FMFP - Complete Summary

Ruben Schenk, ruben.schenk@inf.ethz.ch

April 1, 2022

1 Introduction & Basic Haskell Syntax

1.1 Example: GCD

The **GCD problem** is given as follows: Compute the greatest common divisor of two natural numbers. We have the following *specifications:* Let $x, y \in \mathcal{N}$ be given. The number z is the **greatest common divisor** of x and y iff. $z \mid x$ and $z \mid y$ and there is no z', with $z' > z$, such that $z' \mid x$ and $z' \mid y$. Here, $z \mid x \equiv \exists a \in \mathcal{N}. a \cdot z = x$.

The problem specification is not **constructive,** i.e. it does not describe how the GCD should be computed.

1.1.1 Imperative GCD

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

The **imperative GCD**, as shown above, consists of control flow statements and assignments. Assignments change the computer's *state.* To understand `gcd`, one must understand how its state changes.

Poor man's reasoning would be to simulate and track the memory content during execution. A better way would be to use *Hoare logic* in the form of $\{P\}$ prog $\{Q\}$. Formal reasoning is possible, but not easy!

1.1.2 Functional GCD

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

The functional way formalizes *what* should be computed, rather than *how.* This is an algorithm, provided we have also specified how functions are executed.

1.2 Basic Concepts in Functional Programming

1.2.1 Referential Transparency

Functions compute values. But functions also *are* values: we can compute and return them. It is important to note that functions in functional programming have **no side effects:** `f(x)` always returns the same value. This in contrast to other programming languages we've known so far. Consider the following `Java` example:

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

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

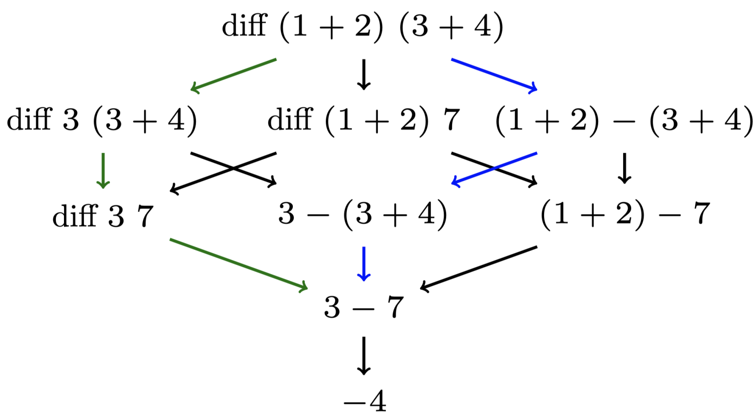
One will immediately see that this prints out `0` and then `1`, which means that `f(0)` returns different values with the same input.

Since functions have no side effects, we can reason with the more easily in mathematics. This property is also called **referential transparency:** an expression evaluates to the same value in every context.

1.2.2 Evaluation

An **evaluation strategy** defines how and when expressions are evaluated during the execution of a program. We differ between two strategies:

- *Eager evaluation:* evaluate arguments first. Also called "call-by-value", corresponds to the left (green) path in the figure below.
- *Lazy evaluation:* evaluate arguments only when needed (used by Haskell). Also called "call-by-need" (or "left-most/outermost"), corresponds to the right (blue) path in the figure below.



1.3 Basic Haskell Syntax

1.3.1 Syntax and Types

We present the basic syntax principles in the following code example:

```
gcd x y        -- functions and arguments start with lower-case letters
    | x == y    = x
    | x > y     = gcd (x - y) y          -- arguments are written in sequence and
    | otherwise = gcd x       (y - x)    -- separated by whitespace
```

Furthermore, functions consist of different cases and a program consists of several definitions:

```
myConstant = 5

afunction y1 y2 ... ym
    | guard1 = expr1
    | guard2 = expr2
    ...
    | guardm = exprm

anotherFucntion z1 z2 ... zk = ...
```

Indentation determines the separation of definitions. All function definitions must start at the same indentation level. If a definition requires $n > 1$ lines, we indent lines 2 to n further. This leads to the following *recommended layout:*

```
f1 x1 x2
    | a long guard which may go over
      a number of lines
        = a long expression that can also go over
          several lines
    | g2 = expr2

f2 x1 x2 x3 = ...
```

1.3.2 Functions

Functions live in a global scope. This means that a function can be called from any other. Example:

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

We can define functions and variables in local scope with `let` and `where`:

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

2 Natural Deduction

2.1 Introduction to Natural Deduction

2.1.1 Abstract Example (without Assumptions)

Consider the following "meaningless" language:

$$\mathcal{L} = \{\oplus, \otimes, \times, +\}$$

We furthermore state the following *rules:*

- α: If $+$, then $\otimes$
- β: If $+$, then $\times$
- γ: If $\otimes$ and $\times$, then $\oplus$
- δ: $+$ holds

Our goal is to prove $\oplus$. We might proceed as follows:

1. $+$ holds by γ.
2. $\otimes$ holds by α with 1.
3. $\times$ holds by β with 1.
4. $\oplus$ holds by γ with 2 and 3.

We might also present this proof as a **derivation tree:**

$$\cfrac{\cfrac{\cfrac{}{+}\,\delta}{\otimes}\,\alpha \qquad \cfrac{\cfrac{}{+}\,\delta}{\times}\,\beta}{\oplus}\,\gamma$$

2.1.2 Abstract Example (with Assumptions)

We revisit the previous example by slightly changing one of our rules:

- α: If $+$, then $\otimes$
- β: If $+$, then $\times$
- γ: If $\otimes$ and $\times$, then $\oplus$
- δ: We may assume $+$ when proving $\oplus$

We can build the following proof system. In this system, Γ is the set of assumptions we make during our proof:

$$\cfrac{}{\ldots, A, \ldots \vdash A}\,\textit{axiom}$$

$$\cfrac{\Gamma \vdash +}{\Gamma \vdash \otimes}\,\alpha \qquad \cfrac{\Gamma \vdash +}{\Gamma \vdash \times}\,\beta$$

$$\cfrac{\Gamma \vdash \otimes \quad \Gamma \vdash \times}{\Gamma \vdash \oplus}\,\gamma \qquad \cfrac{\Gamma, + \vdash \oplus}{\Gamma \vdash \oplus}\,\delta$$

Our derivation tree from previously changes slightly to the following:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{+ \vdash +}\,\textit{axiom}}{+ \vdash \otimes}\,\alpha \qquad \cfrac{\cfrac{}{+ \vdash +}\,\textit{axiom}}{+ \vdash \times}\,\beta}{+ \vdash \oplus}\,\gamma}{\vdash \oplus}\,\delta$$

2.1.3 Summary

Rules are used to construct derivations under assumptions. $A_1, ..., A_n \vdash A$ reads as "A follows from $A_1, ..., A_n$".

Derivations are trees as shown in the examples above.

A **proof** is a derivation whose root has no assumptions.

2.2 Propositional Logic

2.2.1 Syntax

Propositions are built from a collection of variables and closed under disjunction, conjunction, implication, etc. 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 $X \in \mathcal{V}$
- $\perp \in \mathcal{L}_P$
- $A \land B \in \mathcal{L}_P$ if $A \in \mathcal{L}_P$ and $B \in \mathcal{L}_P$
- $A \lor B \in \mathcal{L}_P$ 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 $B \in \mathcal{L}_P$

In the following: X ranges over variables, A and B over formulae.

2.2.2 Semantics

A **valuation** $\sigma : \mathcal{V} \to \{\text{True, False}\}$ is a function mapping variables to truth values. Valuations are simple kinds of models (or interpretations). We denote the set of valuations as Valuations.

Satisfiability is the smallest relation $\models \subseteq \text{Valuations} \times \mathcal{L}_P$ such that:

- $\sigma \models X$ if $\sigma(X) = \text{True}$
- $\sigma \models A \land B$ if $\sigma \models A$ and $\sigma \models B$
- $\sigma \models A \lor B$ if $\sigma \models A$ or $\sigma \models B$
- $\sigma \models A \to B$ if whenever $\sigma \models A$ then $\sigma \models B$

Note that $\sigma \not\models \perp$ for every $\sigma \in \text{Valuations}$.

We furthermore introduce the following characteristics about propositional logic:

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

Examples:

- $X \land Y$ is satisfiable as $\sigma \models X \land Y$ for $\sigma(X) = \sigma(Y) = \text{True}$
- $X \to X$ is valid
- $\neg X$, $X \lor Y \models Y$ holds as $\sigma \models \neg X$ and $\sigma \models X \lor Y$ constraint σ to $\sigma(X) = \text{False}$ and $\sigma(Y) = \text{True}$, so $\sigma \models Y$

2.2.3 Requirements

We need some **requirements** for *deductive systems.* The main requirement is that syntactic entailment $\vdash$ (derivation rules) and semantic entailment *vDash* (truth tables) should agree. This requirement has two parts:

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

Here, $\Gamma \equiv A_1, ..., A_n$ is some collection of formulae.

2.2.4 Natural Deduction for Propositional Logic

A **sequent** is an assertion (judgement) of the form $A_1, ..., A_n \vdash A$, where all $A, A_1, ..., A_n$ are propositional formulae. A **proof** of A is a derivation tree with root $\vdash A$. If the deductive system is sound, then A is a tautology.

Conjunction **Conjunction** proposes rules of two kinds: *introduce* and *eliminate* connectives. The rules are given as follows:

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

Example: The following figure shows an example derivation using conjunction rules.

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

Implication The rules for **implication** are given as follows:

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

Disjunction The rules for **disjunction** are given as follows:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-}IL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-}IR$$

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

2.3 First-Order Logic

2.3.1 Syntax

In **first-order logic** we have two syntactic categories: **terms** and **formulae.**

A **signature** consists of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$. We write f^k (or p^k) to indicate function symbol f (or predicate symbol p) has arity $k \in \mathcal{N}$. Constants are 0-ary function symbols.

Now, let $\mathcal{V}$ be a set of variables. Then:

Definition: $Term$, the **terms of first-order logic,** is the smallest set where:

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

Definition: $Form$, the **formulae of first-order logic,** is the smallest set where:

1. $\perp \in Form$,
2. $p^n(t_1, ..., t_n) \in Form$ if $p^n \in \mathcal{P}$ and $t_j \in Term$, for all $1 \leq j \leq n$,
3. $A \circ B \in Form$ if $A \in Form$, $B \in Form$, and $\circ \in \{\land, \lor, \rightarrow\}$, and
4. $Qx.A \in Form$ if $A \in Form$, $x \in \mathcal{V}$, and $Q \in \{\forall, \exists\}$.

Each occurrence of each variable in a formula is either **bound** or **free.** A variable occurrence x in a formula A is **bound** if x occurs within a subformula B of A of the form $\exists x.B$ or $\forall x.B$.

2.3.2 Binding and α-conversion

Names of bound variables are irrelevant, they just encode the binding structure. We can rename *bound* variables, this process is called α**-conversion**.

It is important to note that the renaming must *preserve the binding structure!*

Some notes on bindings and parentheses:

- $\land$ binds stronger than $\lor$, and $\lor$ binds stronger than $\rightarrow$.
- $\rightarrow$ associates to the right, *land* and *lor* to the left.
- Negation binds stronger than binary operators.
- Quantifiers extend to the right as far as possible: to the end of the line or ')'

$$\left(p \lor \left(q \land (\neg r)\right)\right) \rightarrow (p \lor q)$$

$$p \rightarrow \left((q \lor p) \rightarrow r\right)$$

$$p \land \left(\forall x. \left(q(x) \lor r\right)\right)$$

$$\neg\left(\forall x. \left(p(x) \land \left(\forall x. \left(\left(q(x) \land r(x)\right) \land s\right)\right)\right)\right)$$

2.3.3 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:

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

As a shorthand, we write $p^\mathcal{S}$ for $I_\mathcal{S}(p)$ and $f^\mathcal{S}$ for $I_\mathcal{S}(f)$.

An **interpretation** is a pair $\mathcal{I} = \langle \mathcal{S}, v \rangle$, where $\mathcal{S} = \langle U_\mathcal{S}, I_\mathcal{S}$ is a structure and $v : \mathcal{V} \to U_\mathcal{S}$ is a valuation.

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

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

Satisfiability is the smallest relation $\vDash \subseteq Interpretations \times Form$ satisfying:

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

Here, $v[x \to a]$ is the valuation v' 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 $langle \mathcal{S}, v \rangle$ is a **model** of A. Note that if A does not have free variables, satisfaction does not depend on the valuation v. We write $\mathcal{S} \vDash A$. When every interpretation is a model, we write $\vDash A$ and say that A is **valid.**

A is **satisfiable** if there is at least one model for A (and said to be **contradictory** otherwise).

Example: Consider the following examples:

- $\forall x.\exists y.y * 2 = x$ satisfied w.r.t. rationals.
- $\forall x.\forall y.x < y \to \exists z.x < z \land z < y$ satisfied w.r.t. any dense order.
- $\exists x.x \neq 0$ satisfied w.r.t. structures $\mathcal{S}$ with ≥ 2 elements in $U_\mathcal{S}$.
- $(\forall x.p(x, x)) \to p(a, a)$ is valid.

2.3.4 Substitution

Substitution describes the process of replacing in A all occurrences of a free variable x with some term t. We write $A[x \to t]$ to indicate the substitution.

Example:

$$
\begin{aligned}
A &\equiv \exists y.y * x = x * z \\
A[x \to 2 - 1] &\equiv \exists y.y * (2 - 1) = (2 - 1) * z \\
A[x \to z] &\equiv \exists y.y * z = z * z
\end{aligned}
$$

All free variables of t must still be free in $A[x \to t]$. Avoid *capture!* If necessary, α-convert A before substitution.

2.3.5 Universal Quantification

The rules are as follows:

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

The side condition $*$ is: x must not be free in any assumption in Γ.

2.3.6 Existential Quantification

The rules are as follows:

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

The side condition $*$ is: x is neither free in B nor free in Γ.

2.4 Equality

Equality is a logical symbol with associated proof rules. One speaks of *first-order logic with equality* rather than equality just being another predicate:

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

Equality is an *equivalence* relation with the following rules:

$$\frac{}{\Gamma \vdash t = t}\ \mathit{ref} \qquad \frac{\Gamma \vdash t = s}{\Gamma \vdash s = t}\ \mathit{sym} \qquad \frac{\Gamma \vdash t = s \qquad \Gamma \vdash s = r}{\Gamma \vdash t = r}\ \mathit{trans}$$

And equality is also a *congruence* on terms and all definable relations:

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

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

2.5 Correctness

Correctness is important! But what does correctness mean? What properties should hold?

- *Termination:* Important for many, but not all, programs.
- *Functional behavior:* Function should return "correct" value.

2.5.1 Termination

If f is defined in terms of functions $g_1, ..., g_k$ ($g_i \neq f$), and each g_i terminates, then so does f. The problem we encounter here is *recursion,* i.e. when some $g_i = f$.

A sufficient condition for termination is that arguments must be smaller along a well-founded order on function's domain:

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

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

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

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

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

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

$$\begin{aligned} R^1 &\equiv R \\ R^{n+1} &\equiv R \circ R^n, \text{ for } n \geq 1 \\ R^+ &\equiv \bigcup_{n \geq 1} R^n \end{aligned}$$

So $a\, R^+\, b$ iff. $a\, R^i\, b$ for some $i \geq 1$.

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

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

Example: Consider the following function:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

`fac n` has only `fac (n - 1)` as a recursive call, and $n > n - 1$. Here, $>$ is the standard ordering over the natural numbers. Therefore, the function terminates.

2.5.2 Proofs

Consider the following program:

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

Can we prove that `maxi n m >= n`? We to a **reasoning by cases:**

We have $n \geq m \vee \neg(n \geq m)$. Now we show that `maxi n m >= n` for both cases:

- Case 1: $n \geq m$, then `max n m = n` and $n \geq n$.

- Case 2: $\neg(n \geq m)$, then `maxi n m = m`. But $m > n$, so `maxi n m >= n`.

But how do we prove a formula P (with free variable n), for all $n \in \mathcal{N}$? For example, how do we prove the following equality:

$$\forall n \in \mathcal{N}. 0 + 1 + 2 + ... + n = n \cdot (n+1)/2$$

We can do a **proof by induction:**

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

Example: We have the following conjecture: $\forall n \in \mathcal{N}.(\text{sumPowers } n) + 1 = \text{power2 } (n+1)$ with the following code:

```
power2 :: Int -> Int
power2 0 = 1
power2 r = 2 * power2 (r - 1)

sumPowers :: Int -> Int
sumPowers 0 = 1
sumPowers r = sumPowers (r - 1) + power2 r
```

We want to proof: Let $P \equiv (\text{sumPowers } n) + 1 = \text{power2 } (n+1)$. We show $\forall m \in \mathcal{N}.P$ by induction on n.

Base case: Show $P[n \to 0]$:

$$(\text{sumPowers } 0) + 1 = 1 + 1 = 2$$
$$\text{power2 } (0+1) = 2 \cdot \text{power2 } 0 = 2 \cdot 1 = 2$$

Step case: Assume $P[n \to m]$ for an arbitrary m (not in P), i.e.

$$(\text{sumPowers } m) + 1 = \text{power2 } (m+1)$$

and prove $P[n \to m+1]$, i.e.

$$(\text{sumPowers } (m+1)) + 1 = \text{power2 } ((m+1)+1).$$

Proof:

$$\begin{aligned}
(\text{sumPowers } (m+1)) + 1 &= \text{sumPowers } ((m+1)-1) + \text{power2 } (m+1) + 1 \quad (\text{def.}) \\
&= \text{sumPowers } (m) + 1 + \text{power2 } (m+1) \quad (\text{arithmetic}) \\
&= \text{power2 } (m+1) + \text{power2 } (m+1) \quad (\text{ind- hypothesis}) \\
&= 2 \cdot \text{power2 } (m+1) \quad (\text{arithmetic}) \\
&= \text{power2 } (m+2) \quad (\text{def.})
\end{aligned}$$

We have proven $(\text{sumPowers } n) + 1 = \text{power2 } (n+1)$.

The general schema for **well-founded induction** is given as:

- *To prove:* $\forall n \in \mathcal{N}.P$
- *Fix:* An arbitrary m not free in P
- *Assume:* $\forall l \in \mathcal{N}.l < m \to P[n \to l]$ (*induction hypothesis*)
- *Prove:* $P[n \to m]$

3 More on Haskell

3.1 Lists

3.1.1 List Type

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

- *Empty list:* `[] :: [T]`
- *Non-empty list:* `(x : xs) :: [T]`m if `x :: T` and `xs :: [T]`

Syntactic sugar: We can write `1 : (2 : (3 : []))` as `[1, 2, 3]`.

3.1.2 Patterns

Pattern matching has two main purposes:

- checks if an argument has the proper form
- binds values to variables

Example: `(x : xs)` matches with `[2, 3, 4]` and binds:

```
x  = 2
xs = [3, 4]
```

Patterns are *inductively* defined:

- Constants: `-2, '1', True, []`
- Variables: `x, foo`
- Wild card: `_`
- Tuples: `(p1, p2,..., pk)`, where `p_i` are patterns
- Non-empty list: `(p1 : p2)`, where `p_i` are patterns

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

3.1.3 Advice on Recursion

Defining a recursion is best done by obeying the following simple steps:

- Step 1: Define the type of the function
- Step 2: Enumerate all different cases
- Step 3: Define the most simple cases
- Step 4: Define the remaining cases
- Step 5: Generalize and simplify

Example: The following code snippet shows an example of how we implement *insertion sort* recursively in Haskell:

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

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

Example: The following code snippet shows how we can implement *quicksort* recursively in Haskell:

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

3.1.4 List Comprehensions

List comprehension is a notation for sequential processing of list elements. It is analogous to set comprehension in set theory, i.e. $\{2 \cdot x \mid x \in X\}$. In Haskell, this is equivalent to $[2 * x \mid x \leftarrow xs]$.

List comprehensions are very powerful! The following code snippet, again, implements *quicksort* as shown previously:

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

3.1.5 Induction over Lists

How are elements in `[T]` constructed? `[] :: [T]` and `(y : ys) :: [T]` if `y :: T` and `ys :: [T]`. This corresponds to the following rule:

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

3.2 Abstractions

3.2.1 Polymorhpic Types

If we consider the `length` function, it should output the length of a list of *any* type. We say that the type of the function is **polymorphic,** i.e. `[t] -> Int` for all types t.

This is often called **parametric polymorphism,** which is different from *subtyping polymorphism,* where methods can be applied to objects of sub-classes only.

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

It is important to note that type variables in Haskell start with a *lower-case letter!*

Example: Consider the following polymorphic types:

```
:type (++)
(++) :: [a] -> [a] -> [a]

:type zip
zip :: [a] -> [b] -> [(a, b)]

:type []
[] :: [a]
```

3.2.2 Higher-order Functions

We can distinguish the order of functions in the following way:

- First order: Arguments are base types or constructor types

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

- Second order: Arguments are themselves functions

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

- Third order: Arguments are functions, whose arguments are functions

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

- Higher-order functions: Functions of arbitrary order

Example: Consider the `map` function:

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

times2 x = 2 * x

double xs = map times2 xs
```

Example: Consider the `foldr` function:

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

sumList xs = foldr (+) 0 xs
```

3.2.3 λ-Expressions

Consider the following two functions:

```
times2 x = 2 * x
double xs = map times2 xs

atEnd x xs = xs ++ [x]
rev xs = foldr atEnd [] xs
```

Haskell provides a notation to write functions like `times2` and `atEnd` in-line via so-called **λ-expressions:**

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

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

This is also called *Church's λ-notation*, i.e. replacing λ by the character `'\'`.

3.2.4 Functions as Values

In Haskell, functions can be returned as values! Consider the following simple example where we return the two-times-application of some function f:

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

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

? twice times2 3
12 :: Int
```

3.2.5 Differece Lists

Difference lists are functions `[a] -> [a]` that prepend a list to its argument.

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

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

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

app :: DList a -> DList a -> DList a
ys 'app' zs = \xs -> ys (zs xs)         -- concatenation

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

toList :: DList a -> [a]
toList ys = ys []                       -- conversion to lists
```

3.2.6 Partial Application

Functions of multiple arguments can be **partially applied.** Consider the following example:

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

? :type multiply 7
Int -> Int

? :type map
(a -> b) -> [a] -> [b]

? map (multiply 7) [1, 2, 3, 4]
[7, 14, 21, 28] :: [Int]
```

It is important to note here that each function takes *exactly one argument!* Consider `multiply :: Int -> Int -> Int` means `multiply :: Int -> (Int -> Int)`. Therefore, the application `multiply 2 3` means `(multiply 2) 3`.

Furthermore, we might use **tuple arguments.** They may are equivalent to multiple-argument functions, however they do no not allow partial application!

4 Higher-Order Programming and Types

4.1 Overview

4.1.1 Implement a Function with foldr

1. Identify the recursive argument and static and dynamic arguments

```
mystery a b c []       = a + b - c
mystery a b c (x : xs) = mystery x (b + c) c xs
```

2. Write a helper with only recursive (first) and dynamic arguments

```
aux []       a b = a + b - c
aux (x : xs) a b = aux xs x (b + c)
```

3. Move the dynamic arguments to the right of the equals

```
aux []       = \a b -> a + b - c
aux (x : xs) = \a b -> aux xs x (b + c)
```

4. Rewrite aux using `foldr` replacing `aux xs` with local variable `rec`

```
aux = foldr (\x rec a b -> rec x (b + c)) (\a b -> a + b - c)
```

5. Inline `aux`

```
mystery a b c xs =
  foldr (\x rec a b -> rec x (b + c)) (\a b -> a + b - c) xs a b
```

4.2 Case Study: Operations on Vectors and Matrices

Vectors and vector addition can be easily defined by:

```
type Vector = [Int]

vecAdd :: Vector -> Vector -> Vector
```

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

We could also use `zipWith`, which is a combination of `map` and `zip`. This would look as follows:

```
vecAdd :: Vector -> Vector -> Vector
vecAdd = zipWith (+)
```

An $n \times m$ **matrix** can be represented *column-wise* using lists. We might write this like:

```
type Matrix = [Vector]

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

Some other matrix-related definitions:

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

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

Transposing of a matrix can be implemented as follows:

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

Another very important operation in linear algebra is the **dot product.** We propose different ways to implement it in Haskell:

```
-- Version 1: Loop / accumulator
skProd :: Vector -> Vector -> Int
skProd xs ys = loop xs ys 0
    where
        loop []     []     0 = p
        loop (x:xs) (y:ys) p = loop xs ys (x * y + p)

-- Version 2: Explicit recursion
skProd :: Vector -> Vector -> Int
skProd (x:xs) (y:ys) = x * y + skProd xy ys
skProd _      _      = 0

-- Version 3: Using library functions
skProd :: Vector -> Vector -> Int
skProd v w = sum (zipWith (*) v w)
```

Finally, we can go to the most interesting problem: **matrix multiplication.** WE first start by multiplying an $n \times m$ matrix A with vector b of size m, which is equivalent to the scalar product of A's rows (i.e. the columns of `tr` A) with b:

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

With this problem solved, matrix multiplication simply iterates `vecMult` A over an $m \times k$ matrix B:

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

5 Typing

5.1 Overview

Type checking should prevent "dangerous expressions", such as `2 + True`, `[2] : [3]`, etc. Dangerous expressions lead to *runtime errors.*

The objectives for a type checker are as follows:

- Quick, decidable, static analysis
- Permit as much generality / re-usability as possible
- Prevent runtime errors

5.2 Mini-Haskell