Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Formal Methods & Functional Programming

Roman Böhringer

June 8, 2019

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Functional Programming

1.1 Introduction, syntax

1.1.1 Basic concepts in functional programming

In functional programming, functions compute values but they are also values. It is possible to compute and return them. Functions have no side effects, meaning $f(x)$ always returns the same value (in contrast to e.g. object-oriented languages where this is not the case). Since functions have no side effects, it's possible to reason as in mathematics and it leads to a property called *referential transparency*: An expression is called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior. Furthermore, functional programs are easy to parallelize as computations cannot interfere.

Instead of iteration, recursion is used, which can be demonstrated by this simple *gcd* algorithm:

Listing 1.1: Java *gcd* implementation

```
1 public static int gcd (int x, int y) {
2   while (x != y) {
3     if (x > y) x = x - y;
4     else y = y - x;
5   }
6   return x;
7 }
```

Listing 1.2: Haskell *gcd* implementation

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

A flexible type system avoids many kinds of programming errors and polymorphism supports reusability, e.g. a *sort* function that can be used for numbers and strings.

1.1.2 History

The Lambda calculus was invented by Alonzo Church in the 1930s. It provides a theoretical framework for describing functions and their evaluations. It is equivalent to Turing machines in terms of computational power.

In the 1960s, LISP (List processor) was developed by McCarthy for symbolic computing / AI applications. Lists were the basic data structure and the language was untyped.

In 1978, John W. Backus wrote the paper "Can Programming be Liberated from the von Neumann Style?" which contained many important ideas about Functional Programming.

In 1992, Haskell was introduced as a State-of-art functional programming language with an advanced type system, efficient interpreters and compilers, a huge library, etc... It influenced many other programming languages.

1.1.3 Haskell syntax

Functions in Haskell consist of different cases:

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

A program consists of several definitions.

The indentation determines separation of definitions, meaning:

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

The recommended code layout is:

```
f1 x1 x2
  | a long guard which may go over
    a number of lines
      = a long expression that also can go over
        several lines
  | g2  = e2
f2 x1 x2 x3 = ...
```

Haskell is a strongly typed language, either the programmer provides types along with the function definition (e.g. $gcd :: Int- > Int- > Int$) or the system computes types itself.

Haskell provides an *Int* type with at least the range $\{-2^{29}, \ldots, 2^{29} - 1\}$. It also provides an arbitrary precision *Integer* type that will hold any number no matter how big. Multiple functions for numbers are provided, namely: $+, *, \hat{}, -, div, mod, abs$ (usually defined for instances of the *Num* type class). Backtick turns a name to an infix operator, i.e. $7\ `mod`\ 2$ is equivalent to $mod\ 7\ 2$. Infix binary functions are also called operators. Operators can also be written in prefix notation, e.g. $(+)\ 3\ 4$.

The order and equality operators are identical to other programming languages, except for inequality, where Haskell uses $/=$.

These operators return *True* or *False* of type *Bool*. Haskell provides the binary operators &&, ||, and the unary function *not* that work as expected.

Further types are *Char* ('*a*', '*b*', ...), *String* (e.g. *"Hello"*; which are list of chars) and *Double* (0.42).

Another type are tuples. They are defined like this:

- If $T_1, \ldots, T_n$ ($n \geq 2$) are types, then $(T_1, \ldots, T_n)$ is a (tuple) type, e.g. $(Int, String, Bool)$
- If $v_1 :: T_1, \ldots, v_n :: T_n$, then $(v_1, \ldots, v_n) :: (T_1, \ldots, T_n)$

Because of this definition, nesting of tuples is possible.

Haskell supports patterns in function definitions:

```
1 fun m1 m2 ... mn
2   | g1          = e1
3     :
4   | gm          = em
5   | otherwise = e    -- optional!
```

The patterns *mi* are variables, constants or built from data constructors (like tuples) while the guards *gi* are Boolean expressions.

If no explicit scoping mechanisms are used, functions can be called from any other (i.e. global scope). Local scopes can be defined with *let* and *where*:

Listing 1.3: Local scope using *let*

```
1 let x1 = e1
2       :
3     xn = en
4 in e
```

Listing 1.4: Local scope using *where*

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

A *where* clause is only allowed at the top level of a set of equations or case expression. The same properties and constraints on bindings in *let* expressions apply to those in *where* clauses. These two forms of nested scope seem very similar, but a *let* expression is an expression, whereas a *where* clause is not – it is part of the syntax of function declarations and case expressions. The bindings in *where* clauses are defined over all guards. For both *where* clauses and *let* expressions, it is possible to bind variables or (local) functions. In the following example, it's only possible to evaluate $f\ 10$ but not $sq\ 10$:

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

The 2D layout rules apply for *let* and *where* as well, i.e.:

- The first definition determines indentation for all definitions in block.
- Multi-line definitions must indent more.
- Less indentation closes block.

Pattern matching in Haskell has two purposes. It checks if an argument has the proper form and binds values to variables. For example $(x : xs)$ matches with $[2,3,4] = 2 : [3,4]$. Patterns are inductively defined:

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

Patterns are required to be linear which means that each variable can occur at most once. $(x + +y, z)$ or $[x, y, z, x]$ are not allowed. Some example functions using pattern matching are listed below.

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

```
f (False, False) = False
f (_, _) = True
```

The following steps are a general advice on how to define recursive functions:

1. Define the type
2. Enumerate the cases
3. Define the simple cases
4. Define the other cases
5. Generalize and simplify

The `$` sign is used to avoid parenthesis. Anything appearing after it will take precedence over anything that comes before, which is implemented by the simple definition `f $ x = f x`. Therefore, `putStrLn (show (1 + 1))` can be rewritten to `putStrLn $ show $ 1 + 1`

1.2 Logic, proofs, correctness

Formal reasoning about systems requires a language, semantics and a deductive system for carrying out proofs. These are related by metatheorems, e.g. soundness and completeness.

1.2.1 Natural deduction

In logic and proof theory, natural deduction is a kind of proof calculus in which logical reasoning is expressed by inference rules closely related to the "natural" way of reasoning.

Rules are used to construct derivations under assumptions. $A_1, \ldots, A_n \vdash A$ means that "A follows from $A_1, \ldots, A_n$". Derivations are trees and a proof is a derivation whose root (i.e. the bottom) has no assumptions. A proof of A is therefore a derivation tree with root $\vdash A$. If the logic is sound, then A is a tautology.

The following rule denotes "If $+$, then $\times$": $\dfrac{\Gamma \vdash +}{\Gamma \vdash \times}$

The following rule says "If A and B, then C":

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash C}$$

And with the following rule we can express: "We may assume D when proving E": $\dfrac{\Gamma, D \vdash E}{\Gamma \vdash E}$

An Axiom is written like this: $\dfrac{}{\Gamma, A \vdash A}$

For instance, with the following proof rule the principle of induction on Natural numbers is formulated (with the side condition, that n is not free in Γ):

$$\frac{\Gamma \vdash P(0) \qquad \Gamma, P(n) \vdash P(n+1)}{\Gamma \vdash \forall n \in \mathbb{N}.P(n)}$$

1.2.2 Propositional Logic

Propositions are built from a collection of variables and closed under disjunction, conjunction, implication. This means more formally: Let a set $\mathcal{V}$ of variables be given. $\mathcal{L}_P$, the language of propositional logic, is the smallest set where:

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

A valuation $\sigma : \mathcal{V} \to \{$ True, False $\}$ is a function that maps variables to truth values. They are simple kinds of models / interpretations. Satisfiability is the smallest relation $\vDash \subseteq$ Valuations $\times \mathcal{L}_P$ such that:

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

We have $\sigma \nvDash \bot$ for all $\sigma \in$ Valuations

A formula $A \in \mathcal{L}_P$ is *satisfiable* if $\sigma \vDash A$ for some valuation σ.

A formula $A \in \mathcal{L}_P$ is *valid* (a tautology) if $\sigma \vDash A$ for all valuations σ.

Semantic entailment is defined as follows: $A_1, \ldots, A_n \vDash A$ if for all σ, if $\sigma \vDash A_1, \ldots, \sigma \vDash A_n$ then $\sigma \vDash A$.

The rules of natural deduction for propositional formulae are listed in the appendix of this document. These are only for intuitionistic logic. Therefore

certain theorems like Peirce's Law can't be proven: $((A \to B) \to A) \to A$ Classical logic requires either:

- Axiom of excluded middle ("tertium non datur"): $\dfrac{}{\Gamma \vdash A \vee \neg A}$
- "Reductio ad absurdum": $\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}$

A heuristic for constructing (backwards) ND proofs is to apply safe rules first. A rule is safe if we only enlarge Γ or can get the conclusion back. For instance $\wedge - I$ or $\vee - E +$ axiom is safe. $\wedge - EL$ is unsafe, there's no way to get back to (only) $\Gamma \vdash A$.

1.2.3 Requirements for a deductive system

In a deductive system, syntactic entailment $\vdash$ (derivation rules) and semantic entailment $\vDash$ (truth tables) should agree. This has two parts:

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

1.2.4 First-Order Logic

First-Order logic consists of terms and formulae. A signature consists of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$ (and their arities). We write f^i (or p^i) to indicate the function symbol f (or predicate symbol p) has arity $i \in \mathbb{N}$. Constants are 0-ary function symbols.

Let $\mathcal{V}$ be a set of variables. Then, the *terms* of first-order logic, is 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 for all $1 \leq j \leq n$.

The *formulae* of first-order logic is the smallest set where

- $\bot \in$ Form
- $p^n(t_1, \ldots, t_n) \in$ Form and $t_j \in$ Term for sll $1 \leq j \leq n$.
- $A \circ B \in$ Form if $A \in$ Form and $B \in$ Form , and $\circ \in \{\wedge, \vee, \to\}$
- $Qx.A \in$ Form if $A \in$ Form , $x \in \mathcal{V}$ and $Q \in \{\forall, \exists\}$

Each occurence of every variable in a formula is bound or free. A variable occurence x in a formula A is bound if x occurs withn a subformula of A of the form $\exists x.B(x)$ or $\forall x.B(x)$.

Names of bound variables are irrelevant, they just encode the binding structure. Therefore, bound variables can be renamed at any time. This process

is called *α-conversion*. While renaming, the binding structure must be preserved.

The following is an allowed α-conversion:

$$\forall x.\exists y \cdot p(x,y) \quad \text{to} \quad \forall y.\exists x.p(y,x)$$

Whereas this is **not** allowed:

$$p(x) \to \forall x.p(x) \quad \text{to} \quad p(y) \to \forall y \cdot p(y)$$

For free variables, the convention is to write $A(x,y)$ for a formula where the variables x, y may occur free, but also other variables may occur free. Just A may also have implicit free variables x, y, z. But bound variables may not capture implicit free ones, i.e. $\exists z.(A(x,y) \wedge B(z))$ implies that z is not free in $A(x,y)$.

Syntactic rules

The $\to$ operator is right-associative. This means that $X \to Y \to Z$ is interpreted as $X \to (Y \to Z)$. $\wedge$ and $\vee$ associate to the left.

Furthermore, $\wedge$ binds stronger than $\vee$ which binds stronger than $\to$. Negation binds stronger than binary operators and quantifiers extend to the right as far as possible (end of line or ")"). They override the binding of binary operators.

Some examples:

$$\begin{aligned} A \vee B \wedge \neg C \to A \vee B &\equiv (A \vee (B \wedge (\neg C))) \to (A \vee B) \\ A \to B \vee A \to C &\equiv A \to ((B \vee A) \to C) \\ A \wedge \forall x.B(x) \vee C &\equiv A \wedge (\forall x.(B(x) \vee C)) \end{aligned}$$

Semantics

A structure is a pair $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$ where $U_\mathcal{S}$ is a nonempty set, the universe, and $I_\mathcal{S}$ is a mapping where

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

As a shorthand, one can write $p^\mathcal{S}$ for $I_\mathcal{S}(p^n)$ and $f^\mathcal{S}$ for $I_\mathcal{S}(f^n)$

An interpretation is a pair $\mathcal{I} = \langle \mathcal{S}, v \rangle$, where $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S} \rangle$ is a structure and $v : \mathcal{V} \to U_\mathcal{S}$ a valuation. The value of a term t under the interpretation $\mathcal{I}$ is written as $\mathcal{I}(t)$ and defined by:

- $\mathcal{I}(x) = v(x)$ for $x \in \mathcal{V}$

- $\mathcal{I}\left(f\left(t_1,\ldots,t_n\right)\right) = f^{\mathcal{S}}\left(\mathcal{I}\left(t_1\right),\ldots,\mathcal{I}\left(t_n\right)\right)$

The satisfiability relation $\vDash \subseteq$ Interpretations $\times$ Form is the smallest relation satisfying all the properties of propositional logic and:

- $\langle \mathcal{S}, v\rangle \vDash p\left(t_1,\ldots,t_n\right)$ if $\left(\mathcal{I}\left(t_1\right),\ldots,\mathcal{I}\left(t_n\right)\right) \in p^{\mathcal{S}}$
- $\langle \mathcal{S}, v\rangle \vDash \forall x.A$ if $\langle \mathcal{S}, v[x \mapsto a]\rangle \vDash A$ for all $a \in U_{\mathcal{S}}$
- $\langle \mathcal{S}, v\rangle \vDash \exists x.A$ if $\langle \mathcal{S}, v[x \mapsto a]\rangle \vDash A$ for some $a \in U_{\mathcal{S}}$

$v[x \longmapsto a]$ denotes the valuation v' that is identical to v except that $v'(x) = a$.

When $\langle \mathcal{S}, v\rangle \vDash A$ we say that A is satisfied with respect to $\langle \mathcal{S}, v\rangle$ or that $\langle \mathcal{S}, v\rangle$ is a model of A. When A doesn't have free variables, we write $S \vDash A$ because satisfaction doesn't depend on v.

When every suitable interpretation is a model, we write $\vDash A$ and say that A is valid. An example for a valid formula is:

$$(\forall x.p(x,x)) \to p(a,a)$$

A is satisfiable if there is at least one model for A. The following formula is satisfiable and satisfied with the rationals:

$$\forall x.\exists y.y * 2 = x$$

Substitution is the process of replacing all occurrences of a free variable x with some term t. All free variable of t must still be free in $A(t)$, i.e. capture must be avoided. If necessary, A can be α-converted before substitution.

First-Order Logic with Equality

Equality is a logical symbol with associated proof rules. In the extended languages, we have $t_1 = t_2 \in$ Form if $t_1, t_2 \in$ Term. The definition of semantic entailment is extended as follows:

$$\mathcal{I} \vDash t_1 = t_2 \quad \text{if} \quad \mathcal{I}\left(t_1\right) = \mathcal{I}\left(t_2\right)$$

Equality is an equivalence relation and therefore satisfies reflexivity, symmetry and transitivity. This can be used explicitly as derivation rules but is also often used implicitly / in a linear equational fashion in proofs.

1.2.5 λ-calculus

In the λ-calculus, we call the renaming of bound variables also α-conversion. One has to take care of the same things as in first order logic (no capturing of free variables).

The essence of evaluation in the λ-calculus is the β-reduction. A term $(\lambda x.M)\ N$ is called a redex. β-reduction is the rule for simplifying redexes:

$$(\lambda x.M)N \hookrightarrow M[x \leftarrow N]$$

The result $M[x \leftarrow N]$ is called the contractum. Evaluation fixes a strategy for reducing redexes. The following rules exist for β-reduction:

1. $x[x \leftarrow N] = N$
2. $y[x \leftarrow N] = y$ if $y \neq x$
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N])(Q[x \leftarrow N])$
4. $(\lambda x.P)[x \leftarrow N] = \lambda x.P$
5. $(\lambda y.P)[x \leftarrow N] = \lambda y.(P[x \leftarrow N])$ if $y \neq x$ and $y \notin free(N)$
6. $(\lambda y.P)[x \leftarrow N] = \lambda z.((P[y \leftarrow z])[x \leftarrow N])$ if $y \neq x$ and $y \in free(N)$, where $z \notin free(NP)$

When evaluating $t_1\ t_2$ (the first application of a function to an argument), first $t_1 : t_1 \hookrightarrow r_1$ is evaluated. If $r_1 \neq \lambda x.r$, an error is thrown. Otherwise, β-reduction is applied to $r_1\ t_2$

$$(\lambda x.r)t_2 \hookrightarrow r\,[x \leftarrow t_2]$$

and the result is then further evaluated. Depending on the evaluation strategy, there are differences:

- **Eager evaluation:** t_2 is evaluated prior to the β-reduction. Furthermore, if there's an abstraction $(\lambda x.t)$, the evaluation is carried out under it.
- **Lazy evaluation:** t_2 is substituted without evaluation and there is no evaluation under an abstraction.

Arithmetic functions can be represented in the λ-calculus which is done by giving a total, computable, injective function $\bar{\cdot}$ that maps $\mathbb{N}$ to λ-terms, for instance the Church numerals:

$$\overline{n} = \lambda sz \cdot s^n(z) \quad \text{e.g.} \quad \overline{2} = \lambda sz.s(sz)$$

Addition is then represented by $\lambda x\, y\, s\, z\, .xs(ysz)$

There's also η-conversion which is adding or dropping of abstractions over a function. For instance, $\lambda x.\, abs\ x$ and abs are equivalent under η-conversion.

1.2.6 Correctness

Termination

If f is defined in terms of functions $g_1, \ldots, g_k$ ($g_i \neq f$) and each g_i terminates, then so does f. If we have recursion (e.g. $g_i = f$), a sufficient condition for

termination is that the arguments are smaller along a well-founded order on the function's domain. An order $>$ on a set $\mathcal{S}$ is well-founded iff there is no infinite decreasing chain $x_1 > x_2 > x_3 > \ldots$, for $x_i \in \mathcal{S}$ (e.g. $>_{\mathcal{N}}$, but not $>_{\mathcal{Z}}$).

New well-founded relations can be constructed from existing ones. If $>$ is a well-founded order on the set S, then $>^+$ ($a\ R^+\ b$ iff $a\ R^i\ b$ for some $i \geq 1$) is also well-founded on S.

Equational Reasoning / Induction

Proofs are based on the simple idea that functions are equations and therefore done in first-order logic with equality. To proof properties for all numbers, induction is used. Besides natural induction, noetherian induction can be used. The schema is as follows: $P(n)$ is proven for an arbitrary n under the assumption that $P(m)$ holds, for all $m < n$. Any well-founded ordering $<$ can be used.

1.3 Lists

List types are a type constructor. If T is a type, then $[T]$ is a type. The elements of $[T]$ can be:

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

As a short hand, $1 : (2 : (3 : []))$ is usually written as $[1,2,3]$. Strings are just lists of characters, i.e. $['a','a','b'] == ''aab''$. Haskell supports various abbreviations, e.g. $[3..6]$ for $[3,4,5,6]$ or more generally $[n, p..m]$ which means count from n to m in steps of $p - n$.

When specifying functions on list, one must specifiy how to compute with the empty list [] and how to compute with the non-empty list $(x : xs)$, e.g.:

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

$++$ is used for appending (type `[a] -> [a] -> [a]`) whereas : is used for adding a new element (type `a -> [a] -> [a]`).

The function *zip* takes two lists and creates pairs of the elements (extra elements are discarded), i.e. $zip[2,3,4][4,5,78] = [(2,4),(3,5),(4,78)]$ *zipWith* applies a binary function to the pairs instead of creating a tuple. The definition is:

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

List comprehension is a notation for sequential processing of list elements. It is analogous to set comprehension in set theory, e.g. $\{2 \cdot x | x \in X\}$. The Haskell notation is `[2 * x | x <- xs]`. List comprehensions can be augmented with guards: `[2*x | x <- xs, pred1(x), ...]` For example, the list comprehension `[2*x | x <- [0,1,2,3,4,5,6], x 'mod' 2 == 0, x > 3]` returns $[8, 12]$.

To proof a property $P(xs)$ for all xs in $[T]$, structural induction is used. It works as follows:

- **Base case:** prove $P([])$
- **Step case:** prove $P(x : xs)$ under the assumption $P(xs)$ for an arbitrary $xs :: [T]$ and $x :: T$. This means the induction hypothesis is: $xs :: [T]$, $x :: T$ and $P(xs)$

Sometimes a property needs to be generalized (i.e. stated for all elements instead of a specific, e.g. for all lists instead of only those with one element) so that it can be used in a proof.

1.4 Abstraction, higher-order programming

We differ functions by their order:

- **First order:** Arguments are base types or constructor types, e.g. `Int -> [Int]`
- **Second order:** Arguments are themselves functions, e.g. `(Int -> Int) -> [Int]`
- **Third order:** Arguments are functions, whose arguments are functions, e.g. `((Int -> Int) -> Int) -> [Int]`
- **Higher-order functions:** Functions of arbitrary order.

There are multiple advantages of using higher order functions:

- The definition is easier to understand
- Parts are easier to modify
- Parts are easier to requires
- Correctness is simpler to understand and show

map is a higher-order (second order) function that applies a function f to every element in a list:

```
1 map :: (a -> b) -> [a] -> [b]
2 map f [] = [] -- higher order as
3 map f (x:xs) = f x : map f xs --function f is an argument
```

map is similar to list comprehension, we have: `map f xs = [f x | x <- xs]`

Another higher-order (second order) function is *foldr* with the following type: `foldr :: (a -> b -> b) -> b -> [a] -> b` *foldr* begins at the right-hand end of the list and combines each list entry with the accumulator value using the given function. The result is the final value of the accumulator after "folding" in all the list elements and the starting value for the accumulator has to be provided by the user:

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

Its definition is:

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

Another way of thinking about *foldr* is to look at Haskells list representation, i.e. `[1,2,3,4,5] = 1:(2:(3:(4:(5:[])))) . foldr f x` then replaces each `:` with `f` (in infix notation) and `[]` with `x` and evaluates the result. Therefore, we have: `foldr (+) 0 [1,2,3] = foldr (+) 0 1:(2:(3:[])) = 1+(2+(3+0))` Various standard Haskell functions can be defined using *foldr*, some examples are:

```
1 concat xs = foldr (++) [] xs
2 and bs = foldr (&&) True bs
3 or bs = foldr (||) False bs
```

There's also *foldl* for left-associative folds:

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

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

A third useful higher-order (second order) function is *filter* that filters a list by a given function:

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter p [] = []
3 filter p (x:xs)
4     | p x       = x : filter p xs
5     | otherwise =     filter p xs
```

Filter can be implemented using list comprehension as well: `filter p xs = [x | x <- xs, p x]`

Haskell provides notation for writing λ-expressions. The λ symbol is replaced by a \ and the . by a ->. Therefore, one can write `(\x -> 2 * x)` or `(\x xs -> xs ++ [x])`. `(\x xs -> xs ++ [x])` is a shorthand for `(\x -> \xs -> xs ++ [x])`

Haskell provides a special function `(.)` for function composition with the following definition:

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

It is also possible to return functions as values, e.g.:

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

Functions of multiple arguments can be partially applied and return then another function. For example, given a function `multiply :: Int -> Int -> Int` we have `:type multiply 7` is equal to `Int -> Int`. More formally: If $f :: t_1 \to t_2 \to \cdots \to t_n \to t$ and $e_1 :: t_1, \ldots, e_k :: t_k$ then $f e_1 \ldots e_k :: t_{k+1} \to \cdots \to t_n \to t$.

All haskell functions take exactly one argument, `Int -> Int -> Int` means `Int -> (Int -> Int)` and `multiply 2 3` means `(multiply 2) 3`. Instead of "multiple arguments", functions can also be defined with tuples as arguments. There are two functions *curry* and *uncurry* to "convert" between those two formats:

```
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a,b) -> c
curry f    = f' where f' x1 x2  = f (x1,x2)
uncurry f' = f  where f (x1,x2) = f' x1 x2
```

Like one can see in the definition, *curry* takes a function that expects a tuple and returns the same function but now it expects two parameters. We call a function that takes multiple arguments (one at a time) a curried function.

1.4.1 Example: Difference lists

A difference list is a function `[a] -> [a]` that prepends a list to its argument. It is defined like this:

```
type DList a = [a] -> [a]
empty :: DList a
empty = \xs -> xs

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

```

```
8  app :: DList a -> DList a -> DList a
9  ys `app` zs = \xs -> ys (zs xs)
10
11 fromList :: [a] -> DList a
12 fromList ys = \xs -> ys ++ xs
13
14 toList :: DList a -> [a]
15 toList ys = ys []
```

Appending to the end of a difference list takes $\mathcal{O}(1)$, getting the head takes $\mathcal{O}(n)$ where $n =$ number of appends and getting the remaining elements is $\mathcal{O}(1)$ each.

1.5 Type classes and polymorphism

1.5.1 Polymorphism

Parametric polymorphism allows a function or a data type to be written generically, so that it can handle values uniformly without depending on their type. In Haskell, a function can have e.g. `[t] -> Int`, where t stands for all types t.

Parametric polymorphism differs from subtyping polymorphism where methods can be applied to objects only of sub-classes.

1.5.2 Type checking

We call a type w for f most general (also called principal) type iff for all types s for f, s is an instance of w. Haskell has algorithms for type checking and type reconstruction.

Type checking should prevent "dangerous expressions" (that result in a run-time error) like $2 + \mathit{True}$. The problem of which expressions are good (i.e. non-dangerous) is undecidable. Type systems are therefore in practice conservative and only type a subset of good expressions while also rejecting some good ones.

"Mini-Haskell" is used for examining type checking. In this language, programs are terms:

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

The core of the language is the λ-calculus with variables, abstraction and application. Syntactic sugar like omitting parenthesis is employed, i.e.:

$$x\, y\, z \text{ instead of } ((x\, y)\, z)$$
$$\tau_1 \to \tau_2 \to \tau_3 \text{ instead of } (\tau_1 \to (\tau_2 \to \tau_3))$$

For the types we have ($\mathcal{V}_\mathcal{T}$ is a set of type variables $a, b, \ldots$):

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

The type system notation is based on typing judgements: $\Gamma \vdash t :: \tau$ Where: Γ is a set of bindings $x_i : \tau_i$, mapping variables to types, t is a term and τ is a type. The proof rules are formulated in terms of type judgements and the proofs are built from rules and axioms. The proof rules for Mini Haskell are listed in the appendix.

1.5.3 Type reconstruction / inference

The goal of type inference is to infer the most general type. This is done with the following three steps:

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

Some terms are untypeable, in which cases type inference fails to build inference tree or solve constraints. Self application (applying a function f to itself) is not typeable, for example the term $\lambda f.\, f\, f$

There's a correspondence between propositions and types (Curry-Howard isomorphism). The type constructor "$\to$" corresponds to propositional logic "$\to$" and atomic types correspond to propositional variables. The rules correspond to those for (minimal) propositional logic.

1.5.4 Type Classes

Some functions are monomorphic (e.g. `(\x y -> x || y)`) while others are polymorphic. But there's also a "middle way" called type classes. For instance, if we take a look at the following function: `allEqual x y z = (x == y) && (y == z)` All elements that are passed to it have to be "comparable with respect to equality".

Type classes restrict polymorphism using class constraints. A class defines a set of types, for instance Eq is the equality class. We have $Int \in Eq$ but $Int \to Int \notin Eq$. The class is defined like that:

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4   x /= y = not (x==y)
```

The definition includes the class name (*Eq*), the signature (a list of function names and types) and optional a default implementation that can be overwritten later.

Class constraints in a type declaration are written before a `=>`, e.g.: `allEqual :: Eq t => t -> t -> t -> Bool` Multiple constraints are written in the following "tuple-notation": `f :: (Eq a, Num b) => a -> b`

The elements of the class are called instances. They are declared using `instance` and are built by "interpreting" (declaring) the signature functions, for example:

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

Class membership can depend on membership for other types (e.g. when $[t]$ is an instance of a class because t is).

Classes themselves can also depend on type conditions (derived classes), e.g.:

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

If *a* belongs to *Ord*, then it must also belong to *Eq*. This allows a hierarchical structuring of classes.

Type classes are somewhat similar to interfaces / abstract classes in object-oriented languages like Java.

The execution of parametric polymorphic functions is independent of the type of its arguments. Classes implement "ad hoc" polymorphism where the operation depends on the argument types. The selection of the actual function is done during compilation (if argument types are statically known) or at run time using "look-up" tables.

Some important type classes are:

- *Show* with functions *show* (`a -> String`) and *showList* (`[a] -> String`)
- *Read* which is required by the function *read* (`read :: Read a => String -> a`))
- *Foldable* with function $foldr$, i.e.:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

1.6 Algebraic data types

Besides base types (*Int*, *Bool*), compound types (tuples, lists, functions) and type synonyms (e.g. `type Complex = (Double, Double)`), there are also algebraic data types.

Enumeration types (disjoint unions) start with the keyword `data`. The names are different (uniquely named) constructors where the first letter of each constructor must be upper-case. With enumeration types, a set is defined (e.g. Season $= \{$ Spring, Summer, Fall, Winter $\}$). Some examples are:

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

Functions can be written using pattern matching:

```
whichSeason :: Month -> Season
whichSeason January = Winter
whichSeason February = Winter
whichSeason March = Spring
```

Product types start with the keyword `data` as well. They contain multiple attributes, e.g.

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

The constructors are functions (`Person :: Name -> Age -> People`). Functions may be defined by pattern matching, e.g.:

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

Enumeration and product types can be combined, e.g.: `data Shape = Circle Double | Rectangle Double Double` The functions are again definable by pattern matching:

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

Class instances can be explicitly created, e.g. for `data Foo = D1 | D2 | D3` one would write

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

In some cases, it's also possible to automatically derive class instances:

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

The general definition is:

$$\begin{aligned} data\ T = &\ Constr_1\ T_{11} \ldots T_{1_{k_1}} \\ &| Constr_2\ T_{21} \ldots T_{2_{k_2}} \\ &: \\ &| Constr_n\ T_{n1} \ldots T_{n_{k_n}} \end{aligned}$$

T_{ij} are types, possibly also containing T (i.e. recursion is allowed). T can have type variables as arguments. A recursive type is for instance:

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

Where an evaluator would be written like this:

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

Some examples with type variables:

```
data Pair t = MkPair t t

MkPair 2 3 :: Pair Int
MkPair [] [2,3] :: Pair [Int]
MkPair [] [] :: Pair [t]

data Tree t = Leaf | Node t (Tree t) (Tree t)
              deriving (Eq,Ord,Show)

treeFold:: (a -> b -> b -> b) -> b -> Tree a -> b
treeFold f e Leaf         = e
treeFold f e (Node x l r) = f x (treeFold f e l) (treeFold f e r)

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

In this case, functions can also be polymorphic:

```
equalPair :: Eq t => Pair t -> Bool
equalPair (MkPair x y) = (x == y)
```

Booleans and lists are implemented as algebraic data types. Furthermore, Haskell provides *Maybe* (for optional values) and *Either* (as a disjoint union type):

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)

data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show)
```

1.6.1 Proofs with Algebraic data types

Structural induction is used for proofs on algebraic data types. The rule for trees is for example (with the side condition that a, l, r is not free in Γ or P):

$$\frac{\Gamma \vdash P(Leaf) \qquad \Gamma, P(l), P(r) \vdash P(Node\ a\ l\ r)}{\Gamma \vdash \forall x \in Tree\ \ t.P(x)}$$

The general idea of structural induction is to use the structure of the terms. For example, for a data type:

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

We have to ask: What are the terms in step 0? The only possibility is $\{Leaf\ a | a \in t\}$ Then we have to ask: How de we go from step $i-1$ to step i. Here, we have:

$$\{\ \text{Node1}\ s | s \in T_{i-1}\} \cup \{\ \text{Node2}\ a\ s_1\ s_2 | a \in t \text{ and } s_1, s_2 \in T_{i-1}\}$$

Therefore, we have to proof the cases of step 0 ($\Gamma \vdash P(Leaf a)$) and the cases for step i, where our induction hypothesis is $P(x)$ for all $x \in T_{i-1}$. This leads to

$$\Gamma, P(s) \vdash P(Node1\ s) \text{ and } \Gamma, P(s_1), P(s_2) \vdash P(Node2\ a\ s_1\ s_2)$$

1.7 Lazy evaluation and efficiency

Haskell is lazy which means that expressions are evaluated only when necessary. Substitution occurs without argument evaluation. For a function $fxy = x + y$, we therefore have:

$$f(9-3)\ (f\ 34\ 3) = (9-3) + (f\ 34\ 3)$$

Some expressions may never be evaluated (which can save arbitrarily large amounts of time).

To prevent duplicated computation (e.g. in `square x = x * x`), both occurences are simultaneously reduced. The implementation is based on sharing where the terms are represented as directed graphs.

When pattern matching, arguments are evaluated as far as needed to determine the pattern match. This is illustrated in the following example:

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

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

For guards, the execution proceeds sequentially until a success. Local definitions (with `where`) are also evaluated lazily.

Functions are evaluated top-down (outermost operator first) and otherwise usually from left to right, depending on operator precedence.

Lazy evaluation means that for expressions like `sum (map (^4) [1 .. n])`, the intermediate lists are not fully constructed because the head is immediately turned into an addition. Furthermore, a function like `lmin = head . isort` executes in linear time (and would take quadratic time in a language like Java).

Lazy evaluation also enables finite representation of infinite data, e.g.:

```
ones = 1 : ones
from n = n : from (n+1)
```

Or for instance the sieve of Erastosthenes:

```
dropMults x ys = filter (\y -> y `mod` x /= 0) ys
sieve xs = head xs : sieve (dropMults (head xs) (tail xs))
primes = sieve [2 ..]
take 50 primes
```

One needs to keep in mind that induction is only sound for finite, everywhere defined data.

1.8 Monads

The idea if monads is to separate values from computations producing the values. An ordinary function, `f :: a -> b` returns the value of type *b*.

A monadic function `f :: a -> M b` returns the computation $M\,b$. M is a type constructor that satisfies certain properties (monad laws). By varying M, different notions of computation can be modelled. Or to state it less formally, a Mondad deals with the following question[1]: If we have a fancy value and a function that takes a normal value but returns a fancy value, how do we feed that fancy value into the function? Every monad supports two basic operations: Embedding a value into a computation (taking a normal value and putting it into context) and composing computations which is specified by the Monad typeclass:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b --bind
```

The type constructor `Maybe` instantiates this class like this:

```
instance Monad Maybe where
  return x       = Just x
  Nothing >>= _  = Nothing
  (Just x) >>= f = f x
```

The idea behind this instantiation is that when we compose functions and get `Nothing`, the next function shouldn't be called. When be get a value (`Just x`), the value should be passed to the next function. Now it's possible to wite a function like this:

```
foo2 xs ys =
  safeHead xs >>= (\a ->
  safeHead ys >>= (\b ->
  safeDiv a b >>= (\c ->
  return (c + 1))))

foo2' xs ys = do
  a <- safeHead xs
  b <- safeHead ys
  c <- safeDiv a b
  return (c + 1)
```

The *do*-notation is just syntactic sugar to improve readability. In a *do*-expression, every line is a monadic value. To inspect its results, we use `<-`.

A simple identity monad (that has no side effects) looks like this:

```
data Id a = Id a
instance Monad Id where
  return x      = Id x          -- identity function
  (Id m) >>= k = k m            -- function application
```

[1] Quote from Learn you a Haskell

As a shortcut for convenience `>>` (which is essentially the sequential composition `;` in imperative programming languages) is provided for when the second computation doesn't depend on the result of the first one:

```
(>>) :: m a -> m b -> m b
m1 >> m2 = m1 >>= (\_ -> m2)
```

And `fail` (not part of the mathematical concept of a monad) is provided for when the pattern matching fails in the *do*-notation.

Monads are mathematical objects with additional properties and must satisfy the following laws:

1. **Left unit:** `return x >>= f = f x`
2. **Right unit:** `m >>= return = m`
3. **Associativity:** `(m >>= f) >>= g = m >>= (\x -> (f x >>= g))`

The monad type class has two superclasses `Functor` and `Applicative` that must be instantiated as well. The `Functor` type class captures the functionality of transforming all structurs in a structure (like *map* does):

```
class Functor f where
fmap :: (a -> b) -> f a -> f b

instance Functor Tree where
fmap _ Leaf          = Leaf
fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Functor instances must satisfy two functor laws:

- **Identity:** `fmap id v = v`
- **Composition:** `fmap f (fmap g v) = fmap (f . g) v`

A canonical functor implementation for monads is `fmap f v = v >>= (return . f)`

In an applicative functor, `<*>` mimicks function application. E.g. if we want to apply `(<)` to `Just 2` and `Just 3`, we would use `fmap (<) (Just 2) <*> (Just 3)`.

```
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b


instance Applicative Maybe where
pure x = Just x

Just f <*> Just x = Just (f x)
_      <*> _      = Nothing
```

An example evaluation looks like this:

```
pure (+) <*> Just 5 <*> Just 7
= (pure (+) <*> Just 5) <*> Just 7
= (Just (+) <*> Just 5) <*> Just 7
= Just (5+)             <*> Just 7
= Just (5 + 7) = Just 12
```

The laws for applicative functors are:

- **Identity:** `pure id <*> v = v`
- **Homomorphism:** `pure f <*> pure x = pure (f x)`
- **Composition:** `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`
- **Interchange:** `v <*> pure x = pure (\f -> f x) <*> v`
- **fmap:** `fmap f v = pure f <*> v`

A canonical implementation for monads is:

```
pure x = return x
u <*> v = u >>= \f -> v >>= \x -> return (f x)
```

Monads are more expressive than applicative functors. With `<*>`, the effects of the second computation cannot depend on the result of the first one. But applicative functors are more general, for instance `Stream` is not a monad.

1.8.1 I/O

Haskell uses a monad to distinguish between pure expressions and expressions that interact with the world, namely $IO\,a$ for type of computations performing I/O operations and returning a value of type a. Haskell provides different IO primitives, e.g.:

- `getChar ::  IO Char`
- `putChar ::  Char -> IO ()`
- `return ::  a -> IO a`

It's not possible to "open" the IO type constructor. Because of this, any function doing I/O will have range $IO\,a$ for some type a. Some IO examples are:

```
putString :: String -> IO ()
putString ""     = return ()
putString (x:xs) = do putChar x; putString xs

getString :: IO String
    getString = do c <- getChar
                   if c == '\n'
```

```
8                       then return ""
9                       else do cs <- getString
10                              return (c:cs)
11
12 main :: IO ()
13 main = do putStrLn "Hi, I am HAL. Who are you?\n"
14           name <- getString
15           putStrLn ("Hello " ++ name ++ "!\n")
```

Chapter 2

Formal Methods

2.1 Introduction

Formal methods are 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. Mathematical notations is used to describe:

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

There are different types of requirements (more formally specified in later sections):

- **Safety properties**: Something bad will never happen (e.g. the functional behavior with no "incorrect" return values or the absence of certain faults like null-pointer exceptions or buffer overflows)
- **Liveness properties**: Something good will happen eventually (e.g. termination or that each request gets served eventually)
- **Non-functional requirements** like resource consumption (e.g. memory usage) or the runtime (e.g. realtime guarantees)

Formal logic is used to validate specifications, prove that the design satisfies requirements under given assumptions and prove that a more detailed design implements a more abstract one (refinement). The main proof methods are:

- **Deductive**: Proof system (for example termination is proved in a program logic)

- **Algorithmic**: State space exploration (model checking) (e.g. enumerate and check protocol runs)

The main ingredients for formal methods are:

- An underlying programming / modeling system. Either a programming language with precise (formal) semantics or a modeling language for constructing formal methods of software.
- A specification language where the desired properties are expressed as logical formulas in a formal logic. This gives a precise meaning to "the system satisfies a property".
- Proof method: A method to establish or refute that a system satisfies a property.
- Tool support for specification and verification (e.g. theorem provers and model checkers)

Benefits of formal methods are that it gives strong guarantees (detects faults with greater certainty than testing) and can guarantee the absence of specific faults. Furthermore, they are universal in the sense that they can be used for proofing properties of programs, for software designs (e.g. protocol verification), for programming languages (e.g. for a type safety proof) and for hardware (e.g. a refinement proof between gate and transistor design).

But formal methods have limitations. They do not per se guarantee correctness because the specifications can be wrong and it's difficult to get them right. Furthermore, almost all interesting properties are undecidable, in general and many tools quickly reach limits (in terms of scope and computing resources). Testing is still necessary to validate specifications, test properties not formally proven (e.g. performance) or detect errors in the environment (e.g. the compiler).

Formal semantics are useful in the design of programming languages (formal verification of properties, they can reveal ambiguities and support standardization), implementation of the language (specification for compilers / interpreters and support for portability) and while reasoning about programs (formal verification of program properties).

There are three kinds of programming language semantics:

- **Operational semantics**: Describes the execution of an abstract machine by describing how the effect is achieved. They are further divided in:
  - Natural Semantics (coarse-grained view of execution)
  - Structural Operational Semantics (fine-grained view of execution)

- **Denotational semantics** (not in the course): Programs are regarded as functions in a mathematical domain.
- **Axiomatic semantics**: Specific properties of the effect of executing a program are expressed, while some aspects of the computation may be ignored. There are two kinds of axiomatic semantics:
  - Partial correctness (properties without program termination)
  - Total correctness (termination is proven as an additional property)

2.2 The IMP Language

IMP has boolean and arithmetic expressions with no side-effects. All variables range over integers and are initialized. The language supports *if else* statements, *while* statements, variable assignments, the *skip* statement and sequential composition using ;.

In proofs, meta-variables are used to denote some program variable (where the concrete name is unspecified). $\equiv$ is written for syntactic equality, therefore $x \equiv y$ is always false for program variables but might be true for meta-variables (because they could both denote the same program variable).

Semantic functions map elements of syntactic categories to elements of semantic categories, e.g. from the syntactic category of the numerals to the semantic category Val=$\mathbb{Z}$. For the numerals, the following total (which can be proven by induction on n) function is used:

$$\begin{aligned}&\mathcal{N} : \text{Numeral} \to \text{Val} \quad \text{with} \quad \mathcal{N}[[0]] = 0 \quad \ldots \quad \mathcal{N}[[9]] = 9\\ &\mathcal{N}[[n\,0]] = \mathcal{N}[[n]] \times 10 + 0 \quad \ldots \quad \mathcal{N}[[n\,9]] = \mathcal{N}[[n]] \times 10 + 9\end{aligned}$$

A state is a total function, associating a value with each program variable: State: Var $\to$ Val. σ is used as a meta-variable for states. There is a (constant) state σ_{zero} in which all variables have the value 0:

$$\sigma_{zero}(x) = 0 \text{ for all } x$$

$\sigma[y \mapsto v]$ is the function that overrides the association of y in σ by $y \mapsto v$:

$$(\sigma[y \mapsto v])(x) = \begin{cases} v & \text{if } x \equiv y \\ \sigma(x) & \text{if } x \neq y \end{cases}$$

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

$$\sigma_1 = \sigma_2 \quad \Leftrightarrow \quad \forall x.\, (\sigma_1(x) = \sigma_2(x))$$

The semantic function $\mathcal{A}$: Aexp $\to$ State $\to$ Val maps an arithmetic expression e and a state σ to a value $\mathcal{A}[[\mathrm{e}]]\sigma$:

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

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

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

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

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

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

We can proof by structural induction on e that $\mathcal{A}$ is a total function (yields exactly one $v \in$ Val such that $\mathcal{A}[[e]]\sigma = v$ for every $e \in$ Aexp and $\sigma \in$ State):

- Case $e \equiv n$ for some numeral n: We have $\mathcal{A}[[n]]\sigma = \mathcal{N}[[n]]$ which yields exactly one value in Val because $\mathcal{N}$ is a total function.
- Case $e \equiv x$ for some variable x: $\mathcal{A}[[x]]\sigma = \sigma(x)$ and σ is a total function with $\sigma(x) \in$ Val.
- Case $e \equiv e_1$ op e_2 for some e_1, e_2 and $op \in$ Op: The I.H. gives the property for e_1 and e_2, therefore there is exactly one $v_1 \in$ Val such that $\mathcal{A}\,[[e_1]]\,\sigma = v_1$ (equivalent for e_2 with v_2). We have $\mathcal{A}\,[[e_1\ op\ e_2]]\,\sigma = v_1\,\overline{op}\,v_2$ and since $\overline{op}$ is a total function, the property follows.

The semantics of the language is therefore given by recursive definitions of functions $\mathcal{A}$ and $\mathcal{B}$. The values for composite elements are defined inductively in terms of the immediate elements. This suggests proofs by structural induction.

For substitution, we have the following rules:

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

There is the following lemma about the relationship between substitution / state update:

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

2.3 Operational Semantics

Operational semantics describe how the state is modified during the execution of a statement. Big-step semantics (also called Natural Semantics) describe how the overall results of the executions are obtained. Small-step semantics (also called Structural Operational Semantics) describe how the individual steps of the computations take place.

A transition system is a tuple $(\Gamma, T, \rightarrow)$ where Γ is a set of configurations, T a set of terminal configurations with $T \subseteq \Gamma$ and $\rightarrow$ is a transition relation with $\rightarrow \subseteq \Gamma \times \Gamma$. Operational semantics includes two types of configurations:

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

The transition relation $\rightarrow$ describes how executions take place.

Transition relation are specified by rules of the form (inference rules; optionally with side conditions): $\frac{\varphi_1 \ldots \varphi_n}{\psi}$ Such a rule means if $\varphi_1 \ldots \varphi_n$ are transitions (and side-condition is true), then ψ is a transition. $\varphi_1 \ldots \varphi_n$ are called the premises of the rule and ψ is called the conclusion of the rule. A rule without premises is sometimes called an axiom rule. The inference rule definitions are actually rule schemes with meta-variables as placeholders for statements, states, etc. A rule scheme describes infinitely many rule instances. A rule is instantiated when all meta-variables are replaced with syntactic elements.

2.3.1 Big-Step Semantics / Natural Semantics

Big-step transitions are of the form $\langle s, \sigma \rangle \rightarrow \sigma'$. The big-step semantics inference rules are listed in the appendix.

Rule instances can be combined to derive a transition $\langle s, \sigma \rangle \rightarrow \sigma'$. The result is a derivation tree T. The root of T ("bottom") is $\langle s, \sigma \rangle \rightarrow \sigma'$, i.e. $\text{root}(T) \equiv \langle s, \sigma \rangle \rightarrow \sigma'$. The leaves of T are axiom rule instances and the internal nodes are conclusions of rule instances and have the corresponding premises as immediate children. The side-conditions of all instantiated rules must be satisfied. The transition system permits a transition $\langle s, \sigma \rangle \rightarrow \sigma'$, written as $\vdash \langle s, \sigma \rangle \rightarrow \sigma'$, if and only if there exists a finite derivation tree ending in $\langle s, \sigma \rangle \rightarrow \sigma'$, i.e.:

$$\vdash \langle s, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \exists T.\, \text{root}(T) \equiv \langle s, \sigma \rangle \rightarrow \sigma'$$

The execution of a statement s in state σ terminates successfully iff there exists a state σ' such that $\vdash \langle s, \sigma \rangle \to \sigma'$. It fails to terminate iff there is no state σ' such that $\vdash \langle s, \sigma \rangle \to \sigma'$.

Two statements are semantically equivalent (written $s_1 \simeq s_2$) if:

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

To prove semantic equivalence, both directions have to be showed. One can assume that a derivation tree T exists that has as root the statement of the "left side of the implication". From this assumption, one has to show that the derivation tree for the other side of the implication exists. This is often done by case distinction on the last rule applied in T.

Structural induction over the structure of the statements doesn't work because the transition relation isn't defined inductively over the structure of the statements (for the WH_T rule, one gets another *while* statement that is not a proper sub-statement of s). Because of that, induction on the shape of derivation trees is used. To prove a property $P(T)$ for all derivation trees T, it is proven that $P(T)$ holds for an arbitrary derivation T under the assumption (I.H.) that $P(T')$ holds for all sub-trees T' of T ($T' \sqsubset T$). T' is called a sub-derivation of T if $T' \sqsubset T$. These types of proofs typically proceed by case distinction on the rule applied at the root of the arbitrary derivation tree T. This gives us information about the structure of the derivation and in particular about sub-derivations. To do proofs on the shape of derivation trees, the statements need to be rewritten. E.g.:

$$\begin{aligned} &\forall s, \sigma, \sigma', \sigma''. \vdash \langle s, \sigma \rangle \to \sigma' \wedge \vdash \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'' \\ &\forall s, \sigma, \sigma', \sigma''. \left(\exists T.\ \text{root}\ (T) \equiv \langle s, \sigma \rangle \to \sigma'\right) \wedge \vdash \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'' \\ &\forall T. \forall s, \sigma, \sigma', \sigma''. \left(\ \text{root}\ (T) \equiv \langle s, \sigma \rangle \to \sigma'\right) \wedge \vdash \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'' \end{aligned}$$

Where the $\vdash$ definition was used to get from the first to the second statement and logical equivalence to get from the second to the third.

Big-step semantics cannot distinguish between non-termination and abnormal termination and properties of non-terminating programs cannot be expressed. `abort` (an extension of IMP without additional rules) and `while true do skip end` are semantically equivalent. Furthermore, non-determinism suppresses non-termination, if possible. For instance there is a derivation tree for (and the statement is semantically equivalent to `(x:=2; x:=x+2)`):

$$\langle\ \texttt{while true do skip end}\ \square\ \texttt{(x:=2; x:=x+2)}, \sigma \rangle \to \sigma[\texttt{x} \mapsto 4]$$

Where $\square$ is a non-deterministic extension of IMP. Another disadvantage of big-step semantics is that parallelism (interleavings) cannot be modelled.

2.3.2 Small-Step Semantics / Structural Operational Semantics

Describing small steps of the execution allows one to express the order of execution of individual steps which can be used to express interleaving computations. Furthermore, properties of non-terminating programs can be expressed.

The configurations are the same as for natural semantics. But the transition relation $\rightarrow_1$ can have two forms:

- $\langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle$: The execution of s from σ is not completed and the remaining computation is expressed by the intermediate configuration $\langle s', \sigma' \rangle$
- $\langle s, \sigma \rangle \rightarrow_1 \sigma'$: The execution of s from σ has terminated and the final state is σ'

A transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$ describes the first step of the execution of s in state σ. A non-terminal configuration $\langle s, \sigma \rangle$ is stuck if there doesn't exist a configuration γ such that $\langle s, \sigma \rangle \rightarrow_1 \gamma$. The transition relation $\rightarrow_1$ is again defined using a derivation system (rules in the appendix). $\vdash \langle s, \sigma \rangle \rightarrow_1 \gamma$ means there exists a finite derivation tree ending in $\langle s, \sigma \rangle \rightarrow_1 \gamma$:

$$\vdash \langle s, \sigma \rangle \rightarrow_1 \gamma \Leftrightarrow \exists T.\text{root}(T) \equiv \langle s, \sigma \rangle \rightarrow_1 \gamma$$

k-step execution (with the intuitive meaning that there is an execution from γ to γ' in exactly k steps), written $\gamma \rightarrow_1^k \gamma'$ is defined like this:

1. $\gamma \rightarrow_1^0 \gamma'$ if and only if $\gamma = \gamma'$
2. For $k > 0$, $\gamma \rightarrow_1^k \gamma'$ if and only if there exists γ'' such that both $\vdash \gamma \rightarrow_1 \gamma''$ and $\gamma'' \rightarrow_1^{k-1} \gamma'$

It can be proven that $\gamma \rightarrow_1^{k_1+k_2} \gamma'$ if and only if $\exists \gamma''.\gamma \rightarrow_1^{k_1} \gamma'' \wedge \gamma'' \rightarrow_1^{k_2} \gamma'$ $\gamma \rightarrow_1^\star \gamma'$ is defined as $\exists k.\gamma \rightarrow_1^k \gamma'$, i.e. there is an execution from γ to γ' in some finite number of steps.

A derivation sequence is a sequence of configurations $\gamma_0, \gamma_1, \gamma_2, \ldots$ for which:

- $\gamma_i \rightarrow_1^1 \gamma_{i+1}$ for each $0 \leq i$ such that $i+1$ is in the range of the sequence.
- If the derivation sequence is finite then the last configuration in the sequence is either a terminal configuration or a stuck configuration.

Therefore, this is intuitively a derivation sequence which cannot be extended with further transactions. The length of a derivation sequence is the number of transitions $\gamma_i \rightarrow_1^1 \gamma_{i+1}$, a finite derivation sequence $\gamma_0, \gamma_1, \ldots, \gamma_k$ therefore has length k and a derivation sequence of length k corresponds to a k-step execution $\gamma_0 \rightarrow_1^k \gamma_k$ in which the final configuration γ_k is either stuck or terminal. In a derivation sequence, each individual step is justified by a

derivation tree. Because the small-step semantics of IMP is deterministic, there is exactly one derivation sequence starting in a configuration $\langle s, \sigma \rangle$.

The execution of a statement s in state σ terminates iff there is a finite derivation sequence starting with $\langle s, \sigma \rangle$ and runs forever iff there is an infinite derivation sequence starting with $\langle s, \sigma \rangle$. The execution of a statement s in state σ terminates successfully iff $\exists \sigma'. \langle s, \sigma \rangle \rightarrow_1^\star \sigma'$ (in IMP, an execution terminates successfully iff it terminates; there are no stuck configurations).

When reasoning about finite derivation sequences, we usually use strong induction on the length of a derivation sequence. More generally, we reason about a multi-step execution $\gamma \rightarrow_1^k \gamma'$ by strong induction on the number of steps k and define $P(k) \equiv$ "for all executions of length k, our property holds". $P(k)$ is proven for an arbitrary k with the induction hypothesis $\forall k' < k. P(k')$. Often times, the proof proceeds by dealing with the case of a 0-step execution specially (if applicable). For the other cases, the first execution step is "splitted off". For this first step, one can often get more information by considering the structure of the statement in the initial configuration or the derivation tree validating the first step of the execution.

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

- for all stuck or terminal configurations γ, we have $\langle s_1, \sigma \rangle \rightarrow_1^\star \gamma$ if and only if $\langle s_2, \sigma \rangle \rightarrow_1^\star \gamma$ and
- there is an infinite derivation sequence starting in $\langle s_1, \sigma \rangle$ if and only if there is one starting in $\langle s_2, \sigma \rangle$

2.3.3 Comparison

Natural Semantics	Structural Operational Semantics
• Local variable declarations and procedures can be modeled easily	• Local variable declarations (and procedures) require an explicit encoding of the original state
• No distinction between aborting and running forever	• Distinction between aborting and running forever
• Non-determinism suppresses non-termination (when possible)	• Non-determinism does not suppress non-termination
• Interleaving parallelism cannot be modelled	• Interleaving parallelism can be modelled

For every statement s of IMP, we have

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

Therefore, if the execution of s from some state terminates successfully in one of the semantics then it also does in the other and the resulting final states will be equal. The execution fails to terminate in the big step semantics if and only if it either gets stuck or runs forever in the small-step semantics.

2.4 Axiomatic Semantics

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, i.e. total correctness = partial correctness + termination. Axiomatic semantics are useful when we would like to focus on certain properties of states and considering how whole states are modified is too detailed / cumbersome to be practical.

In axiomatic semantics, properties of programs are specified as Hoare triples $\{\mathbf{P}\}s\{\mathbf{Q}\}$ where s is a statement and $\mathbf{P}$ and $\mathbf{Q}$ are assertions. The assertion $\mathbf{P}$ is called the precondition of a triple whereas the assertion $\mathbf{Q}$ is the postcondition of a triple. The informal meaning of $\{\mathbf{P}\}s\{\mathbf{Q}\}$ is that if $\mathbf{P}$ evaluates to true in an initial state σ and if the execution of s from σ terminates in a state σ', then $\mathbf{Q}$ will evaluate to true in σ'. Assertions are boolean expressions with some additional features. Because we often need to express that a variable in the final state has a certain relationship to a variable in the initial state, it is allowed that assertions contain logical variables. They may

only occur in assertions and are not program variables (and are therefore not allowed to be accessed by programs). They are used to "save" values, e.g.:

$$\{\mathrm{x} = N\}$$
$$\mathrm{y} := 1;\ \text{while not } \mathrm{x} = 1 \text{ do } \mathrm{y} := \mathrm{y} * \mathrm{x}; \mathrm{x} := \mathrm{x} - 1 \text{ end}$$
$$\{\mathrm{y} = N! \wedge N > 0\}$$

Axiomatic semantics are formalized by a derivation system where the premises and conclusions are Hoare triples. Derivation trees (rules in the appendix) are defined as for Operational semantics. We write $\vdash \{\mathbf{P}\}s\{\mathbf{Q}\}$ if and only if there exists a (finite) derivation tree ending in $\vdash \{\mathbf{P}\}s\{\mathbf{Q}\}$, i.e.:

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

Semantic entailment is defined like this: $\mathbf{P} \vDash \mathbf{Q}$ iff "for all states σ, $\mathcal{B}[[\mathbf{P}]]\sigma = tt$ implies $\mathcal{B}[[\mathbf{Q}]]\sigma = tt$". The rule of consequence allows semantic entailments in derivations. If $P \vDash P'$ and $Q' \vDash Q$, we have: $\dfrac{\{\mathbf{P}'\}s\{\mathbf{Q}'\}}{\{\mathbf{P}\}s\{\mathbf{Q}\}}$

We can therefore strengthen the preconditions ($\mathbf{P}$ cannot be weaker than $\mathbf{P}'$) and weaken the postconditions ($\mathbf{Q}$ cannot be stronger than $\mathbf{Q}'$).

It's important to note that all rules except the rule of consequence manipulate assertions syntactically. Without the rule of consequence, it is not possible to derive the triple $\{x = 4 \wedge y = 5\}$ skip $\{y = 5 \wedge x = 4\}$

Because derivation trees tend to get very larg and are inconvenient to write, an alternative is to group the assertions around the program text. Assertions are written before and after each statement to indicate which properties hold in the states before and after the execution of this statement. Therefore, we have for $SKIP_{Ax}$, ASS_{Ax} and SEQ_{Ax}:

$$\begin{array}{l}\{\mathbf{P}\}\\ \quad \texttt{skip}\\ \{\mathbf{P}\}\end{array} \qquad \begin{array}{r}\{\mathbf{P}[\mathtt{x} \mapsto \mathtt{e}]\}\\ \texttt{x := e}\\ \{\mathbf{P}\}\end{array} \qquad \begin{array}{r}\{\mathbf{P}\}\\ s_1;\\ \{\mathbf{Q}\}\\ s_2\\ \{\mathbf{R}\}\end{array}$$

Conditional statements, loops and the rule of consequence becomes:

$\{\mathbf{P}\}$
  `if` b `then`
    $\{b \wedge \mathbf{P}\}$
      s_1
    $\{\mathbf{Q}\}$
  `else`
    $\{\neg b \wedge \mathbf{P}\}$
      s_2
    $\{\mathbf{Q}\}$
  `end`
$\{\mathbf{Q}\}$

$\{\mathbf{P}\}$
  `while` b `do`
    $\{b \wedge \mathbf{P}\}$
      s
    $\{\mathbf{P}\}$
  `end`
$\{\neg b \wedge \mathbf{P}\}$

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

The entailment step is omitted when $\mathbf{P}$ and $\mathbf{P}'$ or $\mathbf{Q}$ and $\mathbf{Q}'$ are syntactically identical.

Proof outlines are often best developed bottom-up.

To prove properties of the axiomatic semantics, induction on the shape of the derivation tree is usually used (structural induction on IMP statements is often insufficient because of the rule of consequence). When doing case distinction, one needs to keep in mind that the last rule can also be the rule of consequence (even if there only is a statement like `skip`).

Two statements s_1 and s_2 are provably equivalent iff:

$$\forall \mathbf{P}, \mathbf{Q}. \vdash \{\mathbf{P}\}s_1\{\mathbf{Q}\} \quad \Leftrightarrow \quad \vdash \{\mathbf{P}\}s_2\{\mathbf{Q}\}$$

Soundness and completeness of the axiomatic semantics can be proven with respect to an operational semantics (here the big-step semantics): The partial correctness triple $\{\mathbf{P}\}s\{\mathbf{Q}\}$ is valid, written as $\models \{\mathbf{P}\}s_1\{\mathbf{Q}\}$ iff:

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

Soundness then means

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

and completeness

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

It can be proven that for all partial correctness triples $\{\mathbf{P}\}s\{\mathbf{Q}\}$ of IMP we have:

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

The weakest precondition (written as $wp(s, \mathbf{Q})$) of a statement s and a post-condition $\mathbf{Q}$ is the weakest predicate that has to hold in the initial state of an execution of s to guarantee that $\mathbf{Q}$ holds in the final state. The weakest pre-condition guarantees termination whereas the weakest liberal precondition (written as $wlp(s, \mathbf{Q})$) does not:

$$\mathcal{B}[[wp(s, \mathbf{Q})]]\sigma = tt \quad \Leftrightarrow \exists \sigma'. (\langle s, \sigma \rangle \rightarrow \sigma' \wedge \mathcal{B}[[\mathbf{Q}]]\sigma')$$
$$\mathcal{B}[[wlp(s, \mathbf{Q})]]\sigma = tt \quad \Leftrightarrow \forall \sigma'. (\langle s, \sigma \rangle \rightarrow \sigma' \Rightarrow \mathcal{B}[[\mathbf{Q}]]\sigma')$$

It can be proven that there exists a function wlp from pairs of statements and assertions to assertions, such that, for every statement s and predicate $\mathbf{Q}$, we have:

1. $\vDash \{wlp(s, \mathbf{Q})\}s\{\mathbf{Q}\}$
2. $\vDash \{\mathbf{P}\}s\{\mathbf{Q}\} \Rightarrow (\mathbf{P} \vDash wlp(s, \mathbf{Q}))$

2.4.1 Total Correctness

For total correctness, an alternative form of Hoare triple $\{\mathbf{P}\}s\{\Downarrow \mathbf{Q}\}$ is used. This has the informal meaning that if $\mathbf{P}$ evaluates to true in the inital state σ, then the execution of s from σ terminates and $\mathbf{Q}$ will evaluate to true in the final state. These triples and those of partial correctness form two separate axiomatic semantics. However, all total correctness derivation rules are analogous to those for partial correctness, except the rule for loops.

Termination is proved using loop variants. A loop variant is an expression that evaluates to a value in a well-founded set (for instance $\mathbb{N}$) before each iteration. Each loop iteration must decrease the value of the loop variant. The loop has to terminate when a minimal value of the well-founded set is reached. One has to proof that the value of the considered arithmetic expression e is non-negative before each loop iteration.

2.5 Modeling

Specifying and proofing some properties (like "all opened files must be closed eventually") is very cumbersome in structural operational semantics and impossible for non-terminating programs in natural and axiomatic semantics. The same holds for parallel programs, where proofs can only be done in structural operational semantics but all possible derivations have to be enumerated (exponential growth), so a manual enumeration is not feasible.

Therefore, the specification challenge is how to specify the properties of sequences of states concisely whereas the verification challenges are how to

prove properties of all possible program executions (concurrent systems) and / or how to automatically prove properties of infinite derivation sequences (reactive systems).

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model. Model checkers enumerate all possible state of a system. We differ between explicit state model checking (states are explicitly represented through concrete values; described in this chapter) and symbolic model checking (states are represented through boolean formulas).

Model checking works by feeding a property specification and a system model into a model checker. As a result, we get that the property is satisfied or that is violated with a concrete counterexample (or out of memory). In the modeling phase, the system has to be modeled under consideration using the description language of the model checker (possibly a programming language). The properties to be checked have to be formalized.

Model checking is mainly used to analyze system designs (as opposed to implementations). Typical properties to be analyzed include deadlocks, reachability of undesired states or protocol violations.

Systems are modelled as finite transition systems. There are communicating sequential processes (agents) and process execution is interleaved. Processes can communicate via shared variables, synchronous message passing and asynchronous message passing.

2.5.1 Promela

The protocol meta language Promela is the input language of the Spin model checker. Main objects are processes, channels and variables. Spin can ”execute” (simulate) the models.

Promela programs consist of:

```
1  // Constant declarations:
2  #define N 5
3  mtype = { ack, req };
4  // Structure declarations:
5  typedef vector { int x; int y };
6  // Global channel declarations:
7  chan buf = [2] of { int };
8  // Global variable declarations:
9  byte counter;
10 // Process declarations:
11 proctype myProc(int p) { ... }
12 // Variable declarations:
13 byte a, b = 5, c;
```

```
14 int d[3], e[4] = 3;
15 mtype msg = ack;
16 vector v;
```

The body of process declarations consists of a sequence of variable declarations, channel declarations and statements. With `active [N] proctype myProc(...) { ... }`, `N` instances of `myProc` are started in the initial system state. The `init` process is started initially, i.e. active in the initial system state. By using `active` prefixes, `init` can be omitted. But one can also use the `init` process exclusively to initialize other processes.

Promela has the following types:

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

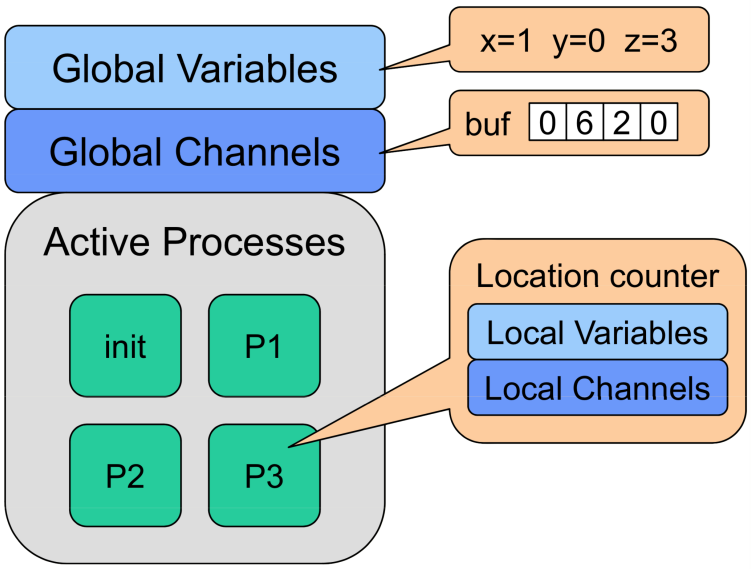
There are no floats or mathemtical (unbounded) integers. Furthermore, there are three user-defined types: Arrays (`int name[4]`), structures and the type of symbolic constants (`mtype`). And there's `chan` as channel type.

Variables are initialized to zero-equivalent values. If we declare channels like this:

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

- `c1` can store up to two messages and messages sent via `c1` consist of three parts (triples).
- `c2` models rendez-vous communication (no message buffer)
- `c3` is uninitialized; must be assigned an initialized channel before usage

Variable and channel declarations are local to a process or global. The state space of a promela system looks like this:



The number of states for sequential programs is

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

where $|\operatorname{dom}(x)|$ denotes the number of possible values of variable x. E.g. a sequential program with 10 locations and 3 boolean variables has 10×2^3 states. Therefore, the number of states grows exponentially in the number of variables (state space explosion). For concurrent programs (with N processes) we have (at most)

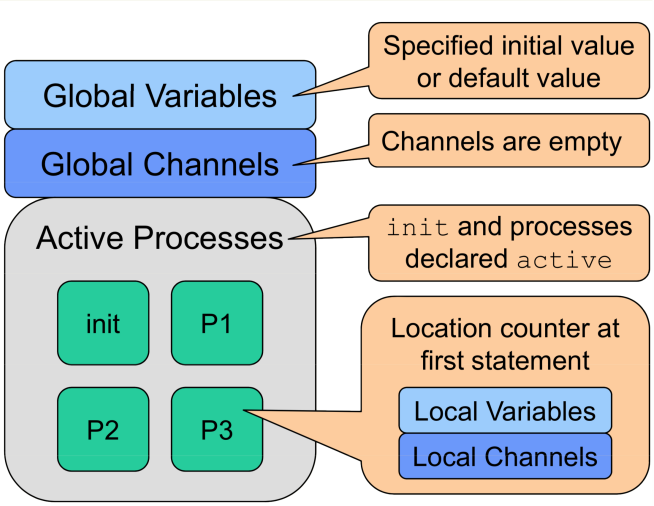
$$\prod_{i=1}^{N} \left(\#\text{ program locations }_i \times \prod_{\text{variable } x_i} |\operatorname{dom}(x_i)| \right)$$

For a system with N processes and K channels we have (at most)

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

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

The initial state looks like this:



A statement can be executable or blocked. Send is blocked if channel is full. A transition is made in three steps:

1. Determine all executable statements of all active processes (if no executable statement exists, the transition system gets stuck)
2. Choose non-deterministically one of the executable statements (Non-determinism models concurrency through interleaving)
3. Change the state according to the chosen statement

Promela expressions can contain variables, constants, literals, structure and array accesses. Unary and binary expressions can be formed with various operators (mostly identical to C). There are different functions that can be applied (like `len()`, `nempty()`, ...) and conditional expressions of the form `E1 -> E2 : E3`, which is the equivalent to C's `E1 ? E2 : E3`. For the statements we have:

- `skip`: Doesn't change state, always executable
- `timeout`: Doesn't change state, executable if all other statements in the system are blocked
- `assert(E)`: Aborts execution if expression `E` evaluates to zero; otherwise equivalent to `skip`. Always executable.
- `x = E`: Assigns the value of `E` to variable `x` whereas `a[n] = E` assigns the value of `E` to array element `a[n]`. Always executable.
- `s1;s2`: Sequential composition, is executable if `s1` is executable
- Expression statement (like `x > 0`) evaluates expression `E` where `E` must not change state (no side effects). It is executable if `E` evaluates to value different from zero.

- Selection:

```
1 if
2 :: s1   /* option 1 */
3 :: ...
4 :: sn   /* option n */
5 fi
```

  Executable if at least one of its options is executable. The statement `else` is executable if no other option is executable.

- Repetition:

```
1 do
2 :: s1   /* option 1 */
3 :: ...
4 :: sn   /* option n */
5 od
```

  Executable if at least one of its options is executable. Chooses repeatedly an option non-deterministically and executes it. Terminates when a `break` or `goto` is executed.

Basic statements are executed atomically, there is no interleaving during execution of the statements. `atomic { s }` executes `s` atomically. It is 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. A lock is therefore implemented like this:

```
1 atomic {
2   locked == 0;
3   locked = 1;
4 }
```

Promela does not contain procedures, but the effect can often be achieved by using macros, e.g.:

```
1 inline swap(a, b) {
2   int tmp;
3   tmp = a;
4   a = b;
5   b = tmp
6 }
```

They are simply inserted in the program (like C macros) and introduce no new scope.

`ch ! e1, ..., en` sends messages to the channel `ch`. Type of `ei` must correspond to `ti` in channel declaration and the send is executable iff buffer is not full. `ch ? a1, ..., an` receives a message where `ai` is a variable or

constant of type `ti`. The receive is executable iff buffer is not empty and the oldest message in the buffer machtes the constants `ai`. The variables `ai` are assigned values of the message.

For unbuffered channels (e.g. `chan ch = [0] of int ;`), `ch ! e1, ..., en` sends a message and is executable if there is a receive operation that can be executed simultaneously whereas `ch ? a1, ..., an` is executable if there is a send operation that can be executed simultaneously. Unbuffered channels model synchronous communication (rendez-vous).

2.5.2 Linear Temporal Logic

Linear-Time Properties

A slightly different transition system is used. A finite transition system is a tuple $(\Gamma, \sigma_l, \rightarrow)$ where:

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

The difference is that the initial configuration is fixed because transition systems model only one program / system, not all programs of a programming language. Furthermore, the terminal configuration is omitted which simplifies the theory (but termination can be modelled by a transition to a special sink state that allows only transitions back to itself).

The configurations are states (with global variables, global channels and per active process with local variables, local channels and a location counter). The initial configuration is the initial state and the transition relation is defined by operational semantics of statements. A promela model has a finite number of states because the number of active processes (255), number of variables / channels, ranges of variables and buffers of channels are finite.

S^ω is the set of infinite sequences of elements of set S, where $s_{[i]}$ denotes the i-th element of the sequence $s \in S^\omega$. $\gamma \in \Gamma^\omega$ is a computation of a transition system if:

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

γ is used to range over the states Γ of a transition system, i.e. $\gamma = \sigma_0\sigma_1\sigma_2\sigma_3\ldots$ and $\gamma_{[i]} = \sigma_i$.

$\mathcal{C}(TS)$ is the set of all computations of a transition system TS. Linear-time properties (LT-properties) can be used to specify the permitted computations of a transition system. A linear-time property P over Γ is a subset of Γ^ω, P specifies a particular set of infinite sequences of configurations.

TS satisfies LT-property P (over Γ):

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

So all computaions of TS have to belong to the set P.

In contrast, branching-time properties (not described) can also express the existence of a computation.

The property "All opened files must be closed eventually" is stated like this as a LT-property:

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

Because the explicit representation like this is not convenient, a set AP of atomic propositions is introduced. An atomic proposition is a proposition containing no logical connectives (i.e. no and, or, ...). For instance $AP = \{open, closed\}$ or $AP = \{x > 0, y \leq x\}$. A labeling $L : \Gamma \to \mathcal{P}(AP)$ function must be provided that maps configurations to sets of atomic propositions from AP. For example:

$$L(\sigma) = \begin{cases} \{ \text{ open } \} & \text{if } \sigma(o) = 1 \\ \{\text{closed}\} & \text{if } \sigma(o) = 0 \\ \{\} & \text{otherwise} \end{cases}$$

$L(\sigma)$ is called an abstract state.

A trace is an abstraction of a computation where we observe only the propositions of each state, not the concrete state itself. They are therefore infinite sequences of abstract states ($\mathcal{P}(AP)^{\omega}$). $t \in \mathcal{P}(AP)^{\omega}$ is a trace of a transition system TS if $t = L\left(\gamma_{[0]}\right) L\left(\gamma_{[1]}\right) L\left(\gamma_{[2]}\right), \ldots$ and γ is a computation of TS. $\mathcal{T}(TS)$ is the set of all traces of a transition system TS. LT-properties are typically specified over infinite sequences of abstract states (instead of configurations), e.g.:

$$P = \left\{ t \in \mathcal{P}(AP)^{\omega} | \forall i \geq 0 : \text{ open } \in t_{[i]} \Rightarrow \exists n > 0 : \text{ closed } \in t_{[i+n]} \right\}$$

Safety properties are properties of the type "something bad is never allowed to happen (and can't be fixed)". An LT-property P is a safety property if for all infinite sequences $t \in \mathcal{P}(AP)^{\omega}$: If $t \notin P$ then there is a finite prefix $\hat{t}$ of t such that for every infinite sequence t' with prefix $\hat{t}$, $t' \notin P$ $\hat{t}$ is called a bad prefix; this finite sequence of steps already violates the property (whatever happens afterwards). Safety properties are therefore violated in finite time and cannot be repaired. Examples are state properties (e.g. invariants like

the file is always either open or closed) or a property like ”money can be withdrawn after the correct pin has been entered”.

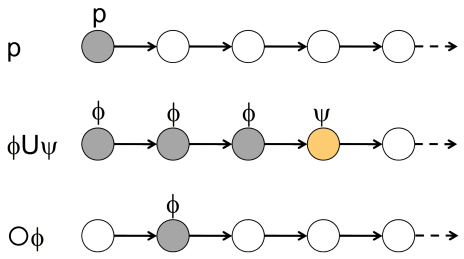
Liveness properties are properties of the type ”something good will happen eventually”. An LT-property P is a liveness property if every finite sequence $\hat{t} \in \mathcal{P}(AP)^*$ is a prefix of an infinite sequence $t \in P$. A liveness property does not rule out any prefix because every finite prefix can be extended to an infinite sequence that is in P. Liveness properties are therefore violated in infite time. Examples are ”all opened files must be closed eventually” or ”the program terminates eventually”.

Linear Temporal Logic

Linear Temporal Logic (LTL) allows us to formalize LT-properties of traces in a convenient way. It is decidable whether or not the traces of a finite transition system satisfy an LTL. The basic operators of LTL are:

$$\varphi = p | \neg\varphi | \varphi \wedge \varphi | \varphi \cup \varphi | \bigcirc \varphi$$

Where p is a proposition from a chosen set of atomic propositions AP. Intuitively, they mean:



Or more formally:

$t \vDash p$ iff $p \in t_{[0]}$
$t \vDash \neg\varphi$ iff not $t \vDash \varphi$
$t \vDash \varphi \wedge \psi$ iff $t \vDash \varphi$ and $t \vDash \psi$
$t \vDash \varphi \cup \psi$ iff there is a $k \geq 0$ with $t_{(\geq k)} \vDash \psi$ and $t_{(\geq j)} \vDash \varphi$ for all j such that $0 \leq j < k$
$t \vDash \bigcirc\varphi$ iff $t_{(\geq 1)} \vDash \varphi$

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

There are the following derived operators:

- Eventually: $\diamond\varphi \equiv (\text{true} \cup \varphi)$
- Always (from now): $\Box\varphi \equiv \neg \diamond \neg\varphi$

Unary operators always have highest precedence, i.e. $\diamond\varphi \Rightarrow \psi$ means $(\diamond\varphi) \Rightarrow \psi$

Some useful specification patterns are:

- Strong invariant: $\Box\psi$ (ψ always holds which is a safety property)
- Monotone invariant: $\Box(\psi \Rightarrow \Box\psi)$ (once ψ is true, it is always true, which is a safety property)
- Establishing an invariant: $\diamond\Box\psi$ (eventually ψ will always hold, which is a liveness property)
- Responsiveness: $\Box(\psi \Rightarrow \diamond\varphi)$ (every time that ψ holds, φ will eventually hold, which is a liveness property)
- Fairness: $\Box \diamond \psi$ (ψ holds infinitely often, which is a liveness property)

Model checking

The model checking problem is, given a finite transition system TS and an LTL formula φ, decide whether $t \vDash \varphi$ for all $t \in \mathcal{T}(TS)$ I.e. $\mathcal{T}(TS) \subseteq P(\varphi)$ needs to be checked. Naively searching all traces is not an option because they have infinite length.

For safety properties, any violation can be observed after a finite prefix. All finite prefixes of the traces are characterized using a (nondeterministic) finite automaton characterizing all finite prefixes $\mathcal{T}_{\text{fin}}(TS)$ of the traces of TS. We therefore have an automaton $\mathcal{FA}_{TS} = (Q, \Sigma, \delta, Q_0, F)$ with:

- $Q = \Gamma \cup \{\sigma_0\}$, where $\sigma_0 \notin \Gamma$
- $\Sigma = \mathcal{P}(AP)$
- $\delta = \{(\sigma, p, \sigma') \,|\sigma \to \sigma' \text{ and } p = L(\sigma')\} \cup \{(\sigma_0, p, \sigma_l) \,|p = L(\sigma_l)\}$ (transitions labeled by abstract states, $\delta \subseteq Q \times \Sigma \times Q$)
- $Q_0 = \{\sigma_0\}$
- $F = Q$ (any prefix is accepted)

A safety property is regular if its bad prefixes are described by a regular language over the alphabet $\mathcal{P}(AP)$. Every invariant over AP is a regular safety property because for the property defined by $\Box p$, all bad prefixes start with S^*T, where S describes any subset of $\mathcal{P}(AP)$ that contains p and T any subset that doesn't contain p. These bad prefixes are described by a finite automaton $\mathcal{FA}_{\overline{p}}$. Then the finite automaton for the product of $\mathcal{FA}_{TS}$ and $\mathcal{FA}_{\overline{p}}$ is constructed. Finally, one checks if the resulting automaton has any reachable accepting states. If not, the property P is never violated in traces of TS and if yes, the property is violated (and one gets a conuterexample, namely each word in the accepted language).

The product of the two NFA is constructed like this: $\mathcal{FA}_{TS\cap\overline{P}} = \mathcal{FA}_{TS} \times \mathcal{FA}_{\overline{P}} = (Q, \Sigma, \delta, Q_0, F)$ where:

- Q is the cartesian product of the states of $\mathcal{FA}_{TS}$ and $\mathcal{FA}_{\overline{P}}$
- $\Sigma = \mathcal{P}(AP)$
- δ is defined by $((\sigma_1, A_1), p, (\sigma_2, A_2)) \in \delta$ iff (σ_1, p, σ_2) is in the transition relation of $\mathcal{FA}_{TS}$ and (A_1, p, A_2) is in the transition relation of $\mathcal{FA}_{\overline{P}}$
- Q_0 is the set of states (σ_0, A_0) where σ_0 is an initial state of $\mathcal{FA}_{TS}$ and A_0 of $\mathcal{FA}_{\overline{P}}$
- F is the set of state (σ, A_F) where A_F is an accepting state of $\mathcal{FA}_{\overline{P}}$

ω-regular expressions denote languages of infinite words and have the form:

$$G = E_1 F_1^\omega + \ldots + E_n F_n^\omega \quad (1 \leq n)$$

Where E_i and F_i are regular expressions and $\mathcal{L}(F^\omega) = \{w_1 w_2 w_3 \ldots | \forall i \cdot w_i \in \mathcal{L}(F)\}$ The class of languages accepted by non-deterministic Büchi automata agrees with the class of ω-regular languages. Traces $\mathcal{T}(TS)$ and the set of traces satisfying an LTL formula can be described as ω-regular expression. Therefore, non regular safety properties and liveness properties can be checked very similar to regular safety properties, but with Non-determenistic Büchi automata.

For a finite transition system TS and an LTL formula φ, the model checking problem $TS \vDash \varphi$ is solvable in $\mathcal{O}\left(|TS| \times 2^{|\varphi|}\right)$ where $|TS|$ is the size of the transition system (which grows exponentially) and $|\varphi|$ is the size of φ.

There are more advanced model checking techniques like:

- On-the-fly model checking: Often violation can be detected without checking all possible states, so generate the transition system and check property step-by-step
- Partial order reduction: Remove redundancy from different interleavings of concurrent executions (because code segments that operate only on local state are not affected by interleaving)
- Bounded model checking: Check only prefixes of traces up to a certain length (closer to testing, but very effective in practice)
- Symbolic model checking: Uses sets of states (that are represented through boolean functions) rather than individual states

Chapter 3

Appendix

We recall the following rules from natural deduction for first order intuitionistic logic:

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

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot E \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg I \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B} \neg E$$

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

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

$$\frac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x.\, A(x)} \forall I^{*} \qquad \frac{\Gamma \vdash \forall x.\, A(x)}{\Gamma \vdash A(t)} \forall E$$

$$\frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x.\, A(x)} \exists I \qquad \frac{\Gamma \vdash \exists x.\, A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B} \exists E^{**}$$

Side conditions: (*) x does not occur free in any formula in Γ and
(**) x does not occur free in any formula in Γ or B.

Recall the proof rules for the Mini-Haskell type system:

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \mathit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \to \tau}\ \mathit{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \mathit{App}$$

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

$$\frac{\Gamma \vdash t :: \mathit{Int}}{\Gamma \vdash \textbf{iszero}\ \ t :: \mathit{Bool}}\ \mathit{iszero} \qquad \frac{\Gamma \vdash t_1 :: \mathit{Int} \qquad \Gamma \vdash t_2 :: \mathit{Int}}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: \mathit{Int}}\ \mathit{BinOp} \text{ for } \textbf{op} \in \{+, *\}$$

$$\frac{\Gamma \vdash t_0 :: \mathit{Bool} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 :: \tau}\ \mathit{if}$$

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

Syntax and Semantics of IMP

Syntax

The statements (Stm) of the programming language **IMP** are given by the grammar

$$s ::= \texttt{skip} \mid x \texttt{ := } e \mid s; s \mid \texttt{if } b \texttt{ then } s \texttt{ else } s \texttt{ end} \mid \texttt{while } b \texttt{ do } s \texttt{ end}$$

where x ranges over variables (Var), e ranges over arithmetic expressions (Aexp), and b ranges over boolean expressions (Bexp).

Natural Semantics (Big-Step Semantics)

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

$$\frac{\langle \underline{s_1}, \underline{\sigma} \rangle \to \underline{\sigma'} \quad \langle \underline{s_2}, \underline{\sigma'} \rangle \to \underline{\sigma''}}{\langle \underline{s_1}\texttt{;}\underline{s_2}, \underline{\sigma} \rangle \to \underline{\sigma''}} \ (\textsc{Seq}_{NS})$$

$$\frac{\langle \underline{s_1}, \underline{\sigma} \rangle \to \underline{\sigma'}}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s_1} \texttt{ else } \underline{s_2} \texttt{ end}, \underline{\sigma} \rangle \to \underline{\sigma'}} \ (\textsc{IfT}_{NS}) \quad \text{if } \mathcal{B}[\![\underline{b}]\!]\underline{\sigma} = \mathit{tt}$$

$$\frac{\langle \underline{s_2}, \underline{\sigma} \rangle \to \underline{\sigma'}}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s_1} \texttt{ else } \underline{s_2} \texttt{ end}, \underline{\sigma} \rangle \to \underline{\sigma'}} \ (\textsc{IfF}_{NS}) \quad \text{if } \mathcal{B}[\![\underline{b}]\!]\underline{\sigma} = \mathit{ff}$$

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

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

Structural Operational Semantics (Small-Step Semantics)

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

$$\frac{\langle \underline{s_1}, \underline{\sigma}\rangle \to_1 \underline{\sigma'}}{\langle \underline{s_1}\texttt{;}\underline{s_2}, \underline{\sigma}\rangle \to_1 \langle \underline{s_2}, \underline{\sigma'}\rangle}\ (\textsc{Seq1}_{SOS}) \qquad \frac{\langle \underline{s_1}, \underline{\sigma}\rangle \to_1 \langle \underline{s_1'}, \underline{\sigma'}\rangle}{\langle \underline{s_1}\texttt{;}\underline{s_2}, \sigma\rangle \to_1 \langle \underline{s_1'}\texttt{;}\underline{s_2}, \underline{\sigma'}\rangle}\ (\textsc{Seq2}_{SOS})$$

$$\frac{}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s_1} \texttt{ else } \underline{s_2} \texttt{ end}, \underline{\sigma}\rangle \to_1 \langle \underline{s_1}, \underline{\sigma}\rangle}\ (\textsc{IfT}_{SOS}) \quad \text{if } \mathcal{B}[\![\underline{b}]\!]\underline{\sigma} = tt$$

$$\frac{}{\langle \texttt{if } \underline{b} \texttt{ then } \underline{s_1} \texttt{ else } \underline{s_2} \texttt{ end}, \underline{\sigma}\rangle \to_1 \langle \underline{s_2}, \underline{\sigma}\rangle}\ (\textsc{IfF}_{SOS}) \quad \text{if } \mathcal{B}[\![\underline{b}]\!]\underline{\sigma} = ff$$

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

Axiomatic Semantics (partial correctness)

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

$$\frac{\{\ \underline{\mathbf{P}}\ \}\ \underline{s_1}\ \{\ \underline{\mathbf{Q}}\ \} \quad \{\ \underline{\mathbf{Q}}\ \}\ \underline{s_2}\ \{\ \underline{\mathbf{R}}\ \}}{\{\ \underline{\mathbf{P}}\ \}\ \underline{s_1}\texttt{;}\underline{s_2}\ \{\ \underline{\mathbf{R}}\ \}}\ (\textsc{Seq}_{Ax})$$

$$\frac{\{\ \underline{b} \wedge \underline{\mathbf{P}}\ \}\ \underline{s_1}\ \{\ \underline{\mathbf{Q}}\ \} \quad \{\ \neg\underline{b} \wedge \underline{\mathbf{P}}\ \}\ \underline{s_2}\ \{\ \underline{\mathbf{Q}}\ \}}{\{\ \underline{\mathbf{P}}\ \}\ \texttt{if } \underline{b} \texttt{ then } \underline{s_1} \texttt{ else } \underline{s_2} \texttt{ end}\ \{\ \underline{\mathbf{Q}}\ \}}\ (\textsc{If}_{Ax})$$

$$\frac{\{\ \underline{b} \wedge \underline{\mathbf{P}}\ \}\ \underline{s}\ \{\ \underline{\mathbf{P}}\ \}}{\{\ \underline{\mathbf{P}}\ \}\ \texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end}\ \{\ \neg\underline{b} \wedge \underline{\mathbf{P}}\ \}}\ (\textsc{Wh}_{Ax})$$

$$\frac{\{\ \underline{\mathbf{P}'}\ \}\ \underline{s}\ \{\ \underline{\mathbf{Q}'}\ \}}{\{\ \underline{\mathbf{P}}\ \}\ \underline{s}\ \{\ \underline{\mathbf{Q}}\ \}}\ (\textsc{Cons}_{Ax}) \quad \text{if } \underline{\mathbf{P}} \vDash \underline{\mathbf{P}'} \text{ and } \underline{\mathbf{Q}'} \vDash \underline{\mathbf{Q}}$$

Axiomatic Semantics (total correctness)

As for partial correctness above, except for the following rule, which replaces $(\textsc{Wh}_{Ax})$:

$$\frac{\{\ \underline{b} \wedge \underline{\mathbf{P}} \wedge \underline{e} = \underline{Z}\ \}\ \underline{s}\ \{\ \Downarrow \underline{\mathbf{P}} \wedge \underline{e} < \underline{Z}\ \}}{\{\ \underline{\mathbf{P}}\ \}\ \texttt{while } \underline{b} \texttt{ do } \underline{s} \texttt{ end}\ \{\ \Downarrow \neg\underline{b} \wedge \underline{\mathbf{P}}\ \}}\ (\textsc{WhTot}_{Ax}) \quad \text{if } \underline{b} \wedge \underline{\mathbf{P}} \ \vDash\ 0 \leq \underline{e}$$