Functional Programming | FP

Summary

CONTENTS

1. FUNCTIONAL LANGUAGE

Term	*Definition*
Functional programming	Style of programming with the focus on application of functions to arguments
Functional language	Languages that encourage functional programming

1.1. TERMS

Term	*Definition*
Mutual recursion	Two or more functions defined recursively in terms of each other (=calling each other)
Higher-order functions	A function is called higher-order if it takes a function as an argument or returns a function as a result.
Effectful programming	“Effectual” simply means that arguments and return values are no longer just plain (pure) values, but may also have so-called “effects” such as: – The possibility of failure, e.g. using the option type Maybe – Aggregating multiple results, e.g. using the list type [] – Performing IO, e.g. using the action type IO
Referential transparency	– **no** (mutable) variables – **no** assignments – **no** imperative control structures – all data structures are **immutable**

1.2. BASIC FEATURES OF FUNCTIONAL PROGRAMMING

– Referential transparency
– Functions as first-class citizens
– Higher-order functions
– Algebraic data types
– Pattern matching
– Recursion
– Types & type inference
– Haskell Specific: Type classes, Functors, Applicatives, Monads
– A Declarative Programming Paradigm.
– Foundation: Church’s Lambda calculus.
– Pure: No (or controlled) mutable state. Expressions are (by default) side effect free.

1.3. PROGRAMMING PARADIGMS

Paradigm	*Definition*
Imperative/Procedural	Normie
Object Oriented	Corpochud
Functional	Gigachad

Paradigm	*Definition*

Array	`Xy ← °⍉˙⊠⊟÷⍥⍤↑`
	`F  ← ⍉◿1⊂⊃(+/÷|÷3+1∿×⍳+)Xy`
	`≡F100 ÷⍥⍤↑10`

2. HASKELL

2.1. FUNCTION DEFINITION

```haskell
funName :: (TypeBound a) ⇒ a → b        -- type definition
funName x = fnRequiringTypeBound x      -- implementation
```

2.1.1. Curried functions

```haskell
add :: Int → Int → Int
add x y = x + y

add2 :: Int → Int
add2 = add 2                            -- curried
```

2.1.2. Guarded Equations

```haskell
abs n
    | n ⩾ 0 = n                         -- fancy if else
    | otherwise = -n
```

2.1.3. Pattern matching

Patterns consist of variables and data constructors. They are matched in order, so more specific patterns should come first.

```haskell
not :: Bool → Bool
not False = True
not True = False
-- ═══
not b = case b of
          True → False
          False → True
```

2.1.4. List patterns

```haskell
head :: [a] → a
head (x : _) = x                        -- first elem
tail :: [a] → [a]
tail (_ : xs) = xs                      -- rest of list
```

2.1.5. Lambda expressions

```haskell
-- \boundvar → body
(\x → x + x) 5
```

2.2. FUNCTION APPLICATION

```
-- function argument
sum [1..5]
```

2.2.1. Point-free notation

== Function composition

```
odd x = not (even x)
odd = not . even
```

The term “point-free” does not refer to the “.” character used for function composition (the number of which typically increase), refers to the fact that the definition does not mention the data points on which functions act.

Every definition has a point-free form that can be computed automatically! http://pointfree.io

2.3. OPERATORS

- Prefix notation is used for function application: `max 7 2`
- If infix notation is desired, one may use so-called operators (e.g. +): `7 + 2`
- Functions can be converted into operators using backticks: `7 `div` 2`
- Operators can be converted into functions using brackets: `(+) 7 2`

Notes:

- Operator symbols are formed from one or more symbol characters `! # $ % & * + . / < = > ? @ ^ | - ~ :` (or any Unicode symbol or punctuation).
- An operator symbol starting with : may only be used for data constructors (e.g. : is the constructor for lists).
- A so-called fixity declaration is used to specify the associativity and precedence level (1 (highest) to 9 (lowest)) of an operator. E.g.:
  - `infixl 6 +` specifies (+) to be left associative with level 6
  - `infixr 5 :` specifies (:) to be right associative with level 5
  - `infix 4 ==` specifies (==) to be neither left nor right associative (making parentheses mandatory while nesting), and with level 4
- Any operator lacking a fixity declaration is assumed to be `infixl 9`.
- Function application has a higher precedence than any infix operator.

2.4. CONDITIONAL EXPRESSIONS

Always ternary.

```
abs :: Int → Int
abs n = if n ⩾ 0 then n else -n
```

2.5. LIST COMPREHENSION

$$\{x^2 \mid x \in \{1, 2, ..., 5\}\}$$

$$\{x^2 \mid x \in \{1, 2, ..., 12\}, 12 \bmod x \equiv 0\}$$

$$\{(x, y) \mid x \in \{1, 2\}, y \in \{4, 5\}\}$$

$$\{(x, y) \mid x \in \{1, 2\}, y \in \{x, ..., 2\}\}$$

```
[x ^ 2 | x ← [1 .. 5]]                  -- [1, 4, 9, 16, 25]
-- guards
[x | x ← [1 .. 12], 12 `mod` x == 0]    -- [2, 3, 4, 6, 12]
-- multiple generators
[(x, y) | x ← [1, 2], y ← [4, 5]]       -- [(1,4),(1,5),(2,4),(2,5)]
-- dependant generators
[(x, y) | x ← [1 .. 2], y ← [x .. 2]] -- [(1,1),(1,2),(2,2)]
```

2.6. TYPES

A Type in Haskell is a name for a collection of related values.

2.6.1. Type declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

Type declarations can also have type parameters

```
type Pair a = (a, a)

mult :: Pair Int → Int
mult (m, n) = m * n
```

Type declarations can be nested but cannot be recursive

```
type Pos = (Int, Int)
type Trans = Pos → Pos      -- OK

type Tree = (Int, [Tree])   -- NOK
```

2.6.2. Data declarations

We can define new custom data types by declaring new types and functions that return values of these types. These functions (a.k.a. “data constructors” or “constructors”) are special since they cannot have a definition that results in a reduction rule. Thery therefore remain unchanged during execution and can be used to hold information.

```
data List a = Nil | Cons a (List a)
```

`List` has values of the form of `Nil` and `Cons` where a is a generic. `Nil` and `Cons` can be viewed as functions that construct values of type `List`.

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

- The two values `False` and `True` are called the constructors for the type `Bool`.
- Type and constructor names must always begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Such data types are often referred to as algebraic datatypes.

- Type variables within types are implicitly universally quantified at the topmost level. For example, `a → Maybe a` actually denotes `forall {a}. a → Maybe a`. The `{}` brackets around `a` denotes that its instantiation can only be done automatically.
- You may ask ghci to explicitly print universal quantifiers as follows:

```
Prelude> :set -fprint-explicit-foralls
Prelude> :t Just
Just :: forall {a}. a → Maybe a
```

– You may explicitly specify the universal quantification in polymorphic type signatures using the `ExplicitForAll` extension.

Maybe and Pair are called Type Constructors since they construct one type from another. This behaviour is expressed using kinds, which is the type of a “type”. The type system for kinds is very simple:

`K ::= * | K → K`
`*` Kind of ordinary/proper/concrete/basic types
`→` Kind of “type constructor” or “type operators”

– This is essentially the “simply typed lambda calculus” one level up, with one base type.
– Ordinary/proper/concrete/basic types are nothing more than nullary type constructors/operators. Analogy: A constant is nothing more than a nullary function.
– The word “type” is therefore often used for any type-level expression: ordinary/proper/concrete/basic types, and for type constructors/operators.

Examples: (Only types of kind * can have a value)

Kind	`*`	`*`	`*`	`*`	`*`	`* → *`	`* → * → *`
Type	`Bool`	`Char`	`[a] → [a]`	`Maybe Char`	`a → Maybe a`	`Maybe`	`(,)`
Value	`True`	`'a'`	`reverse`	`Just 'a'`	`Just`		

`(→)` is also a type constructor!

Kind	`*`	`*`	`* → * → *`	`* → *`	`* → *`
Type	`Bool → Bool`	`(→) Bool Bool`	`(→)`	`(→) Bool`	`(→) a`
Value	`not`	`not`			

The operator `(→)` is overloaded:
– In the context of a type (e.g. `id :: a → a`), it denotes the type constructor for function types.
– In the context of a kind (e.g. `List :: * → *`), it denotes the kind constructor for type constructors.

2.6.2.1. Records

Convenient syntax when data constructors have multiple parameters:

```
data Person = Person
  { name :: String
  , age :: Int
  , children :: [Person]
  } deriving (Show)

-- ⟺

data Person = Person String Int [Person] -- positional constructor
  deriving (Show)
-- v selector functions v
name :: Person → String
name (Person n _ _) = n
```

```haskell
age :: Person → Int
age (Person _ a _) = a

children :: Person → [Person]
children (Person _ _ c) = c
```

Construction:

```haskell
alice :: Person
alice = Person "Alice" 5 []

bob :: Person
bob = Person {name = "Bob", age = 35, children = [alice]}
```

Derived show contains labels and updates can use field labels:

```haskell
--  >>> alice{age = age alice + 1}
-- Person {name = "Alice", age = 6, children = []}
```

2.6.3. Polymorphism

= “Many forms”

Term	*Definition*
Ad-hoc Polymorphism	Function with the same name denotes different implementations (function overloading, Interfaces)
Parametric Polymorphism	Code written to work with many possible types (OO: generics)
Subtype Polymorphism	One type (subtype) can be substituted for another (supertype)

2.6.4. Typeclasses

Haskell uses type classes to achieve ad-hoc polymorphism.

```haskell
elem :: Eq a ⇒ a → [a] → Bool -- Eq is a typeclass
```

Type classes are declared using class declarations:

```haskell
-- Eq = name of the declared type class
class Eq a where                -- a = type parameter
  (==), (/=) :: a → a → Bool -- Required to be a member of Eq
  x /= y = not (x == y)         -- Default implementation (optional)
```

Type classes may be extended:

```haskell
-- Eq = name of the type class to be extended
class Eq a ⇒ Ord a where       -- Ord = name of resulting type class
  (<), (⩽), (⩾), (>) :: a → a → Bool -- Additional requirements
```

– All members of the type class `Ord` are also members of the type class `Eq`.
– Members of `Ord` therefore also require `(==)` to be defined.

A type can be declared to be an instance of a type class using an instance declaration:

```haskell
-- In order for (Maybe m) to be a member of the type class Eq, it is
-- required that the type m belongs to the type class Eq
instance Eq m ⇒ Eq (Maybe m) where
  Just x == Just y   = x == y
```

```
  Nothing == Nothing = True
  _ == _             = False
```

Just like it is possible to have higher- order functions, it is also possible to have higher-kinded types.

As far as the compiler is concerned, the only requirement for a type to be a member of a type class is that it should have a type correct instance declaration. There are of course additional semantic requirements that need to hold. These additional requirements are expressed as type class laws within the documentation of each type class.

Default implementations for some type classes can be generated automatically for data declarations using the deriving keyword:

```
data Bool = False | True
  deriving (Eq, Ord, Show, Read)
```

2.6.5. Newtype declarations

In cases where a datatype only has one constructor, a newtype declaration is used.

```
newtype ListOfBool = [Bool]
```

2.6.6. Instances

2.6.6.1. Functor

Functors are types that can be used to **wrap and extract values** of other types, and allow us to **lift unary functions** over the wrapped value.

Definition

```
class Functor f where
  fmap :: (a → b) → f a → f b
  (<$) :: a → f b → f a
  {-# MINIMAL fmap #-}
```

Type Class Laws

The `fmap` function **should not change the structure** of the Functor, **only its elements**.

An instance of Functor has to abide by the following laws:

1) `fmap` has to preserve identity
   - `fmap id == id`
2) `fmap` has to preserve function composition
   - `fmap (g . h) == fmap g . fmap h`

fmap

```
-- <$> == fmap
(<$>) :: Functor f ⇒ (a → b) → f a → f b
```

Examples

```
(+1) <$> Nothing                    -- Nothing
(+1) <$> Just 1                     -- Just 2
```

Instances

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

2.6.6.2. Applicative

What if we want to lift an n-ary function?

Applicative Functors provide a **generic function to wrap values** (`pure`) and **generalized function application** (`<*>`).

Definition

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
  liftA2 :: (a → b → c) → f a → f b → f c
  (*>) :: f a → f b → f b
  (<*) :: f a → f b → f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Type Class Laws

1) Identity: `pure` has to preserve **identity**
   – `pure id <*> x == x`
2) Homomorphism: `pure` has to preserve **function application**
   – `pure (g x) == pure g <*> pure x`
3) Interchange: **Order of evaluation** must not matter when applying an effectful function to a pure argument
   – `x <*> pure y == pure (\f → f y) <*> x`
4) Composition: `<*>` has to be **associative**
   – `x <*> (y <*> z) == (pure (.) <*> x <*> y) <*> z`

pure

We also need a function pure to lift 0-ary functions (constants):

```
pure :: a → f a
f <$> x == (pure f) <*> x
```

Similarities to plain function application

```
($)   ::   (a → b) →   a →   b
(<*>) :: f (a → b) → f a → f b

(+) $ 11 $ 31                              -- 42
Just (+) <*> Just 11 <*> Just 31           -- Just 42
```

Examples

```
(+) <$> Nothing <*> Just 2                 -- Nothing
(+) <$> Just 1 <*> Just 2                  -- Just 3

(,) <$> [1,2] <*> [6,7]                    -- [(1,6),(1,7),(2,6),(2,7)]

-- intuition: all possible ways of multiplying the two input lists
pure (*) <*> [1,2] <*> [3,4]               -- [3,4,6,8]

-- in the context of lists, pure (*) == [(*)]
[(*),(+)] <*> [1,2] <*> [3,4]              -- [3,4,6,8,4,5,5,6]
```

Instances

```haskell
instance Applicative Maybe where
  pure x = Just x
  Nothing <*> _ = Nothing
  (Just f) <*> mx = fmap f mx

instance Applicative IO where
  pure = return
  mf <*> mx = do
    f ← mf
    x ← mx
    return (f x)
```

2.6.6.3. Monad

So far, we have seen how functors and applicative functors help us to:

– apply pure functions to “effectful” arguments
– compose function application with multiple “effectful” arguments

What is missing is how to apply and compose “effectful” functions with “effectful” arguments.

Definition

```haskell
class Applicative m ⇒ Monad m where
  (>>=) :: m a → (a → m b) → m b
  (>>) :: m a → m b → m b
  return :: a → m a
  {-# MINIMAL (>>=) #-}
```

Type Class Laws

1) Left identity
   – `return x >>= f = f x`
2) Right identity
   – `mx >>= return = mx`
3) Associativity
   – `(mx >>= f) >>= g = mx >>= (\x → (f x >>= g))`

Bind

Function that binds an effectful value into an effectful function

```haskell
(>>=) :: m a → (a → m b) → m b
```

Kleisi arrow (not in haskell)

The Kleisli arrow (<=<) is analogous to normal function composition, except that it works on monadic functions

```haskell
(.)   ::            (b →   c) → (a →   b) → a →   c
(<=<) :: Monad m ⇒ (b → m c) → (a → m b) → a → m c
(f <=< g) x =  g x >>= f
```

Examples

```haskell
safediv 8 0 >>= flip safediv 2              -- Nothing
safediv 8 2 >>= flip safediv 2              -- Just 2
```

```haskell
safediv :: Int → Int → Maybe Int
safediv n m = if m == 0 then Nothing else Just $ div n m

do {x ← [1,2]; y ← [3,4]; return (x,y)}   -- [(1,3),(1,4),(2,3),(2,4)]
-- ⟺
[1,2] >>= \x → [3,4] >>= \y → return (x,y)
-- ⟺
[1,2] >>= \x → [3,4] >>= return . (x,)
-- ⟺
[1,2] >>= \x → (x,) <$> [3,4]
-- ⟺
(,) <$> [1,2] <*> [3,4]
```

Instances

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

instance Monad [] where
  xs >>= f = [y | x ← xs, y ← f x]
```

2.6.6.4. Comparison

	Application operator	*Type of application operator*	*Function*	*Argument*	*Result*
Pure function application	juxtapose, `($)`	`(a → b) → a → b`	pure	pure	pure
Functor	`fmap`, `(<$>)`	`(a → b) → f a -> f b`	pure	effectual	effectual
Applicative functor	`(<*>)`	`f (a → b) → f a → f b`	pure	effectual	effectual
Monad	`(>>=)`	`m a → (a → m b) → m b`	effectual	effectual	effectual

2.7. MODULES

– A collection of related values, types, classes, etc.
– Name: Identifier that starts with a capital letter.
– Naming convention for hierarchy: separated using “.” (e.g. `Data.List`, `Data.List.NonEmpty`, `Data.Set`, ...)
– Each module `Dir.Name` must be contained in a file `Dir/Name.hs`.

```haskell
module BinarySearchTree
( T, toList
) where

data T a = Leaf | Node (T a) a (T a)
  deriving Show
```

```haskell
toList :: T a → [a]

-- ...

import qualified BinarySearchTree
  as Set (T, toList)
```

2.8. LAZY EVALUATION

– Perform an evaluation step only when it is necessary.
– Never perform the same step twice.

2.9. SIDE EFFECTS

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

```haskell
IO a -- The type of actions that return a value of type a
```

One may also use a let statement within a do block to perform a pure computation. Note that the let statement here is different from the let … in … statement for expresisons.

```haskell
act :: IO (Char, Char)
act = do {x ← getChar;
          y ← getChar;
          let {p = (x,y)};
          return p}
-- ⟺

act = do
        x ← getChar
        y ← getChar
        let p = (x,y)
        return p
```

Composing IO actions

```haskell
(>>=) :: Monad m ⇒ m a → (a →  m b) →  m b

do {x ← getChar; putChar x}
-- ⟺
getChar >>= putChar
```

3. LAMBDA CALCULUS

Created by Alonzo Church.

3.1. SYNTAX

```
<M>       ::= <id> | λ<id>. <M> | <M> <M> | ( <M> )
<id> ::= a | b | c | ... | 0 | 1 | 2 | ... | + | - | * | ...
```

Reserved words: “λ”, “.”, “(“, “)”

$$\Leftrightarrow$$
$$\underbrace{M}_{\lambda-\text{term}} ::= \underbrace{x}_{\text{Id}} \mid \underbrace{\lambda x.\ M}_{\lambda\text{ abstraction}} \mid \underbrace{M\ M}_{\text{Application}}$$
$$\Leftrightarrow$$

```
type Id = String
data Term
  = Var Id
  | Abs Id Term
  | App Term Term
```

Rules:

– The scope of a λ abstraction extends as far right as possible: $\lambda x.\ x\ \lambda y.\ y\ x \Leftrightarrow (\lambda x.\ (x\ (\lambda y.\ (y\ x))))$
– Application is left associative: $M_1\ M_2\ M_3 \Leftrightarrow (M_1\ M_2)\ M_3$

3.2. DEFINITIONS

Bound variables: Bound variables are placeholders that refer to their binding occurrences. Their names have no significance and can be renamed (α-conversion).

$$\lambda\ \underbrace{x}_{\substack{\text{binding}\\\text{occurrence}}}\ .\ \underbrace{x}_{\text{bound}}\ \underbrace{z}_{\text{free}}$$

Free variables: Variables that aren’t bound

Open: A lambda term **with at least one free variable** is said to be ***open***

Closed: A lambda term **without any free variables** is said to be ***closed***

Combinators: **Closed lambda terms** are also called ***combinators***, since all they do is combine their parameters in different ways

Closure: A lambda term whose free variables are bound by its enclosing scope

(β) Normal form: A λ term is said to be in ***normal form*** if **no further reductions** can be applied to it

Confluence: Every λ term has **at most one normal form**. This property is sometimes referred to the ***confluence property*** of β reduction

3.2.1. nfin

“Not Free In”

```
nfin :: Id → Term → Bool
x `nfin` (Var y) = x /= y
x `nfin` (Abs y m) = x == y || nfin x m
x `nfin` (App m n) = x `nfin` m && x `nfin` n
```

3.2.2. substitution

The term $[x := L]M$ denotes the term M with all free occurrences of the variable x replaced by the term L.

```
-- NOTE: renameBoundVars does what it sounds like
type Subst = (Id, Term)
applySubst :: Subst → Term → Term
applySubst (x, l) (Var y) = if x == y then l else Var y
applySubst (x, l) (Abs y m)
```

```
  | x == y                = Abs y m
  | x /= y && y `nfin` l  = Abs y $ applySubst (x, l) m
  | otherwise
    =
      applySubst (x, l) $ alphaConvert (Abs y m) newId -- newId `nfin` m
applySubst s (App m n)    = App (applySubst s m) (applySubst s n)
```

3.2.3. α conversion

$$\lambda x.\ M = \lambda y.\ \underbrace{[x := y]}_{\text{substitution}}\ M \quad \text{if} \quad \underbrace{(y \operatorname{nfin} \lambda x.\ M)}_{\substack{\text{needed to avoid} \\ \text{variable capture}}}$$

```
alphaConvert :: Term → Id → Term
alphaConvert (Abs x m) y =
  if y `nfin` m
    then Abs y $ applySubst (x, Var y) m
    else Abs x m
alphaConvert x _ = x
```

***Example** 1:*

$$\begin{aligned} & \lambda \underline{x}.\ \underline{x}\ z \\ = & \lambda \underline{b}.\ \underline{b}\ z \quad \because \alpha \\ = & \lambda n.\ n\ z \because \alpha \end{aligned}$$

3.2.4. β reduction

Replacing formal parameters (placeholders) with actual parameters (values)

$$(\lambda x.\ M)\ N = [x := N]M$$

```
betaReduce :: Term → Maybe Term
betaReduce (App (Abs x m) n) = applySubst (x, n) m
betaReduce = id
```

***Example** 2:*

$$\begin{aligned} & \underline{((\lambda x.\ (\lambda y.\ y))\ ((\lambda x.\ x\ x)\ (\lambda x.\ x)))}\ ((\lambda x.\ x)\ z) \\ = & \underline{(\lambda y.\ y)\ ((\lambda x.\ x)\ z)} && \because \beta \\ = & \underline{(\lambda x.\ x)\ z} && \because \beta \\ = & z && \because \beta \end{aligned}$$

3.2.5. δ reduction

Although not strictly necessary, it is very useful to introduce abbreviations for λ terms to make them more readable. This can be done by introducing definitions as equalities of the following form to a context within which a term can be reduced.

$$x = M$$

The substitution of a defined symbol with its definition is referred as δ reduction

***Example** 3:*

$$
\begin{aligned}
& \underline{\vee}\ \top\ \bot = \top && \\
\Leftrightarrow\ & \underline{(\lambda p.\ \lambda q.\ p\ p\ q)\ \top}\ \bot = \top && \because \delta \\
\Leftrightarrow\ & \underline{(\lambda q.\ \top\ \top\ q)\ \bot} = \top && \because \beta \\
\Leftrightarrow\ & \underline{\top}\ \top\ \bot = \top && \because \beta \\
\Leftrightarrow\ & \underline{(\lambda x.\ \lambda y.\ x)\ \top}\ \bot = \top && \because \delta \\
\Leftrightarrow\ & \underline{(\lambda y.\ \top)\ \bot} = \top && \because \beta \\
\Leftrightarrow\ & \top = \top && \because \beta
\end{aligned}
$$

3.2.6. η contraction

For any f that does not contain any free occurances of x the following equality holds:

```
(\x → f x) == f
\x → f (g x) == f . g
```

$$
\begin{aligned}
\lambda x.\ f\ x &\Leftrightarrow f \\
\lambda x.\ f\ (g\ x) &\Leftrightarrow f\ g
\end{aligned}
$$

3.2.7. As proof trees

$$\frac{}{\mathrm{H} \vdash (\lambda x.M)N = [x := N]M}\,\beta \qquad \frac{}{\mathrm{H} \vdash M = M} = goal$$

$$\frac{\mathrm{H}, M = N \vdash [x := N]P}{\mathrm{H}, M = N \vdash [x := M]P} = hyp \qquad \frac{\mathrm{H} \vdash P \quad \mathrm{H}, P \vdash Q}{\mathrm{H} \vdash Q}\,cut$$

3.3. EVALUATION

Innermost Redex: Every redex not containing any other redex. A term can contain several innermost redexes

Outermost Redex: Every redex not contained in any other redex A term can contain several outermost redexes

Leftmost Innermost Redex: The leftmost redex not containing any other redex. A term can contain at most one leftmost innermost redex

Leftmost Outermost Redex: The leftmost redex not contained in any other redex. A term can contain at most one leftmost outermost redex

Leftmost Innermost Strategy: The leftmost innermost redex is reduced in each step

Leftmost Outermost Strategy: The leftmost outermost redex is reduced in each step

3.3.1. Relation to parameter passing

Call by value: Leftmost innermost reduction except that no reductions are performed within λ abstractions.

Call by name: leftmost outermost reduction except that no reductions are performed within λ abstractions.

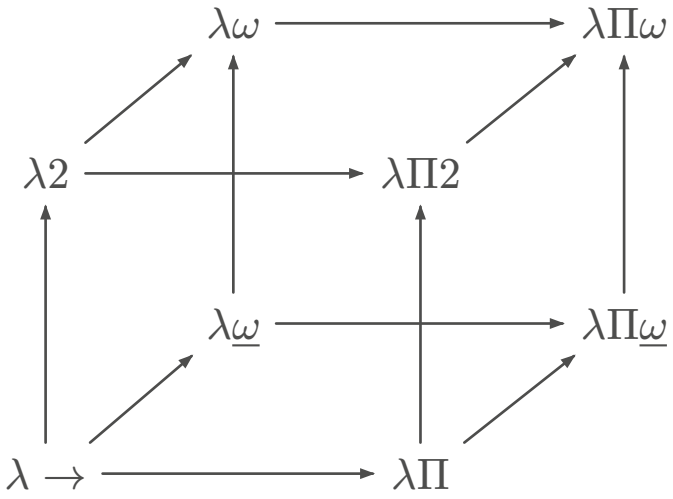
Lazy evaluation: Used in order to make call by name more efficient. Terms that are duplicated during function application are shared such that they are evaluated at most once.

3.4. ENCODING DATA AND OPERATIONS

Term	*Encoding*
True	$\top = \lambda x.\ y\ x$
False	$\bot = \lambda x.\ y\ y$
Conjunction	$\wedge = \lambda p.\ \lambda q.\ p\ q\ p$
Disjunction	$\vee = \lambda p.\ \lambda q.\ p\ p\ q$
Negation	$\neg = \lambda p.\ p\ \bot\ \top$
Zero	$0 = \lambda f.\ \lambda x.\ x$
One	$\lambda f.\ \lambda x.\ f\ x$
Two	$\lambda f.\ \lambda x.\ f\ (f\ x)$
Successor	$\text{succ} = \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
Addition	$+ = \lambda m.\ \lambda n.\ m\ \ \text{succ}\ \ n$
Multiplication	$* = \lambda m.\ \lambda n.\ m\ (+\ n)\ 0$
Power	$\text{power} = \lambda b.\ \lambda e.\ e\ b$
Predecessor	$\text{pred} = \lambda n.\ \lambda f.\ \lambda x.\ n\ (\lambda g.\ \lambda h.\ h\ (g\ f))\ (\lambda u.\ x)\ (\lambda u.\ u)$
Minus	$- = \lambda m.\ \lambda n.\ n\ \ \text{pred}\ \ m$
Is zero	$\text{isZero} = \lambda n.\ n\ (\lambda x.\ \bot)\ \top$
Less than or equal	$\leq = \lambda m.\ \lambda n.\ \ \text{isZero}\ \ (-\ m\ n)$
Are equal	$\text{areEqual} = \lambda m.\ \lambda n.\ (\wedge\ \ (\leq\ \ m\ n)\ (\leq\ \ m\ n))$
Pair	$\text{pair} = \lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y$
First	$\text{fst} = \lambda p.\ p\ \top$
Second	$\text{snd} = \lambda p.\ p\ \bot$

3.5. LAMBDA CUBE

https://www.cs.uoregon.edu/research/summerschool/summer23/_lectures/oplss-lambda-cube1.pdf



Origin: $\lambda\rightarrow$: Simply typed lambda calculus Terms may only depend on Terms Curry-Howard correspondence for $\lambda\rightarrow$: Propositional calculus restricted to only use implication.

Going up (2): $\lambda 2$: System F, second-order lambda calculus Terms may depend on Types (polymorphism, e.g. (Church-style) $\lambda\alpha : * . \lambda x : \alpha . x : \forall\alpha . (\alpha \rightarrow \alpha)$, or (Curry-style) $\lambda x . x : \forall\alpha . (\alpha \rightarrow \alpha)$) Curry-Howard correspondence for $\lambda 2$: fragment of second-order intuitionistic logic that uses only universal quantification.

Going inwards (ω): Types may depend on Types (type operators, e.g. `List a` is a type, where List is a type operator with kind `* → *`) Not very interesting in isolation. Normally combined with $\lambda 2$ (System F) to give $\lambda\omega$ (System F ω) (a variant of this (System FC) is used in Haskell) Curry-Howard correspondence for $\lambda\omega$ (System Fω): Higher-Order Logic.

Going rightwards (Π, or P): Types may depend on values (dependent types, e.g. `FloatList 3` is a type denoting a list of floats with length 3, where `Floatlist : Nat → *`) $\lambda\Pi$: also called λP, LF Curry-Howard correspondence for $\lambda\Pi$: A form of predicate calculus that only uses implication and universal quantification.

Richest calculus of all 8: $\lambda\Pi\omega$: Calculus of Constructions (CC, CoC, λC)

4. PROOFS

PAT: Propositions As Types / Proofs As Terms (using types to prove the correctness of your program)

Propositional logic: The simplest form of formal logic, dealing with statements (called propositions).

Proposition: Can be either true ($\top$) or false ($\bot$)

First-order logic/Predicate logic: Extension of propositional logic. It introduces variables, quantifiers, and predicates to express more nuanced statements about objects and their relationships.

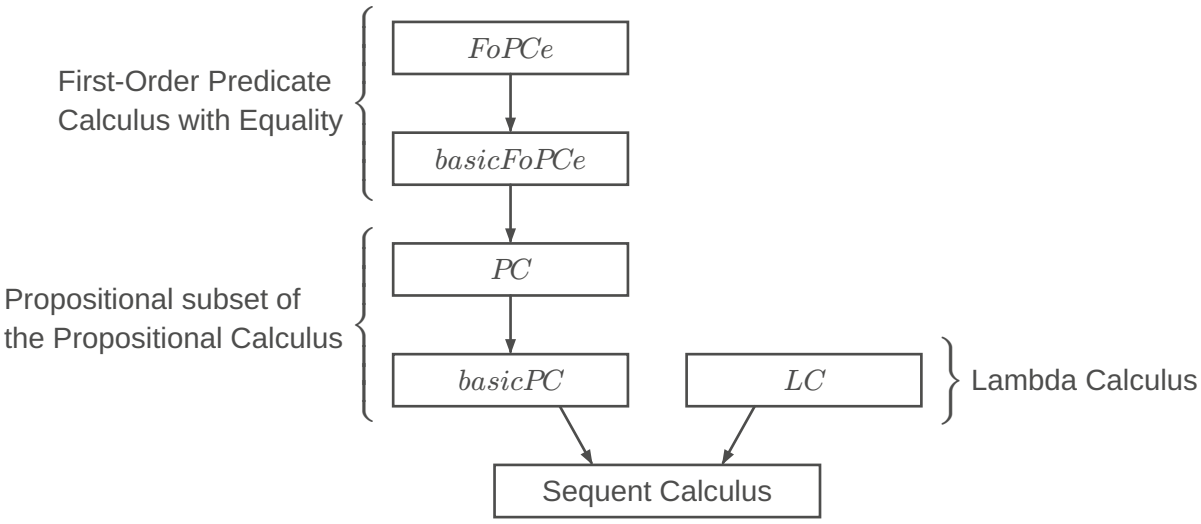
Premise: Statement or assumption that you accept as true as the starting point for reasoning. Premises are the givens in a logical argument.

Sound reasoning: Logical argument is both valid and based on true premises

Propositional Calculus (PC): Formal proof system for propositional logic

First-order PC with equality ($FoPCe$): A (first-order) logic system whose language includes variables, predicate symbols, function symbols, quantifiers and equality

First-order: The logic's quantifiers range over individual objects in a domain, but it does not quantify over predicates/sets/properties as objects.



4.1. NOTATION

4.1.1. Symbols

Term	*Definition*
$\wedge$	And
$\vee$	Or
$\neg$	Not
$\rightarrow$	Implies ($\Rightarrow$)
$\leftrightarrow$	Biconditional (iff/$\Leftrightarrow$/$\equiv$)
$\top$	Tautology (always true)
$\bot$	Contradiction (always false)
$\forall$	Universal quantifier (for all)
$\exists$	Existential quantifier (exists)
$\exists!$	There exists exactly one
$\nexists$	There does not exist
$\models$	Logical consequence (entails)
$\vdash$	Provability (can be proven from)

4.1.2. Proof tree

$$\frac{\text{Premise 1} \quad \text{Premise 2} \quad \ldots \quad \text{Premise n}}{\text{Conclusion}}\ \text{Rule name}$$

***Example** 4:* $\lambda \rightarrow$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\,\text{var} \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\,\text{app}_{\text{term}} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau}\,\text{abs}_{\text{term}}$$

***Example** 5:* **Type inference**

$$\frac{\texttt{f} :: \texttt{A} \to \texttt{B} \quad \texttt{e} :: \texttt{A}}{\texttt{f e} :: \texttt{B}}$$

4.1.3. Two-Column Proof

Format:

Statement	Justification

***Example** 6:*

1. $P \to Q$	Premise
2. P	Premise
3. Q	From 1, 2 by Modus Ponens

4.1.4. Structured Proof

Format:

Theorem: [Statement to prove]

Proof:

1. [First statement with justification]
2. [Second statement with justification]

...

n. [Final conclusion]

QED (or □)

***Example** 7:*

Theorem: If P and $P \to Q$, then Q.

Proof:

1. Assume P and $P \to Q$ as premises. (Assumption)
2. From $P \to Q$ and P , we conclude Q. (Modus Ponens)

□

4.1.5. Sequent calculus

Format:

$$\Gamma \vdash \Delta$$

Where:

- Γ = ***antecedent*** = set of premises (assumptions)
- $\vdash$ = “proves” or “entails”
- Δ = ***succedent*** = conclusion(s)

***Example** 8:*

$$P \to Q, P \vdash Q$$

4.2. FORMULA

A formula is a symbolic expression built from logical operators, variables, quantifiers, and predicates according to strict grammatical rules. It's the written representation of a logical statement.

4.2.1. Components of a formula

Component	*Meaning*	*Example*
Propositional variables	Simple true/false statements	P, Q, R
Predicates	Properties or relationships	$\text{Human}(x), \text{Loves}(x, y)$
Quantifiers	Universal or existential claims	$\forall x, \exists y$
Logical operators	Combine or modify statements	$\land, \lor, \neg, \to, \leftrightarrow$
Parentheses	Clarify structure and precedence	$(P \land Q) \to R$

4.3. PROOF

A proof is a sequence of logical steps that demonstrates the truth of a statement based on accepted premises and logical rules. Each step must follow necessarily from previous steps or established axioms.

4.3.1. Components of a proof

Component	*Meaning*	*Example*
Axioms/Premises	Starting assumptions accepted as true	All humans are mortal
Logical rules	Valid inference methods (e.g., modus ponens)	If $P \to Q$ and P is true, then Q is true
Intermediate steps	Conclusions derived from previous steps	Socrates is human, so Socrates is mortal
Final conclusion	The statement being proven	Socrates is mortal

4.3.2. Common proof techniques

– Direct proof
– Proof by contradiction
– Proof by "Induction" (Page 26)

4.3.3. Logical rules

A ***Rule of inference*** is a logical principle that allows you to derive a new conclusion from one or more premises.

4.3.3.1. Modus ponens

$$\frac{P \to Q \quad P}{Q}$$

Example *9:*

$$\frac{\text{If FP is based, then Haskell is based} \quad \text{FP is based}}{\text{Therefore, haskell is based}}$$

4.3.3.2. Modus tollens

$$\frac{P \to Q \quad \neg Q}{\neg P}$$

Example *10:*

$$\frac{\text{If Java is cool, then cool people write java} \quad \text{Cool people don't write java}}{\text{Therefore, java isn't cool}}$$

4.3.3.3. Hypothetical syllogism

$$\frac{P \to Q \quad Q \to R}{P \to R}$$

Example *11:*

$$\frac{\begin{array}{c}\text{If you live with other people, you are part of a society} \\ \text{If you are part of a society, you must abide by its rules}\end{array}}{\text{Therefore, if you live with other people, you must abide by their rules}}\ \text{Ragebait}$$

4.3.3.4. Disjunctive syllogism

$$\frac{P \vee Q \quad \neg P}{Q}$$

Example *12:*

$$\frac{\text{I am alive or dead} \quad \text{I am not dead :(}}{\text{Therefore, I am alive}}$$

4.3.3.5. Double negation elimination

$$\frac{\neg\neg P}{P}$$

Example *13:*

$$\frac{\text{We were not unable to meet the deadline}}{\text{We were able to meet the deadline}}$$

4.4. SEQUENT CALCULUS

A proof system for logic where judgments are written as sequents of the form $\Gamma \vdash \Delta$, in which Γ are the assumptions (antecedents) and Δ is the conclusion (consequent).

Sequent: A generic name for a statement that we want to prove

Proof rule: A device used to construct proofs of sequents. It consists of a list of antecedent sequents and a consequent sequent.

***Example** 14:* **A Proof rule for the sequent named r, with antecedents A, B and the consequent C**

$$\frac{A \quad B}{C}\, r$$

Axiom: In case the list of antecedents is empty, a proof rule is said to be an ***axiom***.

***Example** 15:* **An axiom r_a**

$$\frac{}{S}\, r_a$$

Theory: A (possibly infinite) set of proof rules, also called ***calculus***.

***Example** 16:* **Theory $\mathcal{T}$ with seven proof rules**

$$\frac{S2 \quad S3 \quad S4}{S1}\, r_1 \qquad \frac{S5 \quad S6}{S2}\, r_2 \qquad \frac{S7}{S3}\, r_3$$

$$\frac{}{S4}\, r_4 \qquad \frac{}{S5}\, r_5 \qquad \frac{}{S6}\, r_6 \qquad \frac{}{S7}\, r_7$$

4.4.1. Proofs and Validity

***Definition** 1:* **Proof**

A proof of a sequent within a given theory is a finite tree whose nodes have the following properties:
- Each node consists of a sequent and an optional proof rule of the theory.
- The root node contains the sequent we want to prove.
- A node with no proof rule has no child nodes.
- In case a node contains a proof rule:
  - Its sequent is identical to the consequent of the proof rule.
  - It has a child node corresponding to each antecedent of the proof rule.
  - The sequent of each child node is identical to its corresponding antecedent.

***Example** 17:* **Incomplete proof of the sequent $S1$ using the theory $\mathcal{T}$ just presented**

$$\dfrac{\dfrac{\dfrac{}{S5}\, r_5 \quad \dfrac{}{S6}\, r_6}{S2}\, r_2 \quad \dfrac{S7}{S3}\, r_3 \quad \dfrac{}{S4}\, r_4}{S1}\, r_1$$

The sequent $S7$ is said to be a ***pending sub-goal*** of the above proof.

Pending Sub-goals of a proof: The leaf nodes of a proof that do not contain proof rules are said to be ***pending nodes***. The sequents of such pending nodes are said to be the ***pending sub-goals*** of a proof.

iff: If and only if

Complete proof: A proof is said to be ***complete*** iff it has no pending sub-goals.

Incomplete proof: A proof is said to be ***incomplete*** iff it has at least one pending sub-goal.

***Example** 18:* **Complete proof**

$$\dfrac{\dfrac{\dfrac{}{S5}\,r_5 \quad \dfrac{}{S6}\,r_6}{S2}\,r_2 \quad \dfrac{\dfrac{}{S7}\,r_7}{S3}\,r_3 \quad \dfrac{}{S4}\,r_4}{S1}\,r_1$$

Valid Sequent: A sequent S is said to be valid with respect to a theory $\mathcal{T}$ iff there exists a complete proof of S within the theory $\mathcal{T}$.

4.5. PROPOSITIONAL CALCULUS (PC)

Predicate: A formal statement expressing a certain property that we may assume, or a certain property that we wish to prove.

***Example** 19:*

$$A \wedge B \Rightarrow B \wedge A$$

***Syntax of** $basicPC$:* **<P>** `::=` $\bot$ | $\neg$**<P>** | **<P>** $\wedge$ **<P>** with binding strength $\wedge$, then $\neg$

4.5.1. Proof rule schemas

A theory is specified using a finite set of ***proof rule schemas***. Each proof rule schema represents an infinite number of proof rules **of the same form**. A proof rule may be derived from its rule schema by instantiating its so-called ***meta variables***. For instance consider the rule schema:

$$\frac{\mathrm{H} \vdash P \qquad \mathrm{H}, P \vdash Q}{\mathrm{H} \vdash Q}\,cut$$

The letters H, P, and Q are ***meta variables***. The letter H is a meta variable standing for a finite set of predicates, whereas the letters P and Q are meta variables standing for single predicates. Replacing these meta variables by actual predicates gives us a proof rule. Here is an instance of the above rule schema:

$$\frac{A \vdash B \qquad A, B \vdash C}{A \vdash C}\,cut_{\text{instantiated}}$$

4.5.2. Proof rules of $basicPC$

$$\frac{}{\mathrm{H}, P \vdash P}\, hyp \qquad \frac{\mathrm{H} \vdash Q}{\mathrm{H}, P \vdash Q}\, mon \qquad \frac{\mathrm{H} \vdash P \quad \mathrm{H}, P \vdash Q}{\mathrm{H} \vdash Q}\, cut \qquad \frac{\mathrm{H}, \neg P \vdash \bot}{\mathrm{H} \vdash P}\, contr$$

$$\frac{}{\mathrm{H}, \bot \vdash P}\, \bot\, hyp \qquad \frac{\mathrm{H}, P \vdash \bot}{\mathrm{H} \vdash \neg P}\, \neg\, goal \qquad \frac{\mathrm{H} \vdash P}{\mathrm{H}, \neg P \vdash Q}\, \neg\, hyp$$

$$\frac{\mathrm{H} \vdash P \quad \mathrm{H} \vdash Q}{\mathrm{H} \vdash P \wedge Q}\, \wedge\, goal \qquad \frac{\mathrm{H}, P, Q \vdash R}{\mathrm{H}, P \wedge Q \vdash R}\, \wedge\, hyp$$

hyp: The proof rule schema ***hyp***, which is used to complete proofs, formally states the obvious fact that a sequent whose goal also occurs as a hypothesis is trivially valid.

The proof rule schemas ***mon*** and ***cut*** are so-called "structural" proof rule schemas since they do not refer to any logical connectives, but operate on the sequents directly to mimic meta-theoretic properties of the logic.

mon: The rule schema ***mon*** (for "monotonicity") formally states that removing (superfluous) hypotheses is a valid proof step (i.e. it does not make an invalid sequent suddenly valid).

cut: The rule schema cut allows one to add an arbitrary predicate P (sometimes referred to as a "lemma") as an hypothesis in the current proof, provided one can prove it separately.

contr: The proof rule schema ***contr*** starts a proof by contradiction: A valid way to prove something is assume that it does not hold, and from this arrive at a absurdity or contradiction.

The rest of the proof rule schemas aim to simplify individual logical operators within the goal or hypotheses. Their names have been chosen accordingly

4.5.3. Derived Logical operators for PC

$$\begin{aligned} \top &\triangleq \neg\bot && (\triangleq_{\top}) \\ P \vee Q &\neg(\neg P \wedge \neg Q) && (\triangleq_{\vee}) \\ P \Rightarrow Q &\triangleq \neg P \vee Q && (\triangleq_{\Rightarrow}) \\ P \Leftrightarrow Q &\triangleq (P \Rightarrow Q) \wedge (Q \Rightarrow P) && (\triangleq_{\Leftrightarrow}) \end{aligned}$$

4.6. FIRST-ORDER PREDICATE CALCULUS $(FoPCe)$

Expression: An ***expression*** is a formal statement denoting a mathematical object.

Example *20:*

$$\underbrace{f(x)}_{\text{expression}} \qquad \underbrace{x}_{\text{expression}} \qquad \underbrace{g(x, f(x))}_{\text{expression}}$$

An important distinction between expressions and predicates is that there is no concept of proof for an expression, as there is for predicates. One cannot prove an expression.

Variable: A ***variable*** is an identifier that denotes an expression.

The identifier x is a variable in the expression $f(x)$ and in the predicate $x = f(x)$.

Syntax of $basicFoPCe$***:***

```
<P> ::= ... | ∀x.<P> | <E> = <E> | R(<E>)
<E> ::= x | f(<E>)
```

with binding strength $=$, then $\forall$

4.6.1. Proof rules for basicFoPCe

$$\frac{\mathrm{H} \vdash P}{\mathrm{H} \vdash \forall x.P}\,\forall goal\left(x \widehat{\text{nfin}}\ \mathrm{H}\right) \qquad \frac{\mathrm{H}, \forall x.P, [x := E]P \vdash Q}{\mathrm{H}, \forall x.P \vdash Q}\,\forall\, hyp$$

$$\frac{}{\mathrm{H} \vdash E = E} = goal \qquad \frac{\mathrm{H}, E = F \vdash [x := F]P}{\mathrm{H}, E = F \vdash [x := E]P} = hyp$$

Substitution: The predicate $[x := E]P$ denotes the syntactic operator for ***substitution*** $[x := E]$, applied to the predicate P.

Non-freeness: The side condition $\left(x \widehat{\text{nfin}}\ \mathrm{H}\right)$ asserts that the variable x is not free in any of the predicates contained in H: the "hat" over the nfin operator denotes that it has been lifted from operating on single predicates to a set of predicates.

4.6.2. Derived logical operators for FoPCe

$$\exists x.P \triangleq \neg\forall x.\neg P \quad (\triangleq_{\exists})$$

4.7. EQUATIONAL REASONING

In this course the '==' symbol is used instead of '=' to be more consistent with the notation of equality used in Haskell.

***Example** 21:*

```
∀x. add Zero x == x                                  (addZero)
∀x. x == x                                           ((==)_refl)

  add Zero (add y z) == add (add Zero y) z
== { applying addZero }
  add y z == add (add Zero y) z
== { applying addZero }
  add y z == add y z
== { applying (==)_refl }
  True
```

4.8. INDUCTION

$\approx$ recursion

Structure:
– Proposition (Required to prove – RTP)
– Proof
  1) Base case
  2) Induction step
– Conclusion

***Example** 22:* `length (xs ++ ys) == length xs + length ys`

Given properties

`length [] = 0`	$(\text{length}_{[]})$
`∀x xs. length (x:xs) = 1 + length xs`	$(\text{length}_{(:)})$
`∀xs. [] ++ xs = xs`	$((++)_{[]})$
`∀x xs ys. (x:xs) ++ ys = x:(xs++ys)`	$((++)_{(:)})$
`∀n. 0 + n = n`	$((+)_{0})$
`∀a b c. (a + b) + c = a + (b + c)`	$((+)_{\text{assoc}})$
`∀x. (x = x) = True`	$((=)_{\text{refl}})$

Required to Prove (RTP)

`∀xs ys. (length (xs ++ ys) = length xs + length ys)`

Proof

Proceed by induction on `xs`:

Let `P xs = ∀ys. (length (xs ++ ys) = length xs + length ys)` and apply the induction rule for lists on `P xs`.

1) Base Case. RTP: `P []`

```
    P []
= {applying definiton of P, choosing a fixed but arbitrary ys}
    length ([] ++ ys) = length [] + length ys
= {applying length[] }
    length ([] ++ ys) = 0 + length ys
= {applying (++)[] }
    length ys = 0 + length ys
= {applying (+)0 }
    length ys = length ys
= {applying (=)refl }
    True
```

2) Induction Step. RTP: `∀x xs.(P xs ⇒ P (x:xs))`

Choose a fixed but arbitrary `x` and `xs`, and assume that following the induction hypothesis `P xs` holds

```
  ∀ys. (length (xs ++ ys) = length xs + length ys) (Induction Hypothesis)

  P (x:xs)
= {applying definiton of P, choosing a fixed but arbitrary ys}
  length ((x:xs) ++ ys) = length (x:xs) + length ys
= {applying length(:) }
  length ((x:xs) ++ ys) = (1 + length xs) + length ys
= {applying (++)(:) }
```

```
  length (x:(xs ++ ys)) == (1 + length xs) + length ys
== {applying length_(:) }
  1 + length (xs ++ ys) == (1 + length xs) + length ys
== {applying Induction Hypothesis}
  1 + (length xs + length ys) == (1 + length xs) + length ys
== {applying (+)_assoc }
  1 + (length xs + length ys) == 1 + (length xs + length ys)
== {applying (==)_refl }
  True
```

Conclusion

Concatenating two lists results in a list of length equal to the sum of the concatenated lists

4.8.1. Logical inference

Logical inference rule / proof rule syntax:

$$\frac{[\text{Base Case}] \quad \overbrace{([\text{Induction Hypothesis}] \Rightarrow [\text{Induction Step}])}^{\text{Inductive Case}}}{[\text{Main goal to be proven}]}$$

***Example** 23:* $\sum_{k=0}^{n} k = {}^{k(k+1)}/_{2}$

- Base case: $0 = {}^{0(0+1)}/_{2}$
- Induction step: Show that for every $k \geq 0$, if $P(k)$ holds, then $P(k+1)$ also holds

$$\begin{aligned}\frac{k(k+1)}{2} + (k+1) &= \frac{k(k+1) + 2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2} \\ &= \frac{(k+1)((k+1)+1)}{2} = \sum_{k=0}^{n} k + (k+1)\end{aligned}$$

Logical inference rule:

$$\frac{P0 \quad \forall n.\ (Pn \Rightarrow P(n+1))}{\forall n.\ (Pn)}$$

$$Pn = \left(\sum_{k=0}^{n} k = \frac{k(k+1)}{2}\right)$$

***Example** 24:* `data [a] = [] | a:[a]`

$$\frac{P\ \texttt{[]} \quad \forall\ \texttt{x}\ \ \texttt{xs}\ .\ (P\ \texttt{xs} \Rightarrow P(\texttt{x:xs}))}{\forall\ \texttt{xs}\ .\ (P\ \texttt{xs})}$$

5. SOME CURSED TYPESCRIPT IG

```ts
type And<A extends boolean, B extends boolean> = A extends true ? B : false;
type Or<A extends boolean, B extends boolean> = A extends true ? true : B;

type Num = { prev?: Num };
type Zero = Num & { prev: undefined };

type Next<N extends Num> = Num & { prev: N };
type Prev<N extends Num> = N extends Zero ? Zero : Num & N["prev"];

type One = Next<Zero>;
type Two = Next<One>;
type Three = Next<Two>;

type Add<A extends Num, B extends Num> = A extends Zero
  ? B
  : A extends Zero
    ? B
    : Add<Prev<A>, Next<B>>;
type Sub<A extends Num, B extends Num> = B extends Zero
  ? A
  : A extends Zero
    ? Zero
    : Sub<Prev<A>, Prev<B>>;

type ToTuple<N extends Num, Acc extends any[] = []> = N extends {
  prev: infer P extends Num;
}
  ? ToTuple<P, [...Acc, "_"]>
  : Acc;
type ToLiteral<N extends Num> = ToTuple<N>["length"];

type ToLiteralTest = ToLiteral<Three>;     // 3
type AddTest = ToLiteral<Add<Three, Two>>; // 5
type SubTest = ToLiteral<Sub<Three, Two>>; // 1
```