Formal Methods and Functional Programming

Spring 2021

Jean-Claude Graf

September 20, 2021

Contents

Part I. Functional Programming

1. Introduction

1.1. Functional Programming

- Like mathematical expressions
- Consists of functions and values
- Functions are actually values themselves
- There is no state
- **Referential Transparency:** Not state $\implies$ expression **always** evaluate to the same value
- No global variables
- Recursion instead of iteration

+ Easy to parallelize
+ Easy to analyze
+ Flexible type system

1.2. Haskell

- **Lazy Evaluation:** expression evaluates always outermost and leftmost expression
  - But pattern matching and some other functions force evaluation

1.2.1. Syntax

- **Function**
  - Function name and arguments start with lower-case
  - Expression after the equal sign is the return value
  - **Pattern Matching**
    - Is used for:
      - Check if argument has proper type
      - Bind values to variables
    - **Pattern**
      - Inductively defined
      - Pattern are
        - Constants
        - Variables
        - Wild Card (`_`)
        - Tuples $(p_1, p_2, ..., p_k)$ where p_i is a pattern
        - Non-Empty Lists $(p_1 : p_2)$ where p_i is a pattern
      - Must be **linear**
        - I.e. each variable cannot occur more than once
        - Does not count for wild card
    - **Pattern Matching**
      - Pattern matching is used to determine right definition
      - Pattern `p` matches term `a` by the following recursion on `p`:
        - **Constant:** `p = c` if `c = a`
        - **Variable:** `p = x` always succeeds with binding `x = a`
        - **Wild Card:** `p = _` always success but without binding
        - **Tuple:** $p = (p_1, \dots, p_k)$ succeeds if $a = (a_1, \dots, a_k)$ and p_i matches $a_i \quad \forall i \in \{1, \dots, k\}$
        - **Non-Empty List:** $p = (p_1 : p_2)$ succeeds if $a = (a_1 : a_2)$ and p_i matches $a_i \quad \forall i \in \{1, 2\}$

        - Forces evaluation (no longer lazy evaluation)
        - Can define the same function multiple times but with different patterns
    - May Contain several cases (*guards*)
        - Boolean expression
        - `otherwise` is the default case
    - **Scope**
        - Functions have a global scope
        - Order of declaration does not matter
        - `let <local func, var, const decl.> in <expr. using these defs>`
            - More powerful than `where`
        - `where <local func, var, const decl.>`
            - Follows guard or function return
            - Top-Down development (use and then declare)
- Constants can be defined outside of functions
- Program consists of multiple function definitions
- **Indentations**
    - Determines separation of definitions
    - All function definitions start at the same indentation
    - The body of a function definition needs to be indented
    - If line is split into two, indent new line again
        - Can be done recursively
    - Spaces have to be used, not tabs

1.2.2. Types

- Strongly typed
- Can explicitly define function definition or let Haskell do that
- **Integral**
    - **Int:** Bound
    - **Integer:** Arbitrary precision
- **Double**
- **Char**
    - Surrounded by `'`
- **String**
    - List of characters
    - Surrounded by `"`
    - Concatenate using `++`
- **Bool**
- **Function/Operator**
    - **Operator:** Binary function which is used infix
    - `` `func` `` makes function infix
    - `(op)` makes operator prefix
- **Tuple**
    - Compose multiple values of different type
    - Composed by a *Type Constructor*
    - If $T_1, \ldots, T_n$ are Types, then $(T_1, \ldots, T_n)$ is a tuple type
    - If $v_1 :: T_1, \ldots, v_n :: T_n$ are values of matching type, then $(v_1, \ldots, v_n) :: (T_1, \ldots, T_n)$ is a valid tuple
    - Can be nested

1.2.3. Input/Output

- I/O is not referential transparent (has side effects)
- Wrap by `IO` to capture side effects
- `getLine :: IO String` reads a string
- `putStrLn :: String -> IO ()` prints a string.
- `do` blocks sequences side effects
- `<-` extract values from IO
- `return` wraps values in IO

2. Natural Deduction

- Allows formal reasoning (proofs) about systems

2.1. Natural Deduction

- **Rules** allow to derive from assumptions $A_1, \dots, A_n \vdash A$
- Derivations model trees
- Can construct derivation bottom-up or top-down
- **Proof** is a derivation without assumptions in the root
- Can be read as:
    - ◇ **Top-Down:** From the upper statement, the lower follows according to some rule
    - ◇ **Bottom-Up:** To proof the lower statement, it is sufficient to show the upper statement

2.2. Propositional Logic

2.2.1. Syntax

- **Language of Propositional Logic $\mathcal{L}_p$:** For set of variables $\mathcal{V}$, $\mathcal{L}_p$ is the minimal set with:
    - ◇ $X \in \mathcal{L}_p$ if $X \in \mathcal{V}$
    - ◇ $\perp \in \mathcal{L}_p$
    - ◇ $A \wedge B \in \mathcal{L}_p$ 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 $B \in \mathcal{L}_p$
    - ◇ $A \rightarrow B \in \mathcal{L}_p$ if $A \in \mathcal{L}_p$ and L_p and $B \in \mathcal{L}_p$
- Convention: X stands for variables, A, B for formulae

2.2.2. Semantics

- **Valuation σ:** Mapping assigning truth values to all variables
    - ◇ $\sigma : \mathcal{V} \rightarrow \{True, False\}$
    - ◇ **Valuations:** set of valuations
- **Satisfiability $\models$:** Smallest relation $\subseteq$ Valuations $\times \mathcal{L}_p$ such that:
    - ◇ $\sigma \models X$ if $\sigma(X) =$ True
    - ◇ $\sigma \models A \wedge B$ if $\sigma \models A$ and $\sigma \models B$
    - ◇ $\sigma \models A \vee B$ if $\sigma \models A$ or $\sigma \models B$
    - ◇ $\sigma \models A \rightarrow B$ if whenever $\sigma \models A$ then $\sigma \models B$
- **Satisfiable:** is formula $A \in \mathcal{L}_p$ if $\exists \sigma, \sigma \models A$
- **Valid/Tautology:** is formula $A \in \mathcal{L}_p$ if $\forall \sigma, \sigma \models A$
- **Semantic Entailment:** $A_1, \dots, A_n \models A$ if $\forall \sigma$ for which $\sigma \models A_i,\ \forall i \in [1, n]$ then $\sigma \models A$

2.2.3. Requirements for Deductive System

- Syntactic ($\vdash$) and semantic ($\models$) entailment should agree:
    - ◇ **Soundness:** If $H \vdash A$ can be derived, then $H \models A$
    - ◇ **Completeness:** If $H \models A$ then $H \vdash A$ can be derived
- Decidability is also a desired property
    - ◇ I.e. is some formula satisfiable, tautology, satisfied by a valuation etc.

2.2.4. Natural Deduction

- **Sequent:** Assertion of the form $A_1, \dots, A_n \vdash A$, where A, A_i are propositional formulae
  - If deduction system is sound, this is a semantic entailment
- **Axiom:** Leaves of a derivation tree
  - Starting point for derivation trees
  - $$\frac{}{\dots A, \dots \vdash A}\ (\text{axiom})$$
- Proof of A if root is $\vdash A$
  - If deduction system is sound, then A is a tautology
- **Rules:**
  - Each rule must be sound
    - I.e. is must preserve semantic entailment
  - If each rule is sound, then the logic is sound
  - **Safe** is rule if we only enlarge Γ or we can get back the conclusion
  - **And:**
    $$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge\text{-I}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\text{-EL}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\text{-ER})$$
  - **Or:**
    $$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee\text{-IL}) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee\text{-IR}) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee\text{-E})$$
  - **Implies:**
    $$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ (\rightarrow\text{-I}) \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\rightarrow\text{-E})$$
  - **Negation:** Define $\neg A$ as $A \rightarrow \perp$
    $$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\neg\text{-E})$$
  - **Falsity:**
    $$\frac{\Gamma \vdash \perp}{\Gamma \vdash A}\ (\perp\text{-E})$$
  - **tertium non datur:**
    $$\frac{}{\Gamma \vdash A \vee \neg A}\ (\text{TND})$$
  - **reductio ad adsurdum:**
    $$\frac{\Gamma, \neg A \vdash \perp}{\Gamma \vdash A}\ (\text{RAA})$$

  TODO: Make safe/unsafe
- **Proof Strategy:** Apply safe rules first

2.3. First-Order Logic

2.3.1. Syntax

- **Signature:** Set of function symbols $\mathcal{F}$ and set of predicate symbols $\mathcal{P}$
  - f^i/p^i indicate the arity of function f/predicate p
- **Term:** For set of variables $\mathcal{V}$, the smallest set where:
  - $x \in$ Term if $x \in \mathcal{V}$
  - $f^n(t_1, \ldots, t_n) \in$ Term if $f^n \in \mathcal{F}$ and $t_j \in$ Term $\forall 1 \leq j \leq n$
- **Formulae:** Smallest set where:
  - $\perp \in$ Form
  - $p^n(t_1, \ldots, t_n) \in$ Form if $p^n \in \mathcal{P}$ and $t_j \in$ Term $\forall 1 \leq j \leq n$
  - $A \circ B \in$ Form if $A \in$ Form, $B \in$ Form, and $\circ \in \{\wedge, \vee, \rightarrow\}$
  - $Qx.A \in$ Form if $A \in$ Form, $x \in \mathcal{V}$, and $Q \in \{\forall, \exists\}$
- Quantifier extend as far as possible (EOL or closing outer bracket)
- Occurrence of a variable is either free or bound
  - Variable x is bound in formula A if it occurs within a subformula B of A of the form $Qx.B,\ Q \in \{\exists, \forall\}$
  - Names of bound variables are irrelevant
  - **α-Conversion:** Rename bound variables
    - Keep binding structure (association between quantifier and variables)
    - Prevent capture (renaming to the name of a free variable)
  - x not free $\not\Longrightarrow$ x bound
    - x could also just not occur
- **Binding**
  1) $\neg$
  2) $\wedge$
  3) $\vee$
  4) $\rightarrow$
- **Associativity**
  - **Right:** $\rightarrow$
  - **Left:** $\wedge, \vee$

2.3.2. Semantics

- **Structure:** Pair $S = \langle U_S, I_S \rangle$
  - U_S is a non-empty universe
  - I_S is a mapping which assigns each predicate $p^n \in P$/formulae $f^n \in \mathcal{F}$ its truth value/definition
- **Interpretation:** Pair $\mathcal{I} = \langle S, v \rangle$
  - $S = \langle U_S, I_S \rangle$ is a structure
  - $v : \mathcal{V} \rightarrow U_S$ is a valuation
- **Value:** of a term t under the interpretation $\mathcal{I}$ is written as $\mathcal{I}(t)$ with
  - $\mathcal{I}(x) = v(x), x \in \mathcal{V}$
  - $\mathcal{I}(f(t_1, \ldots t_n)) = f^S(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$
- **Satisfiability $\models$:** Smallest relation $\subseteq$ Interpretations $\times$ Form such that:
  - $\langle S, v \rangle \models p(t_1, \ldots, t_n)$ if $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in p^S$
  - $\langle S, v \rangle \models \forall x.A$ if $\langle S, v[x \mapsto a] \rangle \models A, \forall a \in U_S$
  - $\langle S, v \rangle \models \exists x.A$ if $\langle S, v[x \mapsto a] \rangle \models A, \exists a \in U_S$
  - etc
  - Where

  * $\mathcal{I} = \langle S, v \rangle$
  * $v[x \mapsto a]$ is valuation v' identical to v except that $v'(x) = a$
- If $\langle S, v \rangle \models A$ and A has no free variables, then $S \models A$
- **Valid:** is A if every suitable interpretation is a model
  - **Notation:** $\models A$
- **Satisfiable:** if $\exists$ a model for A
- **Contradictory:** if $\nexists$ model for A

2.3.3. Substitution

- Replace in A all occurrences of a free variable x with some term t
- **Notation:** $A[x/t]$
- Must avoid capture
  - Free variables of t must still be free in $A[x/t]$
  - May need to α-convert first
  - It is ok if it clashes with another free variable

2.3.4. Natural Deduction

- In addition to the propositional logic rules we have
- **Universal Quantifier:**

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} (\forall\text{-I})^{x \text{ not free in any formula in } \Gamma} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x/t]} (\forall\text{-E})$$

- **Existential Quantifier:**

$$\frac{\Gamma \vdash A[x/t]}{\Gamma \vdash \exists x.A} (\exists\text{-I}) \qquad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B}{\Gamma \vdash B} (\exists\text{-E})^{x \text{ not free in any formula in } \Gamma \text{ or } B}$$

2.4. Equality

- Is a logical symbol and not just a predicate
- **Extend language**
  - **Formula:** $t_1 = t_2 \in \text{Form}$ if $t_1, t_2 \in \text{Term}$
  - **Satisfiability:** $\mathcal{I} \models t_1 \underbrace{=}_{\text{syntactic}} t_2$ if $\mathcal{I}(I_1) \underbrace{=}_{\text{semantic}} \mathcal{I}(t_2)$
- **Rules**
  - **Equivalence Relation:**

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

  - **Congruence Relation:**

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

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

- Equality proofs are easier in linear way than using natural deduction trees

3. Correctness

- Properties of a correct program:
  - **Termination:** Does not count for all, but most programs
  - **Functional Behaviour:** Function should return "correct" value
- Must be proven

3.1. Termination

- If f is composed of functions $g_1, \ldots, g_k$ and $g_i \neq f$ and each g_i terminates then f terminates
- Recursive function terminates if the arguments are smaller along a well-founded order on the function's domain
  - Is a sufficient condition
  - **Well-Founded:** is the order $>$ on set S iff there is no infinite decreasing sequence in S
    - **Relation composition** of two binary relation R_1, R_2 on set S is $R_2 \circ R_1 \equiv \{(a, c) \in S \times S \mid \exists b \in S.aR_1b \wedge bR_2c\}$
      - For $R \subset S \times S$:
        - $R^1 \equiv R$
        - $R^{n+1} \equiv R \circ R^n, n \geq 1$
        - $R^+ \equiv \bigcup_{n \geq 1} R^n$
    - For $R \subseteq S \times S$, $s_0, s_i \in S$ and $i \geq 1$. Then $s_0 R^i s_i$ iff $\exists s_1, \ldots s_{i-1} \in S$ such that $s_0 R s_1 R \ldots R s_{i-1} R s_i$
    - If $>$ is well-founded order on S then so is $>^+$.

3.2. Behaviour

- **Equality Reasoning**
  - **Goal:** Show function return is equal to some value
  - **Idea:** Function are equations
  - Apply equational reasoning
  - Proof using FOL with equality
- **Reasoning by Cases**
  - For predicate functions
  - Often use
    - **Excluded Middle (TND):** For all prepositions $P, P \vee \neg P$
    - **Case Split ($\vee$-E):** Prove $P = Q \vee R$ by proving $Q \implies P$ and $R \implies P$
- **Induction**
  - Dual of recursion
  - Prove $P(n) \forall n \in Nat.$
    - **Base Case:** Proof $P[n/0]$
    - **Step Case:** Proof $P[n/m+1]$ by assuming $P[n/m]$ for some arbitrary but fixed m
      - m must not be free in P
      - Can also take $P[n/n]$ to remove side condition
  - Natural Deduction
    - $$\frac{\Gamma \vdash P[n/0] \quad \Gamma \vdash \forall m \in \text{Nat}.P[n/m] \rightarrow P[n/m+1]}{\Gamma \vdash \forall n \in \text{NAT}.P} \text{(NAT-IND)}^{m \text{ not free in } P}$$

- **Well-Founded Induction/Notherian Induction (not exam relevant)**
  - ◇ **Well-Founded Step:** Prove $P[n/m]$ by assuming $P[n/l] \forall l < m$
    - ∗ m and l not free in P
  - ◇ Stronger than normal induction

4. Lists

4.1. Introduction

- **List Type:** If `T` is a type then `[T]` is a type
  - ◇ Is a new type constructor
- **Empty List:** `[] :: [T]`
- **Non-Empty List:** `(x : xs) :: [T]` iff `x :: T` and `xs :: [T]`
  - ◇ **Cons Operator:** `:` prepends an element to a list
    - ∗ **Concatenate:** `++` concatenates two lists
  - ◇ `[a, b, c]` is syntactic sugar for `a : (b : (c : []))`
- `[n, p .. m]` constructs a list from `n` to `m` with step `p - n`
  - ◇ `p` is optional; Default step is `1`
  - ◇ Can be seen as, "First element is `n`, second element is `p`, continue like this till `m`"
  - ◇ `m` is not necessarily included
- String is a list of chars
  - ◇ `['a', 'b'] == "ab"`

4.2. Sorting Algorithms

- **Insertion Sort**

```
isort :: [Int] -> [Int]
isort [] = []
isort (x: xs) = ins x (isort xs)

ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (x: xs)
    | a <= x = a : (x: xs)
    | otherwise = x : ins a xs
```

- **Quick Sort (long form)**

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x: xs) = qsort ( lesseq x xs) ++ [x] ++ qsort (greater x xs)
where
    lesseq _ [] = []
    lesseq x (y: ys)
        | y <= x = y : lesseq x ys
        | otherwise = lesseq x ys
    greater _ [] = []
    greater x (y: ys)
    | y >x = y : greater x ys
    | otherwise = greater x ys
```

- **Quick Sort (short form)**

```
q :: [Int] -> [Int]
q [] = []
q (p:xs) = q [x | x <- xs, x <= p] ++ [p] ++ q [x | x <- xs, x > p]
```

4.3. List Comprehension

- Notation for sequential processing of list elements
- Analogous to set comprehension in set theory
- General form: `[func x | <gen_1>, ... , <gen_n>, <pred_1, ..., <pred_m>]`
- Generators can depend on each other
  - ◇ E.g. `[x | n <- [1..10], x <- [1..n]]`
- Generators can depend on if then else
  - ◇ E.g. `[x | n <- [1..10], if even x then x <- [1,2] else x <- [1]]`
- TODO: Add more handy dandy examples

4.4. Induction Over Lists

- Prove P for all xs in $[T]$
  - ◇ **Base Case:** prove $P[xs/[]]$
  - ◇ **Step Case:** prove $\forall y :: T, ys :: [T].P[xs/ys] \to P[xs/y : ys]$
    - ∗ **Fix** arbitrary but non-free $y :: T, ys :: [T]$
    - ∗ **Induction Hypothesis:** Assume $P[xs/ys]$
- Sometimes hard to pick right induction variable
  - ◇ Proof may fail depending on the variable
- **Generalisation**
  - ◇ Proof a stronger statement as a subproof
  - ◇ Required for some proofs

5. Abstraction

- **Polymorphic Type t:** A set of types
- **Parametric Polymorphism:** Function works for type `t` iff it works for all types contained in `t`
- A type `w` for function `f` is a **most general** (/principal) **type** iff for all types `s` for `f`, `s` is an instance of `w`.
- Given a function, Haskell always computes the most general type
  - ◇ If we give a type, it must be an instance of the most general type
- Type variables start with lower-case

5.1. Higher-Order Functions

- Types
  - ◇ **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 of arbitrary order
- Advantages

\+ Definition is easier to understand
\+ Parts are easier to modify
\+ Pars are easier to reuse
\+ Correctness is simpler to understand and show

5.1.1. Examples

- **Map**
  - ◇ Apply function to each argument in a list
  - ◇
    ```
    map :: (a -> b) -> [a] -> [b]
    map f [] = []
    map f (x:xs) = f x : map f xs
    ```
- **Folding**
  - ◇ Aggregate all elements of a list
  - ◇ **foldr**
    - ∗ Written as `(f x_1 (f x_2 (f ... (f x_k e)))` for list x, function f and default value e
    - ∗ When seen as a tree, the con is replaced by f and the empty list by e
    - ∗ Can operate on infinite list
      ```
      foldr :: (a -> b -> b) -> b -> [a] -> b
      foldr f e [] = e
      foldr f e (x: xs) = f x (foldr f e xs)
      ```
    - ∗ **Recipe:** Implement some (suitable) function with folder
      - ○ Identify the following arguments:
        - ▷ **Recursive Arguments:** The list which shrinks in each iteration
        - ▷ **Static Arguments:** The ones which do not change
        - ▷ **Dynamic Arguments:** The ones which change arbitrarily
      - ○ Write a helper function `aux` with all recursive and then dynamic arguments

- Move the dynamic arguments to the right of the equals
  - I.e. form a lambda function
  - I.e. η-expansion
- Rewrite the helper function using `foldr` and replace `aux xs` with local variable `rec`
- Inline the helper function

- **foldl**
  - Written as `f(f(f(f e x_1) x_2) ...) x_k` for list x, function f and default value e
  - Runs infinitely on infinite lists
  - ```
    foldl :: (b -> a -> b) -> b -> [a] -> b
    foldl f e [] = e
    foldl f e (x: xs) = foldl f ( f e x) xs
    ```
- foldr and foldl are equivalent for associative functions

5.2. λ-Expression

- Allows in-line function definitions
- Constructed as `(\v_1 -> ... -> \v_k -> <someExpression>)`
  - Syntactic sugar `(\v_1 ... v_k -> <someExpression>)`
- Adoption of Church's λ-notation
- **η-Conversion:** `x -> f x` and `f` are equivalent
  - **η-Contraction:** From left to right
  - **η-Expansion:** From Right to left
  - Useful to simplify expression

5.3. Function as Values

- Function itself can be returned from function
- Returned function cannot be displayed, but only evaluated

5.3.1. Examples

- **Function Composition**
  - Takes two functions as arguments and returned the composite function
  - Application associates to the left
  - ```
    (.) :: (b -> c) -> (a -> b) -> (a -> c)
    (f . g) x = f (g x)
    ```
- **Iteration**
  - Apply a function `a -> a` n times to a input x
  - ```
    iter :: Int -> (a -> a) -> a -> a
    iter 0 f x = x
    iter n f x = f (iter (n - 1) f x)
    ```
- **Difference Lists**
  - **Problem:** Appending to list is expensive
  - **Idea:** Construct list as a higher order (first) function
  - ```
    type DList a = [a] -> [a]

    empty :: DLists a
    empty = \xs -> xs
    ```

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

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 []
```

5.4. Function Arguments

- **Partial Application**
  - One applies only some but not all arguments
  - A new function, still requiring some arguments, is returned
  - Useful for `map`, `filter` etc.
  - If $f :: t_1 \to t_2 \to \cdots \to f_n \to t$ and $e_1 :: t_1, \ldots, e_k :: t_k$ then the partial application has type $f e_1 \ldots e_k :: t_{k+1} \to \cdots \to t_n \to t$
  - Partial application is consistent with the view that function takes multiple arguments
    - But a function takes exactly one arguments
  - For infix operator $\oplus$:
    - $(a\oplus) \equiv \lambda x.a \oplus x$
    - $(\oplus a) \equiv \lambda x.x \oplus a$
    - Importend to consider when operator is not commutative
      - `(a `func`) != (`func` a)`
- **Tupling**
  - Wrapping multiple arguments into tuple lets us apply them as one argument
  - Function is one of:
    - **Curry Func:** Takes multiple arguments
    - **Uncury Func:** Takes a tuple as argument
  - We want convert one representation to the other using:
    - **Curry:** Uncurry $\to$ curry

      ```
      curry :: ((a,b) -> c) -> a -> b -> c
      curry f = f' where f' x1 x2 = f (x1,x2)
      ```
    - **Uncurry:** Curry $\to$ uncurry

      ```
      uncurry :: (a -> b -> c) -> (a,b) -> c
      uncurry f' = f where f (x1,x2) = f' x1 x2
      ```
- **Uncluttering Notation**
  - Right associative operator `$` for arguments
  - Avoids parentheses

6. Types

- Should prevent dangerous expressions
  - Which cause a runtime error
- Classification (good/bad) of expressions is undecidable
  - Type systems are conservative and only allow what they are sure is good
- Type checker should offer:
  - quick, decidable, static analysis
  - permit generality/re-usability
  - prevent runtime-errors

6.1. Mini-Haskell

- Typing system
- Subset of Haskell
- **Syntax**
  - Programs are terms
  - $$t ::= \underbrace{\mathcal{V}}_{\text{Variables}} \mid \underbrace{(\lambda x.t)}_{\text{lambda abstraction}} \mid \underbrace{(t_1\ t_2)}_{\text{functions}} \mid \text{True} \mid \text{False} \mid$$
    $$(\text{iszero } t) \mid \underbrace{\mathcal{Z}}_{\text{Integers}} \mid (t_1 + t_2) \mid (t_1 * t_2) \mid$$
    $$\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \mid \underbrace{(t_1, t_2)}_{\text{Pairing}} \mid (\text{fst } t) \mid (\text{snd } t)$$
  - Can easily be extended
  - Add syntactic sugar: Can leave out parenthesis when not necessary
- **Typing**
  - Set of types $\tau ::= \underbrace{\mathcal{V}_\tau}_{\text{Set of Type Variables } (a,b,\dots)} \mid \text{Bool} \mid \text{Int} \mid \underbrace{(\tau, \tau)}_{\text{Pair Constructor}} \mid \underbrace{(\tau \to \tau)}_{\text{Function Constructor}}$
  - **Typing Judgement:** $\Gamma \vdash t :: \tau$
    - Γ**:** Set of bindings mappings from variables to types
    - t**:** Term
    - τ**:** Type
    - "Given assignments Γ, term t is of type τ"
  - **Rules**
    - **Basic:**

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

    - **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 $\in \{+, *\}$

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

* **Conditional:**

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

* **Tuples:**

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

- **Type Inference**
  - Given term t what is its type?
  - Algorithms:
    1. Start with judgement $\vdash t :: \tau_0$ where τ_0 is the type variable and t is the expression whose type we want to determine
    2. Build derivation tree bottom-up by applying rules and collect constraints. Introduce fresh type variables if need
    3. Solve constraints to get possible types
  - Some terms are untypeable
    * Type inference fails to build inference tree or constraints are unsolvable
- **Type Proof**
  - Given term t and type τ. Prove that $t :: \tau$

  TODO: Add Type Proof section
- **Self Application**
  - Apply function f to itself: $\lambda f.ff$
  - Is not typeable
- **Curry-Howard Isomorphism (not examrelevant)**
  - Type constructor '$\rightarrow$' corresponds to propositional logic connectivity '$\rightarrow$'
  - Atomic types correspond to propositional variables
  - Rules correspond to those minimal propositional logic

6.2. Type Classes

- Defines
  - Set of types
  - Set of allowed functions on these types
- Allow restricted for of type generalisation
- **Monomorphic:** Restricted to a single type (base type)
- **Polymorphic:** Restricted by the type set (a type class)

6.2.1. Type Class

- Definition
  - **Name:** upper-case
  - **Signature:** Function names with their type
    * Required to be implemented by instances of this type
  - **Default Definition:** Definition based on other signatures
    * Optional
    * Can be overwritten
  -
```
class Eq a where -- Class Name
    (==) :: a -> a -> Bool -- Signature
```

```
(/=) :: a -> a -> Bool -- Signature

x /= y = not (x == y) -- Default definition
```

  ◇ To indicate that a certain type `t` if of type class `Eq` we write `Eq t => t`

- **Instance**
  ◇ Application of a type class to a certain type
  ◇ Elements of a class are instances
  ◇ Interprets signature functions
    * Requires defining all signatures and optionally, overwrite default definitions
  ◇ Done using keyword `instance`
  ◇ `instance Eq Bool where`
```
True == True   = True
False == False = True
_ == _         = False
```
  ◇ Can be recursive
    * If `t` is of type `Eq` then so is `[Eq]`
    * I.e. membership depends on membership of other type
    * `instance Eq t => Eq [t]  where`
```
[]     == []     = True
(x:xs) == (y:ys) = x == y && xs == ys
_      == _      = False
```

6.2.2. Derived Classes and Class Hierarchies

- Type classes can build on top of other type classes
- If `a` belongs to the child type, is must also belong to the parent type
- All function of the parent type are inherited and some new ones (may be) added
- `class Eq a => Ord a where...`
- Arbitrarily nested classes can be created

6.3. Overloading

- Execution of parametric polymorphic functions independent of type of arguments
- Classes implements *ad hoc* polymorphism
- Selection of function definition is either
  ◇ At compile time if types are knows
  ◇ else, at runtime

TODO: I am not sure what this all means

7. Algebraic Data Types

- Declare new data types suitable for the object being modeled
- Algebraic means it is the smallest set
+ Less error prone

7.1. Data Types

- **Enumeration Types**
  - ◇ Set of possible types
    - ∗ Each element is a *type constructor*
  - ◇ Initiated by keyword `data`
  - ◇ Constructors must have unique names
  - ◇ First letter of each constructor must be upper-case
  - ◇ `data TypeName = Const1 | Const2 | Const3`
  - ◇ Function can use this type for pattern matching
    - ∗ `func :: TypeName -> SomeOtherType`
  - ◇ Type class can have type variables as arguments
    - ∗ For polymorphism
    - ∗ `data TypeName a = ...`
- **Product Type**
  - ◇ Consists of a type name and a set of "attributes"
    - ∗ Attribute must be a certain type
      - ○ Giving an alias using `type` adds a layer of abstraction
  - ◇ `data TypeName = Name Attr1 Attr2`
    - ∗ `type Attr1 = Sometype`
    - ∗ `TypeName` and `Name` are often the same
  - ◇ Constructor is a function `Name :: Attr1 -> Attr2 -> TypeName`
  - ◇ Functions can use this type for pattern matching
    - ∗
      ```
      func :: TypeName -> SomeOtherType
      func (Name Attr1 Attr2) = ...
      ```
  - ◇ Could use tuples instead of product types
    - ∗ `data TypeName = (Attr1, Attr2)`
    - \- Makes arguments ambiguous
    - \+ Allows application of polymorphic functions like `fst, zip...`
    - \+ Shorter definition
- **Enumeration and Product Types**
  - ◇ Enumeration and product types can be combined
  - ◇ `data TypeName = Name1 Attr1 | Name2 Attr1 Attr2`
  - ◇ Functions can use this type for pattern matching
    - ∗
      ```
      func :: TypeName -> SomeOtherType
      func (Name1 Attr1)       = ...
      func (Name2 Attr1 Attr2) = ...
      ```

7.2. Integration with Classes

- Default function are not applicable to our custom data types
- Have to be explicitly created

  - ```
    data TypeName = Name1 Attr1 | Name2 Attr1 Attr2
    instance TypeClass TypeName where
     ...
    ```
- In some cases class instances can be automatically derived
  - ```
    data TypeName = Name1 Attr1 | Name2 Attr1 Attr2
                    deriving(TypeClass1, TypeClass2, TypeClass3)
    ```

7.3. Recursive Types

- Defined using recursive data types
  - `data Expr = Lit Int | Add Expr Expr`
- Are evaluated recursively
  - ```
    eval :: Expr -> Int
    eval (Lit n) = n
    eval (Add a b) = (eval a) + (eval b)
    ```
- **Example:** Trees
  - Are a prime example
  - Can describe many data structures
  - ```
    data Tree t = Leaf | Node t (Tree t) (Tree t)
                  deriving (Eq, Ord, Show)
    ```

7.4. Algebraic Types and Type Classes

- Algebraic types are *fist class* citizens
  - Fully compatible with polymorphism and type classes
- Standard types are algebraic data types defined in the prelude

+ Make program simpler to read and understand
+ Allow reusability

7.5. Correctness

- **Natural Number**
  - `data Nat = Zero | Succ Nat deriving (Eq, Ord, Show)`
  - Isomorphic to `{Zero, Succ Zero, Succ (Succ Zero), ...}`
  - Build step by step
  - Allows structural induction proofs
- **Lists**
  - `data L t = Nil | Cons t (L t)`
  - Elements in $L\ t$ are build in steps
    * $\{\text{Nil}\}$
    * $\{\text{Cons } a \text{ Nil} \in L\ t \mid a \in t\}$
    * $\{\text{Cons } b(\text{Cons } a \text{ Nil}) \in L\ t \mid a, b \in t\}$
    * $\vdots$
  - $l \in L\ t$ iff l appears in some of the construction step
  - Rule

$$\frac{\Gamma \vdash P[xs/\text{Nil}] \quad \Gamma, P[xs/ys] \vdash P[xs/\text{Cons } y\ ys]}{\Gamma \vdash \forall xs \in L\ t.P}\ (\text{IND on List})^{y,ys \text{ not free in } \Gamma, P}$$

- **Trees**

- `data Tree t = Leaf | Node t (Tree t) (Tree t)`
- Elements in $Tree\ t$ are build in steps
  - {Leaf}
  - {Node a Leaf Leaf $\in Tree\ t \mid a \in t$}
  - $\vdots$
  - Trees in step i are of form Node $a\ l\ r$ where $a \in t$, and l and r were constructed in the previous step
- $s \in Tree\ t$ iff s appears in some of the construction step
- Rule

$$\frac{\Gamma \vdash P[x/\text{Leaf}] \quad \Gamma, P[x/l], P[x/r] \vdash P[x/\text{Node } a\ l\ r]}{\Gamma \vdash \forall x \in \text{Tree } t.P} \text{(IND on Tree)}^{a,l,r \text{ not free in } \Gamma, P}$$

- **General Idea**
  - Adopt induction to the structure of the algebraic data type
  - Proof non-recursively step 0
  - Proof recursively how to get from step $i-1$ to i

8. Lazy Evaluation

- Only evaluate arguments when needed
- Substitute arguments without argument evaluation
- Some expressions are never evaluated
  - ◇ Can save arbitrary amount of work
- **Duplicate Evaluation:**
  - ◇ One argument may be used multiple times
  - ◇ Haskell avoids duplicate evaluation of the same arguments
  - ◇ **Sharing:** Pointer graph of arguments indicated if an argument was already executed
    - ∗ If it was, we can directly take the result
- Arguments are evaluated only when needed and at most once
- **Pattern Matching**
  - ◇ Arguments evaluate as far as needed to determine pattern match
  - ◇ Start matching the top most pattern and on failure go to the next
- **Guards**
  - ◇ Evaluate only what is required to check if guard is true
  - ◇ Start matching the top most guard and on failure go to the next
- **Local Definitions**
  - ◇ `where` and `let` are lazily evaluated
- **Functions**
  - ◇ Outermost operator is first evaluated
    - ∗ Top-down evaluation in a syntax tree
  - ◇ If on same level, evaluate from left to right or according to operator precedence
- **Recipe:** Evaluate `t1 t2` lazily
  - ◇ Evaluate `t1`
  - ◇ The argument `t2` is substituted in `t1` without being evaluated
  - ◇ No evaluation inside lambda abstractions
    - ∗ I.e. in an abstraction `(\t -> f t)`, (where `f` is some arbitrary term), then `f t` is not evaluated
- **Recipe:** Evaluate `t1 t2` eagerly
  - ◇ Evaluate `t1`
  - ◇ `t2` is evaluated prior to substitution in `t1`
  - ◇ Evaluation is carried out inside lambda abstractions

8.1. Application

- **Data-Driven Programming**
  - ◇ Data can be generate on demand
    - ∗ Improved runtime complexity
  - ◇ Due to lazy evaluation, only required data is constructed
- **Infinite Data**
  - ◇ Finite representation of infinite data
    - ∗ E.g. `from n = n : ones (n+1)` generates an infinite list
  - ◇ We can calculate with infinite data in finite time
    - ∗ E.g. `head from 1`
  - ◇ I.e. we describe an infinite stream and compute with arbitrarily large finite prefixes of it

8.2. Correctness

- Complicated analysis of correctness and complexity
- Type like `[Int]` include finite and infinite lists
- Proof by induction is sound only for finite lists
  - We always assume finite lists for this course

9. Case Study

9.1. Overview Interpreter

- Has three basic steps:
- **Read**
    - ◇ **Input:** Text
    - ◇ **Phases:**
        - ∗ **Lexical Analysis**
            - ○ Convert source code to tokens
                - ▷ I.e. tell for each groups of symbols what they are
            - ○ **Tokens:** Identifier (variables), arithmetic symbols, assignment symbol, numbers, etc.
            - ○ White-spaces and comments are removed
        - ∗ **Parsing**
            - ○ Build abstract syntax tree
            - ○ Syntax is specified by a given grammar
                - ▷ I.e. a data type in Haskell
        - ∗ **Outer Phases**
            - ○ Depending on the applications, further phases may come now
            - ○ Things like type conversion, type checking, dependency analysis, etc.
    - ◇ **Output:** Abstract Syntax Tree
    - ◇ Lexical analysis and parsing is required for all systems
- **Evaluate**
    - ◇ **Input:** Abstract Syntax Tree
    - ◇ **Semantic Interpretation**
    - ◇ **Output:** Abstract Syntax Tree
- **Print**
    - ◇ **Input:** Abstract Syntax Tree
    - ◇ **Pretty Print Output**
    - ◇ **Output:** Text

9.2. Overview Parser

- Parser is a function
- **Input:** String
- **Output:** Element of type `a`
    - ◇ Typically `a` is some data type
- A parser may not necessarily parse the whole input
    - ◇ **Combinatory Parsing**
    - ◇ There is a remainder
    - ◇ `res, rem`
    - ◇ Remainder may be parsed by a different parser
- A parser may try to produce different results for the same input
    - ◇ Store `(res_i, rem_i)` in a list
    - ◇ If `rem_i = ""` the parse is complete
    - ◇ `data Parser a = Prs (String -> [(a, String)]`
- **Application**
    - ◇
```
parse :: Parser a -> String -> [(a, String)]
parse (Prs p) inp = p inp
```

- **Result of (first) Complete Parse**
  - ◇
```
completeParse :: Parser a -> String -> a
completeParse p inp
| result == [] = error "Parse unsuccessful"
| otherwise    = head results
where results  = [res | (res, "") <- parse p inp]
```
- **Primitive Parsers**
  - ◇ Server as a basic building block
  - ◇ **Failure:**
    - ∗ Fails trivially
    - ∗ `[]` signifies a unsuccessful parse
    - ∗
```
failure :: Parser a
failure = Prs (\inp -> [])
```
    - ∗ Ex.
```
$ parse failure "3+5"
[] :: [(a, String)]
```
  - ◇ **Return:**
    - ∗ Succeeds trivially
    - ∗ Without progress
    - ∗
```
return :: a -> Parser a
return x = Prs (\inp -> [(x, inp)]
```
    - ∗ Ex.
```
$ parse (return "foo") "3+5"
[("foo", "3+5")] :: [([Char], String)]
```
  - ◇ **Item:**
    - ∗ Succeeds trivially
    - ∗ With progress
    - ∗
```
item :: Parser Char
item = Prs (\inpt -> case inp of
                     "" -> []
                     (x:xs) -> [(x, xs)])
```
    - ∗ Ex.
```
$ parse item "3+5"
[('3', "+5")] :: [(Char, String)]
```
  - ◇ **Sat:**
    - ∗ Parse single char with property p
    - ∗
```
sat :: (Char -> Bool) -> Parser Char
sat p = Prs (\inp -> case inp of
                     "" -> []
                     (x:xs) -> if p x then [(x,xs)] else [])
```
    - ∗ Alternatively
```
sat :: (Char -> Bool) -> Parser Char
sat p = item >>= \x -> if p x then return x else failure
```
    - ∗ Ex. isDigit
```
$ parse (sat (\x -> '0' <= x && x <= '9')) "3+5"
[('3', "+5")] :: [(Char, String)]
```
    - ∗ Ex. isArithOp
```
$ parse (sat (\x -> x == '+' || x == '-')) "3+5"
```

```
[] :: [(Char, String)]
```

- **Char:**
  - `char :: Char -> Parser Char`
    `char x = sat (==x)`
- **String:**
  - ```
    string :: String -> Parser Strin
    string "" = return ""
    string (x:xs) = char x >> string xs >> return (x:xs)
    ```
- **Many:**
  - 0 or more repetitions of p
  - ```
    many :: Parser a -> Parser [a]
    many p = many1 p ||| return []
    ```
- **Many1:**
  - 1 or more repetitions of p
  - ```
    many1 :: Parser a -> Parser [a]
    many1 p = p >>= \t -> many p >>= \ts -> return (t:ts)
    ```
- **numPos:**
  - ```
    numPos :: Parser Int
    numPos = do ts <- many1 (sat isDigit)
        return (read ts)
    ```
- **numNeg**
  - ```
    numNeg :: Parser Int
    numNeg = do char '-'
        t <- numPos
        return (-t)
    ```
- **num**
  - ```
    num :: Parser Int
    num = numPos ||| numNeg
    ```
  - ```
    $ parse num "123"
    [(123, ""), (12, "3"), (1, "23")]
    $ parse num "-123"
    [(-123, ""), (-12, "3"), (-1, "23")]
    ```

- **Combining Parsers**
  - **Mutual Selection:** Apply both parser and concatenate result
    - ```
      (|||) :: Parser a -> Parser a -> Parser a
      p ||| q = Prs (\s -> Parser p s ++ parser q s)
      ```
    - Ex
      ```
      $ parse (return '!' ||| sat isDigit) "3+5"
      [('!', "3+5"), ('3', "+5")]
      ```
  - **Alternative Selection:** Apply second parser only of first fail
    - ```
      (+++) :: Parser a -> Parser a -> Parser a
      p +++ q = Prs (\s -> case parser p s of
                      [] -> parser q s
                      res -> res)
      ```
    - Ex
      ```
      $ parse (return '!' +++ sat isDigit) "3+5"
      [('!', "3+5")]
      ```
  - **Sequencing:** Apply second parser on remainder of first parser. Return result and remainder of second parser
      ```

  - Result of first parser is lost
  - `(>>) :: Parser a -> Parser b -> Parser b`
```
p >> q = Prs (\s -> [(u, s'') | (t, s') <- parse p s,
                                (u, s'') <- parse q s'])
```
  - Ex
```
$ parse (sat isDigit >> sat (== '+')) "3+5"
[('+', "5")]
```
- **Sequencing 2:** Apply second parser on result of first parser. Return combined result and remainder of second parser
  - Second parser is a **parser generator**
  - `(>>=) :: Parser a -> (a -> Parser b) -> Parser b`
```
p >>= g = Prs (\s -> [(u, s'') | (t, s') <- parser p s
                                (u, s'') <- parser (g t) s'])
```
  - Can improve readability by using syntactic sugar
    - `do t1 <- p1`
```
t2 <- p2
...
tn <- pn
return (f t1 t2 ... tn)
```
for
```
p1 >>= \t1 ->
p2 >>= \t2 ->
...
pn >>= \tn ->
return (f t1 t2 ... tn)
```
    - `Parser` must be an instance of `Monad`
  - Ex
```
$ parse (sat isDigit >>=
        \t -> sat isDigit >>=
        \u -> return (t:u:[])) "31+5"
[("31", "+5")]
$ parse (sat isDigit >>=
        \t -> sat isDigit >>=
        \u -> return (t:u:[])) "3+5"
[]
```

9.3. Arithmetic Interpretation

- **Read**
  - **Grammar:** `Expr ::= Int | Expr '+' Expr | Expr '-' Expr`
    - Resp. `data Expr n Lit Int | Add Expr Expr | Sub Expr Expr`
  - **Lexical Analysis:** Recognize integers, `'+'`, `'-'`, parentheses and white space
  - **Parsing:** Convert to abstract syntax tree
- **Evaluation**
  - `eval :: Expr -> Int`
```
eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```
- **Print**

  - ◇ Instance of type class show
  - ◇ `instance Show Expr where`
    ```
    show (Lit n) = show n
    show (Add e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
    show (Sub e1 e2) = "(" ++ show e1 ++ "-" ++ show e2 ++ ")"
    ```
- **Parser**
  - ◇ Given grammar is ambiguous
    - ∗ Provide user a way to get rid of ambiguity
    - ∗ `Expr ::= Int | Expr '+' Expr | Expr '-' Expr | '(' Expr ')'`
  - ◇ Given grammar is left-recursive
    - ∗ Parsing `Expr` requires to first parse `Expr`
    - ∗ We can get an infinitely non-terminating recursion
    - ∗ `Atom ::= Int | '(' Expr ')'`
      `Expr ::= Atom | Atom '+' Expr | Atom '-' Expr/`
  - ◇ Parser
    - ∗
      ```
      data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
          deriving (Show, Eq)

      atom lit ||| pexpr
      expr = atom ||| add ||| sub

      lit = do x <- num
               return (Lit x)

      pexpr = do string "("
                 e <- expr
                 string ")"
                 return e

      add = do a <- atom
               string "+"
               e <- expr
               return (Add a e)

      sub = do a <- atom
               string "-"
               e <- expr
               return (Sub a e)
      ```
  - ◇ Evaluator
    - ∗
      ```
      str2expr :: String -> Expr
      str2expr s = completeParse expr s

      eval :: Expr -> Int
      eval (Lit n) = n
      eval (Add x y) = eval x + eval y
      eval (Sub x y) = eval x - eval y

      calculate :: String -> Int
      calculate = eval . str2expr
      ```

TODO: Example 2

Part II. Formal Methods

10. Introduction

- **Transitional SE**
  - ◇ Documentation is incomplete
  - ◇ Testing is good, but
    - - They are insufficient
    - - Detect concurrency issues is e.g. very difficult
    - - Impossible to cover all instances
- **Formal Methods:** Mathematical approaches to software and system development which support the rigorous specification, design, and verification of computer systems
  - ◇ Programs, programming languages, designs etc. are mathematical objects and can be treated by mathematical methods
  - ◇ Used for
    - ∗ Proving program properties
    - ∗ Formalizing language semantics
    - ∗ Proving language properties
  - ◇ **Steps:**
    - ∗ **Specification:**
      - ◦ **System Design:** What does the system look like?
      - ◦ **Requirements:** What should the system do?
      - ◦ **Assumptions:** What do we assume?
        - ▷ E.g. an attacker cannot break a encryption
      - ◦ Described in mathematical notation
    - ∗ **Verification:**
      - ◦ **Validate Specifications:** Do the specifications make sense?
      - ◦ **Proof:** Requirements are fulfilled under the specifications and requirements
        - ▷ Often simple but tedious
      - ◦ Done using format logic
        - ▷ **Deduction:** Proof system
        - ▷ **Algorithmic:** State space exploration or model checking
  - ◇ **State Space Exploration:** Enumerate all possible states
    - ∗ Done very efficiently
    - ∗ Check for deadlocks
    - - Problem space may be very large
      - ◦ Limit to important properties
    - - Gives weaker correctness guarantees than proofs
  - ◇ Pro/cons
    - + Strong guarantees
      - + Proof for all possible constellations
      - + Unambiguous documentation
    - - Writing (correct) specifications is hard
    - - Many properties are undecidable
      - ◦ Tools are limited
    - - Give often false positive or false negative
    - - FM specialist required
    - - FM application is expensive
    - ∗ FM complements testing
      - ◦ We need tests for
        - ▷ Validate specifications

        - ▷ Test properties not proven
        - ▷ Detect errors in environment
    - ○ FM aids tests
        - ▷ Derive test cases and test data from specifications
        - ▷ Increase test coverage
        - ▷ Replaces tests
- **Used For**
    - ◇ Verification of design
    - ◇ Analysis of safety-crucial software
    - ◇ Detection of security vulnerabilities
    - ◇ Enforce usage of API and/or protocols
    - ◇ Analysis of security protocols
    - ◇ Verification of system implementations
    - ◇ Design of programming languages
    - ◇ Implementation of programming languages
    - ◇ Reasoning about programs

10.1. IMP

- Has boolean and arithmetic expressions
- Expressions have no side effects
- All variables range over integers
- All variables are initialized
- Does not include
  - ◇ Heap allocation and coiners
  - ◇ Variable declaration
  - ◇ Procedures
  - ◇ Concurrency
- Is very extensible
- **Syntax**
  - ◇ **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 Or Aexp ')'
           | Var
           | Numeral

      Op = '+' | '-' | '*'
      ```
  - ◇ **Boolean Expressions:**
    - ∗
      ```
      Bexp = '(' Bexp 'or' Bexp ')'
           | '(' Bexp 'and' Bexp ')'
           | 'not' Bexp
           | Aexp Rop Aexp

      Rop = '=' | '#' | '<' | '<=' | '>' | '>='
      ```
  - ◇ **Statement:**
    - ∗
      ```
      Stm = 'skip'
          | Var ':=' Aexp
          | '(' Stam ';' Stm ')'
          | 'if' Bexp 'then' Stm 'else 'stm 'end'
          | 'while' Bexp 'do' Stm 'end'
      ```
    - ∗ Parentheses are omitted of possible
  - ◇ **Abbreviations:**
    - ∗ "if b then s end" for "if b then s else skip end"
    - ∗ "true" for "$1 = 1$"
    - ∗ "false" for $0 = 1$"
- **Variables**
  - ◇ **Program Variables:**
    - ∗ Are concrete variables in a program
    - ∗ Written in typewriter font
  - ◇ **Meta Variables:**
    - ∗ Stand for arbitrary program variables

- Convention:
  - **n:** for numerals (`Numeral`)
  - **x,y,z:** for variables (`Var`)
  - $\mathbf{e}, \mathbf{e'}, \mathbf{e_1}, \mathbf{e_2}$**:** for arithmetic expressions (`Aexp`)
  - $\mathbf{b}, \mathbf{b'}, \mathbf{b_1}, \mathbf{b_2}$**:** for boolean expressions (`Bexp`)
  - $\mathbf{s}, \mathbf{s'}, \mathbf{s_1}, \mathbf{s_2}$**:** for statements (`Stm`)
  - σ**:** for states
- Meta variables stand for arbitrary program variables
- Written in math font

◇ **Syntactic Equality** $\equiv$**:**
- $x \equiv y$ (meta variables) may be true
  - I.e. both denote the same program variable
- $\mathtt{x} \equiv \mathtt{y}$ (program variables) is always false
  - But two program variables by have the same value $\mathtt{x} = \mathtt{y}$

- **Semantics**

◇ **States**
- An expression depends on the value bound to the variables that occur in it
- State : Var $\to$ Val
  - Assigns each variable a value
  - Total function
- **Sigma State** $\sigma_{\mathbf{zero}}$**:** All variables have the value 0
- **Updating States** $\sigma[\mathbf{y} \mapsto \mathbf{v}]$**:** Assign v to y in the state σ
  - $$(\sigma[y \mapsto v])(x) = \begin{cases} v & \text{if } x \equiv y \\ \sigma(x) & \text{otherwise} \end{cases}$$
- **Equality** of states σ_1, σ_2 if they are equal as functions $\sigma_1 = \sigma_2 \iff \forall x.(\sigma_1(x) = \sigma_2(x))$

◇ **Semantic Functions** map elements of syntactic categories to elements of semantic categories
-
- **Syntactic Category:** E.g. `Numeral`
  - Some ascii symbol
- **Semantic Category:** E.g. $\mathbb{Z}$
  - Actual value

◇ **Numerals:** Syntactic Category `Numeral`
- $\mathcal{N}$: Numeral $\to$ Val
- Maps numeral n to integer value $\mathcal{N}[[n]]$
  - Convention to use double brackets
  - Same as with single bracket
- $$\begin{aligned} \mathcal{N}[[0]] &= 0 & \mathcal{N}[[n0]] &= \mathcal{N}[[n]] \times 10 + 0 \\ \mathcal{N}[[1]] &= 1 & \mathcal{N}[[n1]] &= \mathcal{N}[[n]] \times 10 + 1 \\ &\dots & &\dots \\ \mathcal{N}[[9]] &= 9 & \mathcal{N}[[n9]] &= \mathcal{N}[[n]] \times 10 + 9 \end{aligned}$$

◇ **Arithmetic Expressions:** Syntactic Category `Aexp`
- $\mathcal{A}$: Aexp $\to$ State $\to$ Val

  - Maps arithmetic expression e and a state σ to a value $\mathcal{A}[[e]]\sigma$
  -

$$\begin{aligned}\mathcal{A}[[x]]\sigma &= \sigma(x)\\ \mathcal{A}[[n]]\sigma &= \mathcal{N}[[n]]\\ \mathcal{A}[[e_1 \text{ op } e_2]]\sigma &= \mathcal{A}[[e_1]]\sigma \ \overline{\text{op}}\ \mathcal{A}[[e_2]]\sigma \quad \text{, where op} \in \text{Op}\end{aligned}$$

    - $\overline{\text{op}}$ is the operation Val $\times$ Val $\rightarrow$ Val corresponding to op
    - E.g. op = '+' and $\overline{\text{op}}$ = mathematical addition
- **Arithmetic Operators:** Syntactic Category `Op`
- **Boolean Expressions:** Syntactic Category `Bexp`
  - $\mathcal{B}$: Bexp $\rightarrow$ State $\rightarrow$ Bool = $\{tt, ff\}$
  - Maps boolean expression b and state σ to a truth value $\mathcal{B}[[b]]\sigma$
  -

$$\mathcal{B}[[e_1 \text{ op } e_2]]\sigma = \begin{cases} \text{tt} & \text{if } \mathcal{A}[[e_1]]\sigma \ \overline{\text{op}}\ \mathcal{A}[[e_2]]\sigma \\ \text{ff} & \text{otherwise} \end{cases} \quad \text{, where op} \in \text{Rop}$$

    - $\overline{\text{op}}$ is the operation Val $\times$ Val corresponding to op
  -

$$\begin{aligned}\mathcal{B}[[e_1 \text{ or } e_2]]\sigma &= \begin{cases} \text{tt} & \text{if } \mathcal{B}[[e_1]]\sigma = \text{ tt or} \mathcal{B}[[e_2]]\sigma = \text{ tt} \\ \text{ff} & \text{otherwise} \end{cases}\\ \mathcal{B}[[e_1 \text{ and } e_2]]\sigma &= \begin{cases} \text{tt} & \text{if } \mathcal{B}[[e_1]]\sigma = \text{ tt and} \mathcal{B}[[e_2]]\sigma = \text{ tt} \\ \text{ff} & \text{otherwise} \end{cases}\\ \mathcal{B}[[\text{not } e]]\sigma &= \begin{cases} \text{tt} & \text{if } \mathcal{B}[[e]]\sigma = \text{ff} \\ \text{ff} & \text{otherwise} \end{cases}\end{aligned}$$

- **Relational Operators:** Syntactic Category `Rop`
- **Statements:** Syntactic Category `Stm`

10.2. Properties

- $\mathcal{A}, \mathcal{B}$ are defined recursively
- Base elements are defined directly
- Composite elements are defined inductively in terms of immediate constitutes
- Definition suggests proof by structural induction
- **Structural Induction over Programs**
  -

10.3. Free Variables

- Free Variables
  - All variables occurring in an expression
  - The naming may be confusing since we do not mean "free" in terms of "bound or free" but rather if there were replaced by a concrete value
  - **Arithmetic Expressions:**

*

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

◇ **Boolean Expressions:**

*

$$
\begin{aligned}
&FV(e_1 \text{ op } e_2 = FV(e_1) \cup FV(e_2)\\
&FV(\text{not } b) = FV(b)\\
&FV(b_1 \text{ or } b_2) = FV(b_1) \cup FV(b_2)\\
&FV(b_1 \text{ and } b_2) = FV(b_1) \cup FV(b_2)
\end{aligned}
$$

◇ **Statements:**

*

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

- Substitution
  - ◇ $\_[x \mapsto e]$
    - * Replace free variable x by e in some expression
  - ◇ **Arithmetic Expressions:**

*

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

◇ **Boolean Expressions:**

*

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

TODO: Move lemma to right location

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

11. Operational Semantics

- Describes execution on a abstract machine
- Describes how to effect is achieved
- Describes how the state is modified during the execution of a statement
- Useful for proofs about language design and implementations

11.1. Big-Step Semantics

- Describes how the overall result of the execution are obtained
- **Natural Semantics (NS):** The system we use
  - **Configuration:**
    - Two types
    - **Normal Configuration $\langle \mathbf{s}, \sigma\rangle$:** Statement s is to be executed in state σ
    - **Terminal Configuration σ:** Final state
  - **Transition System:** Tuple $(\Gamma, T, \rightarrow)$
    - **Γ:** Set of configurations
      - $\Gamma = \{\langle s, \sigma\rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \cup \text{State}$
    - **T:** Set of terminal configurations
      - $T = \text{State} \subseteq \Gamma$
    - **→:** Transition relation
      - $\rightarrow \subseteq \{\langle s, \sigma\rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \times \text{State} \subseteq \Gamma \times \Gamma$
      - Described how execution takes place
      - $\langle s, \sigma\rangle \rightarrow \sigma'$
  - **Inference Rules:**
    - **Rule Schemas:** Contain meta-variables
    - **Rule Instance:** Replacing all meta-variables with syntactic elements
      - Only rule instances can be applied
    - Meta-variables are written using underline
    - **Rules**
      - **Skip**
        - Does not modify the state
        - $\dfrac{}{\langle \texttt{skip}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}}$ $(\text{SKIP}_{\text{NS}})$
      - **Assignment**
        - Assigns some value to a variable
        - $\dfrac{}{\langle \underline{x} := \underline{e}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}[\underline{x} \mapsto \mathcal{A}[[\underline{e}]]\underline{\sigma}]}$ (ASS_{NS})
      - **Sequential composition**
        - Execute the first statement in the initial state, then the second statement in the intermediate state, resulting to some new final state
        - $\dfrac{\langle \underline{s}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}' \quad \langle \underline{s}', \underline{\sigma}'\rangle \rightarrow \underline{\sigma}''}{\langle \underline{s}; \underline{s}', \underline{\sigma}\rangle \rightarrow \underline{\sigma}''}$ (SEQ_{NS})
      - **If**
        - If the conditional is true, execute the first statement, else the second
        - if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \text{tt}$: $\dfrac{\langle \underline{s}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}'}{\langle \text{if } \underline{b} \text{ then } \underline{s} \text{ else } \underline{s}' \text{ end}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}'}$ (IFT_{NS})
        - if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \text{ff}$: $\dfrac{\langle \underline{s}', \underline{\sigma}\rangle \rightarrow \underline{\sigma}'}{\langle \text{if } \underline{b} \text{ then } \underline{s} \text{ else } \underline{s}' \text{ end}, \underline{\sigma}\rangle \rightarrow \underline{\sigma}'}$ (IFF_{NS})
      - **While**

        - ▷ If the condition hold, execute the body once, leading in a new state
        - ▷ if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \text{tt}$: $\dfrac{\langle \underline{s}, \underline{\sigma}\rangle \to \underline{\sigma}' \quad \langle \texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end}, \underline{\sigma}'\rangle \to \underline{\sigma}''}{\langle \texttt{while } \underline{b} \texttt{ to } \underline{s} \texttt{ end}, \underline{\sigma}\rangle \to \underline{\sigma}''}$ (WHT_{NS})
        - ▷ If the condition does not hold, the state is not modified
        - ▷ if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \text{ff}$: $\dfrac{}{\langle \texttt{while } \underline{b} \texttt{ to } \underline{s} \texttt{ end}, \underline{\sigma}\rangle \to \underline{\sigma}}$ (WHF_{NS})
    - ∗ **Derivation Tree T:**
      - ◦ Combination of rule instances
      - ◦ **Root** of T is root(T)
      - ◦ **Leaves** are axiom rule instances
      - ◦ **Internal nodes** are conclusion rule instances, having the premises are immediate children
      - ◦ **Side condition** of all instances must be satisfied
      - ◦ $\vdash \langle s, \sigma\rangle \to \sigma' \iff \exists T.\text{root}(T) \equiv \langle s, \sigma\rangle \to \sigma'$
        - ▷ I.e. if there exists a valid tree with $\langle s, \sigma\rangle \to \sigma'$ in its root
  - ◇ **Termination:**
    - ∗ Execution of statement s in σ:
      - ◦ **Termination Successful:** Iff there exists a state σ' such that $\vdash \langle s, \sigma\rangle \to \sigma'$
      - ◦ **Termination Fails:** Iff there is not state σ' such that $\vdash \langle s, \sigma\rangle \to \sigma'$
- • **Properties**
  - ◇ **Semantic Equivalence**
    - ∗ **Semantically equivalent** are two statements s_1, s_2 iff $\forall \sigma, \sigma'.(\vdash \langle s_1, \sigma\rangle \to \sigma' \iff \vdash \langle s_2, \sigma\rangle \to \sigma')$
      - ◦ **Notation:** $s_1 \simeq s_2$
    - ∗ Loop unrolling is semantically equivalent in IMP
      - ◦ $\forall b, s.(\texttt{while } b \texttt{ do } s \texttt{ end } \simeq \texttt{if } b \texttt{ then } s; \texttt{while } b \texttt{ do } s \texttt{ end end}$
      - ◦ Does not hold for imperative languages
      - ◦ **Proof Idea:**
        - ▷ Show statement in both directions
        - ▷ For each direction, use structural induction
  - ◇ **Deterministic Semantics**
    - ∗ **Lemma:** Big-step semantics of IMP is deterministic
      - ◦ $\forall s, \sigma, \sigma', \sigma''.(\vdash \langle s, \sigma\rangle \to \sigma' \wedge \vdash \langle s, \sigma\rangle \to \sigma'' \implies \sigma' = \sigma'')$
    - ∗ **Proof Idea:**
      - ◦ Structural induction fails if the state does not change
        - ▷ I.e. be have no proper sub-statements
      - ◦ Use induction on the shape of derivation tree
        - ▷ To prove property $P(T)$ for all derivation trees T, prove that $P(T)$ holds for an arbitrary derivation tree T under the assumption that $P(T')$ holds for all sub-trees T' of T
        - ▷ $T' \sqsubset T$
        - ▷ Often we do a case distinction on the rule applied at the root of the tree T
- • **IMP Extension** TODO: IMP Extension
  - ◇
- • **Limitation**
  - ◇ Properties of non-terminating programs cannot be expressed
  - ◇ Non distinction between aborting and non-termination
  - ◇ Non-determinism suppresses non-termination
  - ◇ Parallelism cannot be modeled
  - ◇ Definition of semantic equivalence is coarse

11.2. Small-Step Semantics

- Describes how the individual steps of the computation take place
- Allows to express the order of individual steps
- **Structural Operational Semantics (SOS):** The system we use
  - **Configuration:**
    - Same as for NS
    - Use γ as meta-variables
  - **Transition System:**
    - **Γ:** Set of configurations
      - $\Gamma = \{\langle s, \sigma\rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \cup \text{State}$
      - Same as for NS
      - **Stuck** is non-terminal configuration $\langle s, \sigma\rangle$ if $\nexists\gamma$ such that $\langle s, \sigma\rangle \rightarrow_1 \gamma$
        - Terminal configurations are never stuck
    - **T:** Set of terminal configurations
      - $T = \text{State} \subseteq \Gamma$
      - Same as for NS
    - **$\rightarrow_1$:** Transition relation
      - $\rightarrow_1 \subseteq \{\langle s, \sigma\rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \times \Gamma$
      - $\langle s, \sigma\rangle \rightarrow_1 \gamma$ describes the **first step** of executing s in σ
      - γ can have to forms
      - $\gamma = \langle \mathbf{s}', \sigma'\rangle$: Execution is **not complete** and we get the configuration $\langle s', \sigma'\rangle$
      - $\gamma = \sigma'$: Execution has **terminate** and the final state is σ'
      - **k-step Execution:** $\gamma \rightarrow_1^k \gamma'$
        - I.e. there $\exists$ execution from γ to γ' in exactly k steps
        - Defined inductively over k
        - $\gamma \rightarrow_1^* \gamma'$ means $\exists k.\gamma \rightarrow_1^5 \gamma'$
          - I.e. there is some finite execution
  - **Inference Rules:**
    - **Rules**
      - **Skip**
        - Same as for NS
        - $\dfrac{}{\langle \texttt{skip}, \underline{\sigma}\rangle \rightarrow_1 \underline{\sigma}}$ (SKIP$_{\text{SOS}}$)
        - Same as for NS
      - **Assignment**
        - Same as for NS
        - $\dfrac{}{\langle \underline{x} := \underline{e}, \underline{\sigma}\rangle \rightarrow_1 \underline{\sigma}[\underline{x} \mapsto \mathcal{A}[[\underline{e}]]\underline{\sigma}]}$ (ASS$_{\text{SOS}}$)
      - **Sequential composition**
        - First step of executing the composition is executing the first step of the first statement
        - If the first statement is done after one step
        - $\dfrac{\langle \underline{s}, \underline{\sigma}\rangle \rightarrow_1 \underline{\sigma}'}{\langle \underline{s}; \underline{s}', \underline{\sigma}\rangle \rightarrow_1 \langle \underline{s}', \underline{\sigma}'\rangle}$ (SEQ1$_{\text{SOS}}$)
        - If the first statement is not done after one step
        - $\dfrac{\langle \underline{s}, \underline{\sigma}\rangle \rightarrow_1 \langle \underline{s}'', \underline{\sigma}'\rangle}{\langle \underline{s}; \underline{s}', \underline{\sigma}\rangle \rightarrow_1 \langle \underline{s}''; \underline{s}', \underline{\sigma}'\rangle}$ (SEQ2$_{\text{SOS}}$)
      - **If**
        - The first step of executing an if statement is determine the boolean value of the condition

- ▷ if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \texttt{tt}$: $\dfrac{}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s} \texttt{ else } \underline{s}' \texttt{ end}, \underline{\sigma}\rangle \rightarrow_1 \langle \underline{s}, \underline{\sigma}\rangle}$ (IFT$_{\text{SOS}}$)
- ▷ if $\mathcal{B}[[\underline{b}]]\underline{\sigma} = \texttt{ff}$: $\dfrac{}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s} \texttt{ else } \underline{s}' \texttt{ end}, \underline{\sigma}\rangle \rightarrow_1 \langle \underline{s}', \underline{\sigma}\rangle}$ (IFT$_{\text{SOS}}$)
- ◦ **While**
  - ▷ The first step is to unroll the loop
  - ▷ $\dfrac{}{\langle \texttt{while } \underline{b} \texttt{ to } \underline{s} \texttt{ end}, \underline{\sigma}\rangle \rightarrow_1 \langle \texttt{if } \underline{b} \texttt{ then } \underline{s}; \texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end} \quad \texttt{else} \quad \texttt{skip} \quad \texttt{end}, \sigma\rangle}$ (WH
- ∗ **Derivation Sequence**
  - ◦ Sequence of transitions which cannot be extended with further transitions
  - ◦ Non-empty
  - ◦ Finite or infinite
  - ◦ Sequence of configuration $\gamma_0, \gamma_1, \ldots$ for which
    - ▷ $\gamma_i \rightarrow_1^1 \gamma_{i+1}$ for each $0 \le i$ such that $i+1$ is in the range of sequence
    - ▷ If the derivation sequence is finite, then the last configuration is either a terminal or a stuck configuration
  - ◦ **Length:** Number of transitions
- ∗ **Derivation Tree T:**
  - ◦ Justify a single step in a derivation sequence
  - ◦ Combination of rule instances
  - ◦ $\vdash \langle s, \sigma\rangle \rightarrow_1 \sigma' \iff \exists T.\text{root}(T) \equiv \langle s, \sigma\rangle \rightarrow_1 \sigma'$

◇ **Termination:**
- ∗ Execution of statement s in σ:
  - ◦ **Terminates:** Iff there exists a finite derivation sequence starting with $\langle s, \sigma\rangle$
  - ◦ **Runs Forever:** Iff there exists a infinite derivation sequence starting with $\langle s, \sigma\rangle$

- **Properties**
  - ◇ Proofs over a multi-step execution $\gamma \rightarrow_1^k \gamma'$ are done using strong induction on the number of steps k
    - ∗ Proof the 0-step execution
    - ∗ Proof all other steps using strong mathematical induction
      - ◦ Define $P(k)$
      - ◦ Prove $P(k)$ for arbitrary k with IH. $\forall k' < k.P(k')$
  - ◇ Semantic Equivalence
    - ∗ **Semantically Equivalent** are two statements s_1, s_2 iff $\forall\sigma$ both:
      - ◦ for all stuck or terminal configurations γ we have $\langle s_1, \sigma\rangle \rightarrow_1^* \gamma \iff \langle s_2, \sigma\rangle \rightarrow_1^* \gamma$
        - ▷ The length may be different
        - ▷ The intermediate configurations may be different
      - ◦ there is an infinite derivation sequence starting in $\langle s_1, \sigma\rangle$ iff there is one starting in $\langle s_2, \sigma\rangle$
      - ◦ **Notation:** $s_1 \simeq s_2$
  - ◇ **Determinism**
    - ∗ **Lemma:** Small-step semantics of IMP is deterministic
      - ◦ $\forall s, \sigma, \gamma, \gamma'. \vdash \langle s, \sigma\rangle \rightarrow_1 \gamma \;\wedge\; \vdash \langle s, \sigma\rangle \rightarrow_1 \gamma' \implies \gamma = \gamma'$
    - ∗ **Corollary:** There is exactly one derivation sequence starting in a configuration $\langle s, \sigma\rangle$
    - ∗ **Proof Idea:**
      - ◦ Induction on the spae of the derivation tree for the transition $\langle s, \sigma\rangle \rightarrow_1 \gamma$
- **IMP Extension** TODO: Imp Extension
  - ◇

11.3. Equivalence

- **Theorem:** For every statement s in IMP, $\vdash \langle s, \sigma \rangle \to \sigma' \iff \langle s, \sigma \rangle \to_1^* \sigma'$
  - ◇ If a statement terminates successfully in one semantic, then it also does so in the other, and the finial state is equivalent
  - ◇ The termination fails to terminate in the big-step semantics iff if gets stuck of runs forever in the small-step semantic
- **Proof Idea:**
  - ◇ **⇒:** Induction in the shape of the derivation tree for $\langle s, \sigma \rangle \to \sigma'$
  - ◇ **⇐:** Induction on the number of steps k

12. Axiomatic Semantics

- Expresses specific properties of the effect of executing a program
- Some aspects of the computation may be ignored
- Useful for program verification
- **Partial Correctness:** Expresses that if a program terminates then there will be a certain relationship between the initial and the final state
- **Total Correctness:** Expresses that a program will terminate and there will be a certain relationship between the initial and the final state
  - ◇ Total Correctness = Partial Correctness + Termination
- Proofs are too detailed when using operational semantics
- **Hoare Triples:** The system we use
  - ◇ $\{P\}s\{Q\}$
    - ∗ **P:** Precondition (Assertion)
    - ∗ **Q:** Postcondition (Assertion)
    - ∗ **s:** Statement
  - ◇ If P evaluates to true in an initial state σ, and if the execution of s from σ terminates in an state σ' then Q will evaluate to true in σ'
    - ∗ Describes parietal correctness
  - ◇ **Local Variables**
    - ∗ Can be used to save a value in the inital state so that it can be referenced later
    - ∗ Occur only in assertions
    - ∗ Are never assigned to and are not used by the program
  - ◇ **Assertions**
    - ∗ Consists of boolean expression with local variables (optional)
      - ○ Can be extended with other expressions like quantifiers, new operators etc.
    - ∗ Pre- and postcondition are assertions
    - ∗ We use some convenience notations like $\wedge$ for `and` etc.
  - ◇ **Derivation System**
    - ∗ **Rules**
      - ○ **Skip**
        - ▷ $\dfrac{}{\{\underline{P}\}\texttt{skip}\{\underline{P}\}}$ (SKIP$_{\text{Ax}}$)
      - ○ **Assignment**
        - ▷ $\dfrac{}{\{\underline{P}[\underline{x} \mapsto \underline{e}]\}\underline{x} := \underline{e}\{\underline{P}\}}$ (ASS$_{\text{Ax}}$)
      - ○ **Sequential Composition**
        - ▷ $\dfrac{\{\underline{P}\}\underline{s}\{\underline{Q}\} \quad \{\underline{Q}\}\underline{s}'\{\underline{R}\}}{\{\underline{P}\}\underline{s};\underline{s}'\{\underline{R}\}}$ (SEQ$_{\text{AX}}$)
      - ○ **Conditional Statement**
        - ▷ $\dfrac{\{\underline{b} \wedge \underline{P}\}\underline{s}\{\underline{Q}\} \quad \{\neg\underline{b} \wedge \underline{P}\}\underline{s}'\{\underline{Q}\}}{\{\underline{P}\}\texttt{if } \underline{b} \texttt{ then } \underline{s} \texttt{ else } \underline{s}' \texttt{ end}\{\underline{Q}\}}$ (IF$_{\text{AX}}$)
      - ○ **Loop**
        - ▷ $\dfrac{\{\underline{b} \wedge \underline{P}\}\underline{s}\{\underline{P}\}}{\{\underline{P}\}\texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end}\{\neg\underline{b} \wedge \underline{P}\}}$ (WH$_{\text{Ax}}$)
        - ▷ The assertion P is the loop invariant
      - ○ **Consequence**
        - ▷ $\dfrac{\{\underline{P}'\}\underline{s}\{\underline{Q}'\}}{\{\underline{P}\}\underline{s}\{\underline{Q}\}}$ (CONS$_{\text{Ax}}$)$^{\texttt{if } \underline{P} \models \underline{P}' \texttt{ and } \underline{Q}' \models \underline{Q}}$
        - ▷ **Semantic Entailment** $\models$**:** $P \models Q \iff \forall\sigma, \mathcal{B}[[P]]\sigma = \text{tt} \implies \mathcal{B}[[Q]]\sigma =$

tt
- Strengen precondition
- Weaken postcondition

- **Derivation Tree**
  - As we are used to
  - $\vdash \{P\}s\{Q\} \iff \exists T.\text{root}(T) \equiv \{P\}s\{Q\}$
- **Proof**
  - Two main methods
    - **Proof Trees:**
      - Write as derivation trees
      - Tend to get very long
      - Start from the bottom (/end)
    - **Proof Outlines:**
      - Write proof vertically
      - Not a proof since there is no unique interpretation
        - But most of the time it is ok since we want to show that there exists a derivation tree
  - Loop-invariant is determined by looking how the value changes in consecutive iterations
    - Could use a table with iteration $0, 1, 2, i, N-1$ on the x-axis and the variables we care about on the y-axis
    - Loop invariant is often very similar to the post condition we have

- **Properties**
  - Properties are typically proven by induction on the shape of derivation tree
    - Structural induction does often not work due to the rule of consequence
  - **Semantic Equivalence**
    - **Semantically equivalent** are two statements s_1, s_2 if $\forall P, Q, \vdash \{P\}s_1\{Q\} \iff \vdash \{P\}s_2\{Q\}$
- **Total Correctness (Termination)**
  - **Total Correctness:** 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 statement
  - **Notation:** $\{P\}s\{\Downarrow Q\}$
  - **Loop Variant:**
    - Expression that evaluates to a value in a well-founded set before each iteration
      - Normally we use $\mathbb{N}$
    - Each loop iteration must decrease the value of the invariant
    - Loop has to terminate once the minimal value of the well-founded set is reached
    - Used to prove termination
  - This is a separate axiomatic semantic and is not mixed with the previous one
  - **Rules**
    - **Loop**
      - $\dfrac{\{\underline{b} \land \underline{P} \land \underline{e} = Z\}\underline{s}\{\Downarrow \underline{P} \land \underline{e} < Z\}}{\{\underline{P}\}\texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end}\{\Downarrow \neg\underline{b} \land \underline{P}\}} (\text{WHTOT}_{\text{Ax}})^{\texttt{if } \underline{b}\land\underline{P}\models 0\leq\underline{e} \text{ and } Z\notin\underline{P}}$
    - All other rules are equivalent to before except that we add $\Downarrow$ to the postcondition
  - In proof schemas asserts are often pre- and postcondition. Therefore, we do not write an arrow there. For asserts which are only postcondition, we write an arrow

12.1. Soundness and Completeness

- **Soundness:** If a property can be prove then it does indeed hold

  - $\vdash \{P\}s\{Q\} \implies \models \{P\}s\{Q\}$
- **Completeness:** If a property does hold then it can be proved
  - $\models \{P\}s\{Q\} \implies \vdash \{P\}s\{Q\}$
- Hard to create an axiomatic semantic which is sound and complete
- Soundness and completeness can be proved with respect to an operational semantics
  - $\{P\}s\{Q\}$ is valid, written as $\models \{P\}s\{Q\}$ iff:
    $\forall\sigma, \sigma'.\mathcal{B}[[P]]\sigma = \text{tt} \land \vdash \langle s, \sigma\rangle \to \sigma' \implies \mathcal{B}[[Q]]\sigma' = \text{tt}$
  - I.e. $\models \{P\}s\{Q\}$ is ture if, whenever we start execution of s from a state where P holds, if the execution terminates, then Q will hold in the final state
- **Theorem:** For all partial correctness triplets $\{P\}s\{Q\}$ of IMP we have $\vdash \{P\}s\{Q\} \iff \models \{P\}s\{Q\}$
  - **Proof Idea:**
    - **⇒:** Induction on the shape of the derivation tree for $\{P\}s\{Q\}$
    - **⇐:** Induction but using some weakest precondition stuff

13. Model Checking

- With operational/axiomatic semantics:
  - ◇ Hard to specify properties of sequences of states
  - ◇ Hard to proof interleaving of concurrent systems
  - ◇ Hard to prove programs with infinite derivation sequences
- **Modelling:** Automated technique that, given a finite-state model of a system and a formal property systematically check whether this property holds for (a given state in) that model
- Abstraction of the real world
- Enumerates all possible states of a system
- Mainly used to analyse system designs
  - ◇ And not implementations
- **Explicit State Model Checking:** Represent states explicitly through concrete values
  - ◇ Our focus
- **Symbolic Model Checking:** Represent state through (boolean) formulas
- **Model Checking Process**
  - ◇ **Modelling Phase**
    - ∗ Model the system under consideration using the description language of the model checker
    - ∗ 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 the property is violated, analyse the counter example
    - ∗ If we run out of memory we have to reduce the model
- **Modeling Concurrent Systems**
  - ◇ Systems are modelled as finite transition systems
  - ◇ Systems are modelled as communication sequential processes
  - ◇ Processes can communicate via
    - ∗ Shared variables
    - ∗ Synchronous message passing
    - ∗ Asynchronous message passing

13.1. Promela:

Model checking language we use

- Input language of the Spin model checker
- Main objects are processes, channels and variables
- C-like
- **Syntax:**
  - ◇ Constant declaration
    - ∗ `#define N 5`
      `mytype = {ack, req};`
  - ◇ Variable declaration
    - ∗ `byte a, b = 5, c;`
      `int d[3], e[4] = 3;`
    - ∗ Initialized to zero-equivalent values
    - ∗ Are either local to a process or global
  - ◇ Structure declaration

    * `typedef verctor {int x; int y};`
  ◇ Channel declaration
    * `chan c1 = [2] of {mytype, bit, chan};`
      `chan c2 = [0] of {int};`
      `chan c3;`
    * `c1` can store up to two messages and messages sent via `c1` consists of three parts
    * `c2` models rendez-vous communication as it has no buffer
    * `c3` is uninitialized and must be assigned an initialized channel before usage
    * Are either local to a process or global
  ◇ Process declaration
    * `proctype myProc(int p) {...}`
    * Body contains of a sequence of variable declarations, channel declarations and statements
  ◇ Activate process
    * `active [N] proctype myProc(...) {...}`
    * Start N instances of `myProc` in the initial state
    * The `init` process is started in the initial state
  ◇ **Types**
    * Primitive types

Type	Value range
bit or bool	$0 \dots 1$
byte	$0 \dots 255$
short	$-2^{15} \dots 2^{15} - 1$
int	$-2^{31} \dots 2^{31} - 1$

    * User-defined types
      ○ Arrays: `int name[4]`
      ○ Structures
      ○ Type of symbolic contents: `mtype`
    * Channel type: `chan`
- **State Space:**
  ◇ **Sequential Programs**
    * #states = #program locations $\times \Pi_{\text{variable } x} |\underbrace{\text{dom}(x)}_{\#\text{possible values of } x}|$
    * Exponential growth of states in number of variables
    * State space explosion
  ◇ **Concurrent Programs**
    * Upper bound for # states of $P \equiv P_1 \parallel \cdots \parallel P_N$
    * #states of $P_1 \times \cdots \times$ #states of $P_N = \Pi_{i=n}^{N}(\#\text{program locations}_i \times \Pi_{\text{variable } x_i} |\text{dom}(x_i)|)$
    * Exponential growth of states in number of processes
    * State space explosion
  ◇ **Promela Model**
    * Number of states of a system with N processes and K channels is bounded by

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

$$\Pi_{j=1}^{K} |\underbrace{\text{dom}(c_j)}_{\#\text{possible messages of channel } c}|^{\overbrace{\text{cap}(c_j)}^{\text{capacity/buffer size of channel } c}}$$

    * Exponential growth of states in number of channels and the capacity of channels
    * State space explosion

- **State Transitions:**
  - Statement can be **executable** or **blocked**
    - Send is blocked if channel is full
    - `s1;s2` is blocked if `s1` is blocked
    - `timeout` is executable if all other statements are blocked
  - Transitions is made in three steps
    - Determine all executable statements of all active processes
      - If there are none, transition system gets stuck
    - Choose non-deterministically one of the executable statements
    - Change the state according to the chosen statement
- **Expressions**
  - Variables, constants and literals
  - Structure and array accesses
  - Unary and binary expression with operators
    - `+ - * / % > >= < <= == != ! & || && | ~ >> << ^ ++ --`
  - Function applications
    - `len() empty() nempty() nfull() full() run eval() enable() pcvalue()`
  - Conditional expressions `(E1 -> E2 : E3)`
- **Statements**
  - **skip**
    - Does not change the state
    - Always executable
  - **timeout**
    - Does not change the state
    - Executable if all other statements in the system are blocked
  - **assert(E)**
    - Aborts execution if expression `E` evaluates to zero and 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 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 a value different form zero
    - `E` must not change state
  - **Selection**
    - ```
      if
      :: s1
      :: ...
      :: s=
      fi
      ```
    - Executable if at least one of its options is executable
    - Chooses an option non-deterministically and executes it
    - Optional statement `else` is executed if non of the other options is
  - **Repetition**
      ```

    - `do`
      `:: s1`
      `:: ...`
      `:: sn`
      `od`
    - Executable if at least on of its options is executable
    - Chooses repeatedly an option non-deterministically and executed it
    - Terminates when a `break` or `goto` is executed
  - **Atomic**
    - Basic statements are executed atomically
      - Includes `skip, timeout, assert`, assignment, expression statement
    - `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
- **Macros**
  - Does not contain procedures
    - Can most of the time be achieved with macros
  - String replacement as in C
- **Channels**
  - Declare `chan ch = [d] of {t1, ..., tn}`
  - Buffer up to d messages
    - **d > 0:** FIFO buffer channel
    - **d = 0:** Rendez-vous unbuffered channel
  - Each message is a tuple whose elements are of type `t1, ..., tn`
  - **Buffered Channel:**
    - **Send Message**
      - `ch ! e1, ..., en`
      - Type of `ei` musts match type `ti` of channel declaration
      - Executable iff buffer is not full
    - **Receive Message**
      - `ch ? a1, ..., an`
      - `ai` is a variable or constant of type `ti`
      - Executable iff buffer is not empty and oldest message in the buffer matches the constants `ai`
      - Variables `ai` are assigned values of the message
  - **Unbuffered Channel:**
    - Models synchronous communication
    - **Send Message**
      - `ch ! e1, ..., en`
      - Executable is there is a receive operation that can be executed simultaneously
    - **Receive Message**
      - `ch ? a1, ..., an`
      - Executable if there is a send operation that can be executed simultaneously

13.2. Linear Temporal Logic (LTL)

- Many interesting properties relate several states
- **Transition System**
  - Slightly different from what we are used to

- ◇ Tuple $(\Gamma, \sigma_I, \rightarrow)$
  - ∗ **Γ:** Finite set of configurations
    - ◦ Different is that now it is finite
  - ∗ $\boldsymbol{\sigma_I}$**:** Internal configuration
    - ◦ $\sigma_I \in \Gamma$
    - ◦ Different is that we only consider the initial configuration
    - ◦ Terminal configuration can be modelled by introducing a special **sink state** which cannot be left again
  - ∗ **→:** transition relation
    - ◦ $\rightarrow \subseteq \Gamma \times \Gamma$
- ◇ **Promela Model**
  - ∗ Configurations are just states
    - ◦ We no not need statement since this is appointed by the location counter
  - ∗ The initial configuration is the initial state
    - ◦ Init process is active
    - ◦ Everything is initialized to zero-equivalent
  - ∗ Transition relation is defined by OS statements
  - ∗ Promela model has a finite number of states
    - ◦ Still very large, but finite

- **Computations**
  - ◇ S^ω is a infinite sequence of elements of the set S
    - ∗ $s_{[i]}$ is the i-th element is this sequence
    - ∗ Opposed to S^* which is a finite sequence
  - ◇ **Computation:** Infinite sequence $\gamma \in \Gamma^\omega$ of states for which:
    - ∗ $\gamma_{[0]} = \sigma_I$
    - ∗ $\gamma_{[i]} \rightarrow \gamma_{[i+1]}, i \geq 0$
  - ◇ $\mathcal{C}(TS)$ is the set of all computations of a transition system TS
- **Linear-Time Properties (LT-Properties)**
  - ◇ Limits the permitted computations of a transition system
  - ◇ $P \subseteq \Gamma^\omega$
  - ◇ $\text{TS} \models P \iff \mathcal{C}(\text{TS}) \subseteq P$
    - ∗ All computations of TS belong to the set P
  - ◇ LT-properties express properties of computations
    - ∗ Non-termination is handled by infinite sequences
    - ∗ Non-determinism is handled by considering each computation separately
  - ◇ Try to simplify it more
  - ◇ **Atomic Propositions (AP):** Set of properties we care about
    - ∗ Called atomic since they contain no logical connectives
  - ◇ **Labeling Function:** Maps configurations to sets of atomic propositions from AP
    - ∗ $L : \Gamma \rightarrow \mathcal{P}(AP)$
    - ∗ $\mathcal{P}$ is the powerset. Once configuration can be part of multiple APs
    - ∗ **Abstract State:** $L(\sigma)$ labeled state
  - ◇ We consider AP and L as part of the system
    - ∗ We have a 5-tuple instead of tripple
  - ◇ **Trace**
    - ∗ Abstraction of a computation
    - ∗ Infinite sequence of abstract states
      - ◦ $\mathcal{P}(\text{AP})^\omega$
    - ∗ $t \in \mathcal{P}(AP)^\omega$ is a trace of transition system TS if $t = L(\gamma_{[0]})L_{\gamma_{[1]}} \dots$ and γ is a

computation of TS
  * $\mathcal{T}$(TS) set of all traces of transition system TS
- ◇ **Safety Properties**
  * I.e. something bad is never allowed to happen
    - ◦ And once it happened, it cannot be fixed
  * LT-property P is a safety property if for all infinite sequences $t \in \mathcal{P}(\text{AP})^\omega$: if $t \notin P$, then there is a finite prefix $\hat{t}$ of t such that every infinite sequence t' with prefix $\hat{t}, t' \notin P$
  * **Bad Prefix $\hat{t}$:** Finite sequence which already violates the property
    - ◦ Even if the violation only happens after the sequence
  * Safety properties are violated in finite time
    - ◦ Even if the sequence is infinite
    - ◦ Can be tested
- ◇ **Liveness Properties**
  * I.e. something good will happen eventually
  * LT-property P is a liveness property if all finite sequences $\hat{t} \in \mathcal{P}(\text{AP})^*$ are a prefix of an infinite sequence $t \in P$
    - ◦ Every finite prefix can be extended to an infinite sequence which is in P
  * Liveness properties are violated in infinite time
    - ◦ Cannot be tested

- **Linear Temporal Logic (LTL)**
  - ◇ Logic which makes it easy to reason about LT-properties
  - ◇ Fully blown logic
  - \+ Whether or not a trace of a finite transition system satisfies an LTL formula is decidable
  - ◇ Reasons about traces and not single states
  - ◇ **Syntax:**
    * $\phi = p \mid \neg\phi \mid \phi \wedge \phi \mid \underbrace{\phi \text{U} \phi}_{\text{until}} \mid \underbrace{\bigcirc\phi}_{\text{next}}$
      - ◦ Where p is a proposition from a chosen set of atomic propositions AT $\neq \emptyset$
  - ◇ **Semantics:**
    * Trace $t \in \mathcal{P}(\text{AP})^\omega$ satisfies LTL formula ϕ: $t \models \phi$
    * $t_{(\geq i)}$ is the suffix of t starting at t_i
    *

$$
\begin{aligned}
t &\models p && \text{iff } p \in t_{[0]} \\
t &\models \neg\phi && \text{iff not } t \models \phi \\
t &\models \phi \wedge \psi && \text{iff } t \models \phi \text{ and } t \models \phi \\
t &\models \phi \text{U} \psi && \text{iff } \exists k \geq 0, t_{(\geq k)} \models \psi \text{ and } t_{(\geq j)} \models \phi \ \forall j, 0 \leq j < k \\
t &\models \bigcirc\phi && \text{iff } t_{(\geq 1)} \models \phi
\end{aligned}
$$

  - ◇ **Derived Operators:**
    * true, false, $\vee$, $\implies$, $\iff$ are defined as usual
    * **Eventually:** $\Diamond\phi \equiv$ (true Uϕ)
    * **Always from now:** $\Box\phi \equiv \neg\Diamond\neg\phi$
    * **Precedence:** Unary over binary
    * **Specification Patterns:**
      - ◦ **Strong Invariant:** $\Box\psi$
        - ▷ ψ always holds

- ▷ Safety property
- ○ **Monotone Invariant:** $\Box(\psi \implies \Box\psi)$
  - ▷ Once ψ is true, then ψ is always true
  - ▷ Safety property
- ○ **Establishing an invariant:** $\Diamond\Box\psi$
  - ▷ Eventually ψ will always hold
  - ▷ Liveness property
- ○ **Responsiveness:** $\Box(\psi \implies \Diamond\phi)$
  - ▷ Every time that ψ holds, ϕ will eventually hold
  - ▷ Liveness property
- ○ **Fairness:** $\Box\Diamond\psi$
  - ▷ ψ holds infinitely often
  - ▷ Liveness property

◇ **Model Checking**

- ∗ Given a finite transition system TS and a LTL formula ϕ, decide whether $t \models \phi$ for all $t \in \mathcal{T}(\text{TS})$
  - ○ I.e. $\mathcal{T}(\text{TS}) \subseteq P(\phi)$
- ∗ Hard because traces are in general infinite
- ∗ **Checking Safety Properties:**
  - ○ Violation is observed in an finite prefix
  - ○ **Idea:**
    - ▷ Characterize all finite prefixes of the traces of the transition system using a finite automata
      - · In TS labels are on the states where there are on the transittons for the FA
      - · FA is Tuple $(Q, \sum, \delta, Q_0, F)$
        - · Q**:** finite set of states
        - · $\sum$**:** finite alphabet
        - · δ**:** transition relation
          - · $\delta \subseteq Q \times \sum \times Q$
        - · $Q_0 \subseteq Q$**:** initial state
        - · $F \subseteq Q$**:** accepting states
      - · Given transition system $TS = (\Gamma, \sigma_I, \rightarrow)$ we define NFA $\mathcal{F}\,\mathcal{A}_{\text{TS}}$ characterizing all finite prefixes $\mathcal{T}_{\text{fin}}(\text{TS})$ of the traces of TS
      - · $\mathcal{F}\,\mathcal{A}_{\text{TS}} = (Q, \sum, \delta, Q_0, F)$
        - · $Q = \Gamma \cup \{\sigma_0\}, \sigma_0 \notin \Gamma$
        - · $\sum = \mathcal{P}(\text{AP})$
        - · $\delta = \{(\sigma, p, \sigma') \mid \sigma \rightarrow \sigma' \text{ and } p = L(\sigma')\} \cup \{(\sigma_0, p, \sigma_I) \mid p = L(\sigma_I)\}$
        - · $Q_0 = \{\sigma_0\}$
        - · $F = Q$
    - ▷ Check whether any of them violates the safety property
      - · Manual checking possible for simple FA
      - · Automatic checking not possible
  - ○ **Regular Safe Properties:**
    - ▷ Restriction
    - ▷ Safety property is regular if its bad prefixes are described by a regular language over the alphabet $\mathcal{P}(\text{AP})$
    - ▷ Every invariant over AP is a regular safety property
    - ▷ **Checking Regular Safety Properties:**

- Describe finite prefixes $\mathcal{T}_{\text{fin}}(\text{TS})$ by finite automate $\mathcal{F}\ \mathcal{A}_{\text{TS}}$
- Describe bad prefixes of regular safety property P by finite automata $\mathcal{F}\ \mathcal{A}_{\overline{P}}$
- Construct finite automata for product of $\mathcal{F}\ \mathcal{A}_{\text{TS}}$ and $\mathcal{F}\ \mathcal{A}_{\overline{P}}$
  - Product corresponds to the intersection of both FA TODO: describe construction
- Check if resulting automaton has any reachable accepting states
  - If not, property P is never violated in traces of TS
  - If yes, the property P is violated
    - Counterexample is any accepted word by the automata

- So far we can not check non-regular safety properties and liveness properties

* **ω-Regular Languages**
  - Denote languages of infinite works
  - Expression G has the form $G = E_1F_1^\omega + \cdots + E_nF_n^\omega$ $(1 \leq n)$
    - E_i and F_i are regular expression
    - $+$ means or
  - **Büchi Automata (NBA)**
    - Very similar to FA
    - Accept infinite words
    - Accepted language agrees with the class of ω-regular languages
    - Non-deterministic
    - Tuple $(Q, \sum, \delta, Q_0, F)$
      - Q**:** finite set of states
      - $\sum$**:** finite alphabet
      - δ**:** transition relation
        - $\delta \subseteq Q \times \sum \times Q$
      - $Q_0 \subseteq Q$**:** initial state
      - $F \subseteq Q$**:** accepting states
    - Accept word if it passes infinitely often through an accepting state
  - **Checking:**
    - Describe traces $\mathcal{T}(\text{TS})$ by NBA $\mathcal{B}\ \mathcal{A}_{\text{TS}}$
    - For an LTL formula ϕ, construct NBA $\mathcal{B}\ \mathcal{A}_{\neg\phi}$ that accepts the traces (i.e. the bad traces) characterized by $\neg\phi$
    - Construct NBA for products of $\mathcal{B}\ \mathcal{A}_{\text{TS}}$ and $\mathcal{B}\ \mathcal{A}_{\neg\phi}$
    - Check whether the language accepted by product NBA is empty
      - If language is non-empty, property ϕ is violated
      - Each word in the language is a counterexample
* **Complexity**
  - For a finite transition system TS and an LTL formula ϕ the model checking problem $\text{TS} \models \phi$ is solvable in $\mathcal{O}(|\text{TS}| \times 2^{|\phi|}$
    - $|\text{TS}|$: size of the transition system
      - Grows exponentially in the number of variables, processes and channels
    - $|\phi|$: size of ϕ
      - Grows exponentially tue to the construction of the $\mathcal{B}\ \mathcal{A}_{\neg\phi}$

Part III.
Appendix

14. Prelude

- `curry :: ((a, b) -> c) -> a -> b -> c`
  - Converts uncurried function to curried function
- `uncurry :: (a -> b -> c) -> (a, b) -> c`
  - Converts curried function to function on tuple
- `fromEnum :: a -> Int`
  - Gives ascii value of a char
- `toEnum :: Int -> a`
  - Gives character of a certain ascii value
- `abs :: Num => a -> a`
  - ABS value of number
- `signum :: Num => a -> a`
  - Returns $-1, 0$ or 1
- `foldMap :: Monoid m => (a -> m) -> t a -> m`
  - Map each element of the passed list to a monoid (array of one element) and apply the function on it
  - Example:
    - `foldMap (replicate 3) [1,2,3] = [1,1,1,2,2,2,3,3,3]`
- `foldr :: (a -> b -> b) -> b -> t a -> b`
- `foldl :: (b -> a -> b) -> b -> t a -> b`
- `elem :: Eq a => a -> t a -> Bool`
  - Does element occur in list
- `maximum :: Ord a => t a -> a`
  - Largest element of non-empty list
- `minimum :: Ord a => t a -> a`
  - Least element of non-empty list
- `sum :: Num a => t a -> a`
  - Sum of list
- `product :: Num a => t a -> a`
  - Product of list
- `(.) :: (b -> c) -> (a -> b) -> a -> c`
  - Function composition
- `flip :: (a -> b -> c) -> b -> a -> c`
  - Takes function with two arguments and applies the arguments in switched order
- `($)`
  - Useful to omit parentheses
  - Example: `f $ g $ h x = f ( g ( h x))`
- `until :: (a -> Bool) -> (a -> a) -> a -> a`
  - Yields the result of applying the function until the condition hold
- `map :: (a -> b) -> [a] -> [b]`
- `(++) :: [a] -> [a] -> [a]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `head :: [a] -> a`
  - First element of the list
- `last :: [a] -> a`
  - Last element of the list
- `tail :: [a] -> [a]`
  - All except the first element of the list

- `init :: [a] -> [a]`
  - All expect the last
- `(!!) :: [a] -> Int -> a`
  - Returns the n-th element of a list
- `null :: Foldable t => t a -> Bool`
  - Test whether list is empty
- `length :: Foldable t => t a -> Int`
  - Length of list
- `reverse :: [a] -> [a]`
  - Reverses given list
  - Only works for finite list
- `and :: Foldable t => t Bool -> Bool`
  - Return true iff all elements in the list are true
- `or :: Foldable t => t Bool -> Bool`
  - Return true iff at least one element of the list is true
- `any :: Foldable t => (a -> Bool) -> t a -> Bool`
  - Check if any element of the list satisfies the predict
- `all :: Foldable t => (a -> Bool) -> t a -> Bool`
  - Check if all element of the list satisfies the predict
- `concat :: foldable t => t [a] -> [a]`
  - The concatenation of all the elements of a container of lists
  - Example: `concat [[1,2,3],[4,5],[6],[] = [1,2,3,4,5,6]`
- `concatMap :: Foldable t => (a -> [b]) -> t a -> [b]`
  - Example: `concatMap (take 3) [[1..],[10..],[100..]] = [1,2,3,10,11,12,100,101,102]`
- `scanl :: (b -> a -> b) -> b -> [a] -> [b]`
  - Similar to `foldl` but gives intermediate results
  - Example: `scanl (+) 0 [1..4] = [0,1,3,6,10]`
  - Example: `scanl (-) 100 [1..4] = [100,99,97,94,90]`
- `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
  - Similar to `foldl` but gives intermediate results
  - Example: `scanl (+) 0 [1..4] = [10,9,7,4,0]`
  - Example: `scanl (-) 100 [1..4] = [98,-97,99,-96,100]`
- `iterate :: (a -> a) -> a -> [a]`
  - Infinitely often apply the function to the value
  - Example: `iterate (+3) 42 = [42,45,48,51,54,...]`
- `repeat :: a -> [a]`
  - Repeat the value in an infinite list
  - Example: `repeat 0 = [0,0,0...]`
- `replicate :: Int -> a -> [a]`
  - Create list containing x n times
  - Example: `replicate 4 True = [True, True, True, True]`
- `cycle :: [a] -> [a]`
  - Create infinite list from given list
  - Example: `cycle [1,2] = [1,2,1,2,1,2,...]`
- `take :: Int -> [a] -> [a]`
  - Returns prefix of length n or xs if n is larger than its size
  - Example: `take 3 "test" = "tes"`
- `drop :: Int -> [a] -> [a]`
  - Returns suffix after the first n elements or [] if n is larger than length of list

  - Example: `drop 3 "test" = "t"`
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
  - Returns longest prefix of list that all satisfy the predicate
  - Example: `takeWhile (<3) [1..5] = [1,2]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
  - Returns suffix after applying `takewhile`
  - Example: `dropWhile (<3) [1..5] = [3,4,5]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
  - Tuple of `takeWhile` and `dropWhile`
  - Example: `span (<3) [1..5] = ([1,2],[3,4,5])`
- `splitAt :: Int -> [a] -> ([a], [a])`
  - Split list at n (first element is n long)
  - Example: `splitAt 3 "Test" = ("Tes", "t")`
- `zip :: [a] -> [b] -> [(a, b)]`
  - Combines two list into tuple
  - Final length is length of the shorter list
- `zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]`
- `zipWith :: (a -> b -> c) -> [a] -> [b]  -> [c]`
  - Example: `zipWith (+) [1,2,3] [4,5,6] = [5,7,9]`
  - `zipWith3` also exists
- `unzip :: (a, b) -> [a] -> [b]`
- `unzip3 :: (a, b, c) -> [a] -> [b] -> [c]`
- `show :: a -> String`
  - Convert anything to string
- `read :: Read a => String -> a`
  - Convert string to anything
  - Often we need to give the type
  - Example: `read "123" :: Int = 123`

15. Data.List

- `intersperse :: a -> [a] -> [a]`
  - Example: `intersperse ',', "abcdef" = "a,b,c,d,e,f"`
- `tranpose :: [[a]] -> [[a]]`
  - Can be useful in concussion with infinite list
- `subsequences :: [a] -> [[a]]`
  - Powerset of given set (/list)
  - Example: `subsequences "abc" = ["", "a", "b", "c", "ab" "ac", "cd", "abc"`
- `permutations :: [a] -> [[a]]`
  - Example: `permutations "abc" = ["abc, "bac", "cba", "cab", "acb"]`
- `group :: Eq a => [a] -> [[a]]`
  - Split list into sublist where each elements contains only the same element
  - Example: `group "Mississippi" = ["M", "i", "ss", "i", "ss", "i", "pp", "i"]`
- `isPrefixOf :: Eq a => [a] -> [a] -> Bool`
- `isSuffixOf :: Eq a => [a] -> [a] -> Bool`
- `isInfixOf :: Eq a => [a] -> [a] -> Bool`
- `isSubsequenceOf :: Eq a => [a] -> [a] -> Bool`
  - If all elements of the first list are present in the second
- `nub :: Eq a => [a] -> [a]`
  - Convert list into set by removing duplicates
- `(\\)) :: Eq a => [a] -> [a] -> [a]`
  - Set difference
- `union :: Eq a => [a] -> [a] -> [a]`
- `intersect :: Eq a => [a] -> [a] -> [a]`
- `sort :: Ord a => [a] -> [a]`