Promela

`Promela` is the input language of the `Spin` model checker. Main objects are processes, channels, and variables. Promela has a C-like syntax:

```
init {
    printf("Hello World!\n)
}
```

5.1.1 Promela Programs

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 The simple form of a process is given by:

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

- The body consists of a sequence of variable declarations, channel declarations, and statements
- There are no arrays as parameters

An `active process` can be declared with:

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

This starts N instances of myProc in the initial state. The init process is started in the initial state.

5.1.2 Promela Types

The following primitive types exist in Promela:

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

There are no floats or mathematical (unbounded) integers! Furthermore, the following used-defined types are available:

- `Arrays`: int name [4]
- `Structures`
- `Types of symbolic constants`: mytype
- `Channel type`: chan

5.1.3 Promela Variable and Channel Declarations

Variable declaration works as follows:

```
byte a, b = 5, c;
int d[3], e[4] = 3;
mytype msg = ack;
vector v;
```

Channel declarations looks like:

```
chan c1 = [2] of { mytype, bit, chan };
chan c2 = [0] of { int };
chan c3;
```

- $c1$ can store up to two messages. Messages sent via $c1$ consists of three parts.
- $c2$ models rendez-vous communication.
- $c3$ is uninitialised.

5.1.4 State Space

State Space of Sequential Programs The number of states is given by:

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

where $|dom(x)|$ denotes the number of possible values of variable x.

State Space of Concurrent Programs The number of states of $P \equiv P_1||...||P_N$ is at most

$$\#\text{states of } P_1 \times ... \times \#\text{states of } P_N =$$

$$\prod_{i=1}^{N}(\#\text{program locations}_i \times \prod_{\text{variable } x_i} |dom(x_i)|)$$

State Space of Promela Models The number of states of a system with N processes and K channels is at most

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

where $|dom(c)|$ denotes the number of possible messages of channel c and $cap(c)$ is the capacity of channel c.

5.1.5 State Transitions

A `state transition` is made in three steps:

1. Determine all executable statements of all active processes. If no executable statement exists, the transition system gets stuck.
2. Choose non-deterministically one of the executable statements.
3. Change the state according to the chosen statement.

5.1.6 Promela Expressions and Statements

Promela expressions include:

- Variables, constants, and literals
- Structure and array accesses
- Unary and binary expressions
- Function applications
- Conditional expressions `(E1 -> E2: E3)`

Furthermore, we have the following `promela statements`:

- skip : Does not change the state in is always executable
- timeout : Does not change the state and is executable if all other statements in the system are blocked.
- assert(E) : Aborts execution if expression E evaluates to zero, otherwise it is equivalent to skip. Is always executable.
- Assignment: $x = E$ assigns the value of E to variable x. Is always executable.

5.2 Motivation

5.2.1 Example: Verification of Parallel Programs

Consider the following simplified Java program:

```
class Cell {
    int x = 0;

    static void main(...) {
        Cell c = new Cell():
        Thread t1 = new Even(c);
        Thread t2 = new Even(c);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(c.x)
    }
}
```

```
class Even extends Thread {
    Cell c;

    Even(Cell c) {
        this.c = c;
    }

    void run() {
        c.x = c.x + 1;
        c.x = c.x + 1;
    }
}
```

Modeling Even.run We can model the Even.run method in Promela as follows:

```
int x;

proctype EvenRun() {
    int y = x;
    y = y + 1;
    x = y;
    y = x;
    y = y + 1;
    x = y;
}
```

Note that we need to extend the model of `c.x = c.x + 1` (instead of simply writing `x = x + 1`), since assignments in Promela are atomic, but in Java they are not.

Modeling Cell.main Modelling the main program could look as follows:

```
init {
    x = 0;

    run EvenRun();
    run EvenRun();

    /* Wait for termination */
    _nr_pr == 1;

    printf("x: %d\n", x);
    assert x % 2 == 0;
}
```

`_nr_pr` is a predefined global variable that yields the number of active processes.

5.3 More on Promela

5.3.1 Promela Statements

Selection Selection in Promela looks as follows:

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

This is executable if at least one of its options is executable. It chooses an option *non-deterministically* and executes it.

Repetition Repetitions look the following way:

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

This is executable if at least one of its options is executable. Chooses *repeatedly* an option non-deterministically and executes it. It terminates when a break or goto is executed.

Atomic Basic statements are executed `atomically` in Promela. This means that there is no interleaving during the execution of statements such as skip, timeout, assert, assignments, etc.

Furthermore, `atomic { s }` executes atomically. It is executable if the first statement of s is executable.

5.3.2 Promela Macros

Promela does not contain any procedures. However, a similar effect can often be achieved using `macros`. A macro just defines a replacement text for a symbolic name, possibly with parameters.

Example:

```
inline swap(a, b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

The inline call swap(a, b) is replaced by the body of the definition.

5.3.3 Promela Channels

`chan ch = [d] of { t1,..., tn }` declares a `channel`. A channel can buffer up to d messages. Each message is a tuple whose elements have types `t1,..., tn`.

Example:

```
mytype = { req, ack, err };

chan ch = [5] of { mytype, int };
```

Send and Receive `ch ! e1,..., en` sends a message. Type of e_i must correspond to t_i in the channel declaration.

`ch ? a1,..., an` receives messages. Receive is executable iff buffer is not empty and the oldest message in the buffer matches the constants a_i.

In an `unbuffered channel`, i.e. a channel of type `chan ch = 0 ...`, send is executable if there is a receive operation that can be executed simultaneously and vice-versa.

6. Linear Temporal Logic

6.1 Linear-Time Properties

Transition Systems Revisited We use a slightly different definition here. A finite transition system is a tuple $(\Gamma, \sigma_I, \rightarrow)$, where:

- 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 key differences are that we have a fixed initial configuration and we omit terminal configurations from the definition.

Computations We denote:

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

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

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

$\mathcal{C}(TS)$ denotes the set of all computations of a transition system TS.

Linear-Time Properties Linear-time 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.

A TS satisfies an LT-property P over Γ

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

Example: "All opened files must be closed eventually":

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

From Configurations to Propositions For a transition system TS, we additionally specify a set AP of `atomic propositions`:

- An atomic proposition is a proposition containing no logical connectives
- Example: $AP = \{\text{open, closed}\}$ (for files)

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

- $L : \Gamma \to \mathcal{P}(AP)$
- Example: $L(\sigma) = \{\text{open},\}$ if $\sigma(o) = 1$, $\{\text{closed}\},$ if $\sigma(o) = 0$, $\{\}$ otherwise

We call $L(\sigma)$ an `abstract state`.

Traces A `trace` is an abstraction of a computation. We observe only the propositions of each state. not the concrete state itself.

$t \in \mathcal{P}(AP)^\omega$ is a `trace of a transition system` TS if $t = L(\gamma_{[0]})L(\gamma_{[1]})L(\gamma_{[2]}), ...$ and γ is a computation of TS.

We denote the set of all traces of a transition system TS as $\mathcal{T}(TS)$.

Safety Properties Intuition: "*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 $\hat{t}$, $t' \notin P$.

Liveness Properties Intuition: "*Something good will happen eventually*".

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

6.2 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. Whether or not the traces of a finite transition system satisfy an LTL formula is `decidable`.

LTL: Basic Operators Syntax:

$$\phi = p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \, U \, \phi \mid \bigcirc \phi$$

where p is a proposition from a chosen set of atomic propositions $AP \neq \emptyset$.

LTL: Semantics $t \vDash \phi$ expresses that trace $t \in \mathcal{P}(AP)^\omega$ satisfies LTL formula ϕ | | | |—|–| | $t \vDash p$ | iff $p \in t_{[0]}$ | | $t \vDash \neg\phi$ | iff not $t \vDash \phi$ | | $t \vDash \phi \wedge \psi$ | iff $t \vDash \phi$ and $t \vDash \psi$ | | $t \vDash \phi U \psi$ | iff there is a $k \geq 0$ with $t_{\geq k} \vDash \psi$ and $t_{\geq j} \vDash \phi$ for all j such that $0 \leq j < k$ | | $t \vDash \bigcirc\phi$ | iff $t_{\geq 1} \vDash \phi$ |

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

Derived Operators

- $true$, $false$, $\vee$, $\implies$, $\iff$ are defined as usual
- Eventually: $\diamond\phi \equiv (\text{true } U \, \phi)$
- Always: $\Box\phi \equiv \neg \diamond \neg\phi$

Useful Specification Patterns

- Strong invariant: $\Box\psi$
  - ψ always holds
- Monotone invariant: $\Box(\psi \implies \Box\psi)$
  - Once ψ is true, then ψ is always true
- Establishing an invariant: $\diamond\Box\psi$
  - Eventually ψ will always hold
- Responsiveness: $\Box(\psi \implies \diamond\phi)$
  - Every time that ψ holds, ϕ will eventually hold
- Fairness: $\Box \diamond \psi$
  - ψ holds infinitely often

7. Model Checking

We would like to solve the following LTL model checking problem:

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

Finite Automaton for Finite Prefixes

Reminder: an NFA is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where

- Q : is a finite set of states
- Σ : is a finite alphabet
- δ : a transition relation
- Q_0 : the set of initial states
- F : the set of accepting states

Given a transition system $TS = (\Gamma, \sigma_I, \rightarrow)$, we define an NFA $\mathcal{FA}_{TS}$ characterizing all finite prefixes $\mathcal{T}_{fin}(TS)$ of the traces of TS.

Regular Safety Properties

Given an automaton, we can check simple LTL formulas manually. An LTL formula is valid in a transition system, if every trace of the transition system satisfies the formula.

A safety property is `regular` if its bad prefixes are described by a regular language over the alphabet $\mathcal{P}(AP)$.

We check regular safety properties in the following way:

1. Describe the finite prefixes $\mathcal{T}_{fin}(TS)$ 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 of $\mathcal{FA}_{TS}$ and $\mathcal{FA}_{\bar{P}}$
4. Check if the resulting automaton has any reachable accepting states
    1. If not, the property P is never violated in traces of TS
    2. If yes, the property P is violated

Product Automaton

Construct NFA $\mathcal{FA}_{TS \cap \bar{P}}$ that accepts the intersection of the languages accepted by both automatons.

Emptiness Check

If a product automaton $\mathcal{FA}_{TS \cap \bar{P}}$ accepts a word w then:

- $w \in \mathcal{T}_{fin}(TS)$ because it is accepted by $\mathcal{FA}_{TS}$ and
- w is bad prefix because it is accepted by $\mathcal{FA}_{\bar{P}}$
- Therefore, P is not satisfied, and w is a counterexample

ω-Regular Languages

Regular expressions denote languages of finite words. ω-regular expression denote languages of infinite words.

An ω-regular expression G has the form

$$G = E_1 F_1^{\omega} | ... | E_n F_n^{\omega} \quad (1 \leq n)$$

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

Büchi Automata

Büchi automata are similar to finite automata, but accept infinite words. A run of an NBA accepts its inputs if it passes infinitely often through an accepting state.

LTL Model Checking

1. Describe traces $\mathcal{T}(NS)$ by NBA $\mathcal{BA}_{TS}$
2. For an LTL formula ϕ, construct NBA $\mathcal{BA}_{\neg\phi}$ that accepts the traces characterized by $\neg\phi$ (the 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
    1. If no, property ϕ is violated and each word in the language is a counterexample