# SC: Sequent Calculus

|  |  |  |
| --- | --- | --- |
| A ⊢ B | Turnstile symbol (entails, daraus folgt, yields, liefert, therefore) | |
| A ⇒ B | If A then B | |
| Sequent | Statement that we want to prove | |
| Proof | Tree of proof rules, whose root node contains the sequent we want to prove | |
| Proof Rule | Antecedent part (list of ordered sequences) + consequent part (single) 🡪 | |
| Die Proof Rule liefert einen Beweis für das Consequent, sobald alle Antecedent bewiesen sind. | |
| Theory | Set of proof rules | |
| Complete | A proof is said to be complete if it has no pending sub-goals | |
| Valid | A sequent is valid if there exists a complete proof of the sequence | |
| Axiom | Proof rule ohne Antecedents | |
| **/Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 15.48.39.png** | | **/Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 15.49.25.png** |

# PC: Propositional Calculus (Aussagelogik)

|  |  |  |
| --- | --- | --- |
| Predicate | A property we wish to prove. Predicates have a truth value | |
| Sequent | Finite set of hypotheses predicates and a single goal predicate H ⊢ G | |
| Bei Wahl zwischen Goal und Hyp, wenn immer möglich Hyp wählen → weniger Schreibarbeit! | |
| Syntactic Equivalence | Syntactic rewriting 🡪 [⊥](https://en.wikipedia.org/wiki/%E2%8A%A5) ≙ ¬[⊤](https://en.wikipedia.org/wiki/%E2%8A%A4) | |
| Well Formed | Parse Tree kann erstellt werden, welcher nicht mehrdeutig ist (parsable) 🡪 Präzedenz! | |
| Präzedenz | Steigende Präzedenz 🡪 Von links nachs rechts (false bindet am stärksten) | |
| Grammar | **P (Pred) ::=** [**⊥**](https://en.wikipedia.org/wiki/%E2%8A%A5)(false) |[**⊤**](https://en.wikipedia.org/wiki/%E2%8A%A4)(true) |**¬P** (not) |**P ∧ P** (and) | **P ∨ P** (or)|**P ⇒ P** | **P ⇔ P** | |
| Vorgehen | 1. Well formed? 🡪 Parse Tree gemäss Grammar  2. Proof validity of sequents? 🡪 Apply proof rules 3. Proof validity of proof rule schemas?   🡪 Variante 1: Wie kommt man von der Consequence zur Antecetant?  🡪 Variante 2: Consequence und Antecetant separat beweisen.  3.1 Proof Rule vorhanden?  3.2 Syntactic Equivalence?  3.3 CUT 🡪 P entspricht dem gesuchtem GOAL | |
| H = Metavariable for multiple Predicated | | P,Q = Metavariables for single predicate |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 16.15.04.png | |  |

# FoPCe: First Order Predicate Calculus with equality (Prädikatenlogik)

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | **Bound** = placeholder within a scope (like parameter) **Free** = global variable | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 16.31.55.png | | |
| Expression | Mathematical object (cannot be proven) 🡪 f(x) | | |
| Predicate | Has truth value 🡪 x = f(x) | | |
| Replacement  NFIN = Not Free In | **/Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 16.36.42.png** | | |
| Grammar | **P (Pred) ::=** [**⊥**](https://en.wikipedia.org/wiki/%E2%8A%A5)(false) |[**⊤**](https://en.wikipedia.org/wiki/%E2%8A%A4)(true) |**¬P** (not) |**P ∧ P** (and) | **P ∨ P** (or)|**P ⇒ P** | **P ⇔ P** |  **∀x.P** (for-all) | **∃x.P** (there-is) | **E = E** | **R()** **E** **::= x** | **f()** | | |
| **x** = variable (x,y,z) **R** = relation predicate symbol (A,B,C,D,...)  **f** = function (a,b,c,f,g,h,…) = list of expressions (may be empty) | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 16.46.48.png | | | **+ Alle PC Rules** |
| Example | | H(x) : x is human (a man). M(x) : x is mortal.  s : Sokrates. | **../../../../../Desktop/Bildschirmfoto%202017-09-26%20um%2020.2** |
| ../../../../../Desktop/Bildschirmfoto%202017-09-26%20um%2020.2 | |

# Prolog

|  |  |  |  |
| --- | --- | --- | --- |
| Fact | color(red). color(green). | | |
| Rule | coloring(A, B) :- color(A), color(B), A \= B. RULE HEAD und GOALS (BODY) | | |
| Clause | facts + rules | | |
| Atoms (const) | **Lowercase**, no spaces (unless ‘ ’) 🡪 butch, ac\_dc, ’Vincent’ , ; : . :- | | |
| Numbers (const) | 12, 42, -34, 0.45, | | |
| Variables | **Uppercase or \_**, no spaces, no parenthesis 🡪 X, Y, Varia, \_tag | | |
| Complex terms | **Lowercase**, Functor: start with the predicate 🡪 loves(vincent, X). | | |
| Arity | Anzahl Parameter 🡪 loves**/**2 | | |
| Unification | Two terms unify, if they contain variables that can be instantiated  (or equal atoms e.g 'egg' = egg or beacon = beacon) | | |
| Infinite Terms | father(X) = X. Können mit **Occurs Checks** unterbunden werden. | | |
| Syntax | **Implication**= A :- B  Conjunction(**AND**) = A , B  Disjunction(**OR**) = A ; B  **Wildcard** = \_ | | |
| Search Rules | Knowledge Base is searched from top to bottom! Clauses are processed from left to right! | | |
| Search Trees | **/Users/Michi/Desktop/Bildschirmfoto 2017-12-27 um 18.09.45.png** | | |
| **Recusion** | | | |
| Recipe | 1. Write down Signature i.e. functor with arity 2. Choose recursion param 3. Write down the case distinctions for the recusion parameter (base case and recurisve case) 4. Each recursive case should make the input parameter smaller 5. Complete rule | | |
| Der rekursive Aufruf sollte wenn möglich am Ende der Clause stehen! | | |
| Recursion | len([], 0).  len([\_|T], L) :- len(T,X), L is X + 1. | | ?- len([a,b,c,d,e,[a,x],t], X ).  🡪 X=7 |
| Tail Recursion  (shorter search tree) | acclen([], Acc, Length) :- Length = Acc.  acclen([\_|L], OldAcc, Length) : -  NewAcc is OldAcc + 1,  acclen(L, NewAcc, Length). | | ?- acclen([a,b,[c],d], 0, Len).  🡪 Len = 4. |
| Cut / Fail (Control Backtracking)  Order is important! | If we don't know something, we assume that it's false | | |
| When Prolog fails, it tries to backtrack 🡪 Force backtracking | | |
| Cut !/0 blocks backtracking and fail/0 does immediately fail 🡪 returns false | | |
| neg(Goal) :- Goal, !, fail.  neg(Goal). | will succeed if Goal does not succeed | |
| enjoys(vincent,X) :- burger(X), neg(big\_kahuna\_burger(X)). | | |
| Negation as Failure +\ | enjoys(vincent,X) :- burger(X), \+ big\_kahuna\_burger(X). | | |
| **Lists** | | | |
| Head und Tail | ../../../../../Desktop/Bildschirmfoto%202017-10-16%20um%2009.4 | | |
| Anonymous Variables | ?- [X1, X2| Tail] = [mia, vincent, marsellus, jody]  🡪 X1 = mia, X2 = vincent, Tail = [marsellus, jody] | | |
| FindAll  (always succeeds) | findall(O,G,L). 🡪 produces a list L of all the objects O that satisfy the goal G | | |
| ?- findall(X,descend(martha,X),L).  🡪 L = [charlotte, caroline, laura, rose]. | | |
| **Arithmetik** | | | |
| 3 + 2 🡪 +(3,2)  X is 3 + 2 🡪 is(X,+(3,2)) | **../../../../../Desktop/arithemtic.p** | **../../../../../Desktop/Bildschirmfoto%202017-10-23%20um%2009.2** | |

# Lambda Calculus

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Lambda Calculus | LC is the functional equivalnet of the state-based Turing machine!  🡪 functions are black boxes, with no internal state! | | | |
| Components | 1. variables (..., x,y,z oder auch Zahlen)  2. Building functions (λ abstractions)  3. A way of applying functions ((λ x. x + 1) 5) 🡪 only one variable at a time | | | |
| Grammar | **P ::= M = M M ::= x** | **λx.M** | **M M** | | | |
| Application binds tighter than abstraction. 🡪 λ x. M1 M2 represents λ x. (M1 M2) Application is left associative. 🡪 x λ y . x y represents ((x (λ y . x )) y)  Choose M1 as big as possible‼ | | | |
|  | | | | |
| Comparison | **Math Syntax** | | | **λ -term Syntax** |
| f(x) | | | f x |
| sin(x) | | | sin x |
| f(g(x)) | | | f(g x) |
| f(x,y) | | | f x y |
| x+y | | | + x y |
| x^y | | | power x y |
| Normal Form | A λ-term is in normal form if no further reductions can be applied. | | | |
| Confluence | Language property 🡪 There are no restrictions to execution order. Every λ-term has at most one normal form | | | |
| Currying | Allows us to define functions with more than one argument. f x y 🡪 (f x) y | | | |
| Higher Order Functions | Functions that accept functions as input or return functions as output /Users/Michi/Desktop/Bildschirmfoto 2017-12-28 um 10.27.26.png | | | |
| **Evaluation Strategies 🡪** Order in which redexes (Reducable Expresssion) are reduced | | | | |
| Vorgehen | 1. Definiere Kandiaten 2. Wähle Innermost rsp. Outermost Candiate 3. Gibt es zwei Redex auf der gleichen Stufe, wähle den linken von beiden. | | | |
| β Reduction | Application of a function parameter | | | |
| δ Reduction | Substitution of a defined symbol with its definition | | | |
| **Innermost First** (applicative order) 🡪 not always shorter!  *Call by value*: The arguments are reduced exactly once! | Function arguments are substituted into the body of a function after they are reduced.   A redex is innermost, if there is no other redex inside it. | | ../../../../../Desktop/Bildschirmfoto%202017-11-06%20um%2009.0DELTA REDUCTION FIRST! | |
| **Outermost First** (normal order) 🡪 better termination properties!  *Call by name*: Arguments are reduced as often as needed | Function arguments are substituted into the body of a function before they are reduced.   A redex is outermost, if there is no other redex outside it. | | ../../../../../Desktop/Bildschirmfoto%202017-11-06%20um%2009.1 | |
| **Church Encodings** | | | | |
|  |  |  | | |

# Haskell

|  |  |  |  |
| --- | --- | --- | --- |
| Syntax | Functions and arguments must begin lowercase!  Indentation makes {} redundant (layout rule). | | |
| **Operator** | | | |
| Operator Selection | Operatoren können verschieden geschrieben werden:  1 + 2 ⇔ (+) 1 2 ⇔ (1+) 2 ⇔ (+2) 1 = 3 | | |
| Operators | + - \* `div` `mod` ^ | && || < > <= >= == /= | |
| Function composition operator (DOT) | (.) :: (b -> c) -> (a -> b) -> (a -> c)  f . g = \ x -> f (g x) | | |
| Function application | ($) :: (a -> b) -> a -> b | | |
| **Functions** | | | |
| Recursive Function | qsort :: Ord a => [a] -> [a]  qsort [] = []  qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger  where  smaller = [a | a <- xs, a <= x]  larger = [b | b <- xs, b > x] | | **1. Define the type!!!** 2. Enumerate cases 3. Define base case 4. Define other cases 5. Generalize and simplify |
| Conditional expression | abs :: Int -> Int  abs n = if n >= 0 then n else –n | | |
| Guarded expression | abs :: Int -> Int  abs n  | n >= 0 = n  | otherwise = -n | | |
| Pattern Matching  (Top Down) | not :: Bool -> Bool  not False = True  not True = False | | |
| Lambda expression | add :: Int -> (Int -> (Int -> Int)) add = \ x y -> x + y) | | |
| Common Functions | head [0,1,2,3,4] | 0 | |
| tail [0,1,2,3,4] | [1,2,3,4] | |
| init [0,1,2,3,4] | [0,1,2,3] | |
| sum [0,1,2,3,4] | 10 | |
| product [1,2,3,4] | 24 | |
| take 3 [0,1,2,3,4] | [0,1,2] | |
| drop 3 [0,1,2,3,4] | [3,4] | |
| length [0,1,2,3,4] | 5 | |
| splitAt 3 [0,1,2,3,4] | ( [0,1,2] , [3,4] ) | |
| reverse [0,1,2,3,4] | [4,3,2,1,0] | |
| concat [[1, 2], [3], [0, 4]] | [1,2,3,0,4] | |
| zip ['a', 'b', 'c'] [1, 2, 3, 4] | [('a',1),('b',2),('c',3)] | |
| map (+1) [0,1,2,3,4] | [1,2,3,4,5] | |
| filter even [1 .. 10] | [2,4,6,8,10] | |
| foldr (+) 5 [1,2,3,4] | 15 | |
| foldl (/) 64 [4,2,4] | 2 | |
| id 3 | 3 | |
| const 3 2 | 3 | |
| [1,2,3,4,5] !! 2 (zero indexed) | 3 | |
| [1,2,3] ++ [4,5] | [1,2,3,4,5] | |
| 1 : (2 : (3 : (4 : []))) | [1, 2, 3, 4] | |
| **Higher Order Function** | | | |
| Map | map :: (a -> b) -> [a] -> [b]  map f xs = [f x | x <- xs] | Applies a function to every element of a list (works only on lists) | |
| Filter | filter :: (a -> Bool) -> [a] -> [a]  filter p xs = [x | x <- xs, p x] | Selects every element from a list that satisfies a predicate. | |
| Foldr | foldr :: (a -> b -> b) -> b -> [a] -> b  foldr f v [] = v  foldr f v (x : xs) = f x (foldr f v xs) | 1. Take the rightmost item of the list and the second argument and apply *f*. 2. Take the penultimate right item of the list and the previous result and apply *f*. | |
| Foldl | Foldl :: (a -> b -> a) -> a -> [b] -> a foldl f v [] = v  foldl f v (x : xs) = foldl f (f z x) xs | 1. Take the leftmost item of the list and the second argument and apply *f*. 2. Take the penultimate left item of the list and the previous result and apply *f*. | |
| **Lists** | | | |
| Head and Tail | myFunction (n : ns) | | |
| Multiple Heads | myFunction (n1 : \_ : n3 : n4 : ns) | | |
| **List comprehension** | | | |
| List comprehension | [x ^ 2 | x <- [1 .. 5], x `mod` 2 == 0 ]  🡪 [Collector | var <- Generator/Function/Enumeration, Guard] 🡪 Read as: [x ^2 for x in [1 .. 5] if x `mod` 2 == 0] | | |
| Dependent generator | [(x, y) | x <- [1 .. 3], y <- [x .. 3]] [(1,1),(1,2),(1,3), (2,2),(2,3), (3,3)] | Like nested loops: (otherwise use zip) x = Outer, y = Inner, (z= Innermost,...) | |
| Infinite generator | [0..] | | |
| String comprehension | Strings are just lists of characters and therefore the same functions can be applied 🡪 length "abcde" zip "abc" [1, 2, 3, 4] | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | | | |
| Basic Types | - Bool - Char, String - Int = fixed precision - Integer = arbitrary precision - Float, Double - [t] = list type,  - (t1, t2) = tuple type,  - (t1 -> t2) = function type | | myFunction :: Num a => a -> a myFunction :: [a] -> a myFunction :: a -> b -> (a,b) myFunction :: (a -> a) -> a -> a |
| Type declaration  **(Synonyms)** | type Pos = (Int, Int) type Trans = Pos -> Pos | Cannot be recursive 🡪 type Tree = (Int, [Tree]) Es sind auch Parameter möglich  🡪 type Pair a = (a, a) 🡪 myFunc :: Pair Int -> Int | |
| Data declaration  **(Completely new types)** | data Bool = False | True myFunc :: Bool 🡪 Bool | - The values False and True are constructors and must therefore start with an uppercase letter! - Can be recursive - Can be used like numbers 🡪 [A,A,B] - Type constructors can have parameters (kinds) | |
| Custom types and conventional types | data Expr = Val Int | Add Expr Expr | Mul Expr Expr | | |
| size :: Expr -> Int  size (Val n) = 1  size (Add x y) = size x + size y  size (Mul x y) = size x + size y | | |
| Derive | Generate default implementation of build-in type classes automatically | | |
| data Bool = False | True deriving (Eq, Ord, Show, Read) | | |
| **Type Classes** | | | |
| Class declaration  **(Interface which described the behaviour of a type)** | Haskell uses type classes to achieve ad-hoc polymorphism | | |
| Num = types which have a numeric property (Parent of Integral and Floating) Eq = types that support equality testing Ord = types that have an ordering Show = types that can be presented as strings Read = opposite of show, takes a string and returns a type which is a member of Read. Enum = types that are sequentially ordered Bounded = types that have an upper and lower bound | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-28 um 18.32.54.png | | |
| Extend Class  **Context inheritance from class operator (=>)** | /Users/Michi/Desktop/Bildschirmfoto 2017-12-28 um 18.33.20.png | | |
| Instance | instance [base] [child] where .. | | |
| instance Functor [] where fmap = map | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-28 um 18.34.34.png | | |

|  |  |  |
| --- | --- | --- |
| **IO / Interaction** | | |
| Pure functions | Haskell Programs do not have side effects. (pure) However interaction needs side effects. Therefore encode side effects in types. | |
| IO Type | Takes anything, and returns this anything with some return type a type IO Char 🡪 Returns a character  /Users/Michi/Desktop/Bildschirmfoto 2017-12-28 um 18.44.16.png | |
| IO Empty Value **()** | Empty value in IO () type 🡪 No return / result value, for example print | |
| getChar :: IO Char | reads a character from the keyboard, echoes it to the screen, and returns the character as its result value 🡪 getChar | |
| putChar :: Char -> IO () | writes the character c to the screen, and returns no result value 🡪 putChar c | |
| return :: a -> IO a | simply returns the value v, without performing any interaction 🡪 return v | |
| Sequencing  Combine multiple actions with ’do’ | getLine :: IO String  getLine = do  x <- getChar  if x == '\n'   then % only one subsequent action -> no do  return []  else  do % two subsequent actions  xs <- getLine  return (x : xs) | |
| strlen :: IO ()  strlen = do putStr "Enter a string:"  xs <- getLine  putStr "The string has"   putStr (show (length xs))   putStrLn " characters" | |
| **Functors** | | |
| Problem | When a value is wrapped in a context, you can't apply a normal function to it  🡪 (+3) Just 2 | |
| Solution | Functors are types that wrap values of other types and **allow us to map functions over the wrapped values.** | |
| We can only map unary functions over them! | |
| Typeclass | class Functor f where  fmap :: (a -> b) -> f a -> f b | |
| Fmap / <$> | Fmap unwraps the value from the context, applies the function, and rewraps the value in the context 🡪 fmap (+3) (Just 2) 🡪 Just 5 | |
| <$> is the infix version of fmap 🡪 fmap f1 f2 == f1 <$> f2 | |
| **Applicative Functor** | | |
| Problem | When a function is wrapped in a context 🡪 apply functions wrapped in a context to a value wrapped in a context. | |
| Solution | **Apply functions to Functors arguments** 🡪pure (+) <\*> (Just 11) <\*> (Just 31) | |
| We can also map n-ary functions over them! | |
| Typeclass | class Functor f => Applicative f where   pure :: a -> f a  (<\*>) :: f (a -> b) -> f a -> f b | |
| Pure | Generic function to wrap values 🡪 pure :: a -> f a | |
| **Monads** (A new way of thinking about programming with effects) | | |
| Solution | **Apply and compose “effectful” functions with “effectful” arguments** | |
| Monads apply a function that returns a wrapped value to wrapped value  🡪 Just 3 >>= Function that returns a Maybe | |
| Typeclass | class Applicative m => Monad m where   return :: a -> m a  (>>=) :: m a -> (a -> m b) -> m b  return = pure | |
| Bind operator (sequencing) | Bind takes a monad and a function that returns a monad and it returns a monad (variable substitution) | |
| >>= :: m a -> (a -> m b) -> m b | |
| Use do Notation instead of Bind | eval :: Expr -> Maybe Int eval (Val n) = return n  eval (Div l r) = do  x <- eval l   y <- eval r  safediv x y | eval :: Expr -> Maybe Int eval (Val n) = Just n  eval (Div l r) = eval l >>= \x ->  eval r >>= \y ->   safediv x y |
| **Maybe Monad** | | |
| Maybe | The Maybe monad returns different results depending on the context  🡪 (Just / Nothing) | |
| data Maybe a = Nothing | Just a | |
| Instance Functor | instance Functor Maybe where   fmap \_ Nothing = Nothing   fmap f (Just x) = Just (f x) | |
| Instance Applicative | instance Applicative Maybe where   pure = Just  Nothing <\*> \_ = Nothing   (Just f) <\*> e = fmap f e | |
| Instance Maybe | instance Monad Maybe where  Nothing >>= \_ = Nothing  (Just x) >>= f = f x | |