Putting the Λ in HaskeΛΛ

Floris Westerman

June 5, 2023

Abstract

We provide an implementation of various λ-calculi in Haskell. We define a typeclass for such calculi, and then implement untyped, simply-typed, and polymorphic versions. We focus on developer-friendly and accessible syntax, so that a developer can use the library to work with λ-calculus in their own code. Because of the developer focus, we do not provide a parser or any other "user-friendly" tools, except for some pretty printing. Lastly, we include some sample λ-terms for each implemented calculus.

Contents

1 Typeclasses

We start our work by defining some useful typeclasses to represent generic substitutable terms, generic λ-calculi, and a typability extension to the generic `ΛCalculus`-typeclass.

```
{-# OPTIONS_GHC -Wno-orphans #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE InstanceSigs #-}

module Lambda (
    Substitutable (
        freeVariables, renameVariable,
        prepareSubstitution, substitute, performSubstitution
    ),
    ΛCalculus (
        Variable, VariableName,
        fromVar, fromVarName, fromΛ, fromApp,
        prettyΛ, prettyLambda, showΛ, showLambda,
        αEquiv, βReductions, betaReductions,
        isNormalForm, normalForm, (===)
    ),
    EquivalenceContext,
    TypedΛCalculus (
        Type,
        prettyType, showType, showTermType,
        typeOf, typesEquivalent,
        deduceTypes, hasValidType
    ),
    TypeMapping
) where

import Data.Set(Set, toList)
```

This is a simple module header with some language features and imports - nothing special yet.

```
class Substitutable term var | term -> var where
    freeVariables :: term -> Set var
    renameVariable :: term -> var -> var -> term
    prepareSubstitution :: term -> var -> term
    performSubstitution :: term -> var -> term -> Maybe term
    substitute :: term -> var -> term -> Maybe term
    substitute term var new = performSubstitution prepared var new
        where prepared = foldr (flip prepareSubstitution) term $ toList $ freeVariables new
```

Here we define a basic `Substitutable` typeclass. It is intended to represent any terms with variables on which a substitution can be performed. This will later be re-used for both λ-terms as well as type expressions in the polymorphic calculus. The name of the function `freeVariables` is slightly influenced by the intended usage context, it is supposed to be the list of 'eligible substitution targets'.

The implementation of the substitution itself is split in three parts: analysis, preparation, and substitution itself. The intention is to prevent accidental name clashes after substitution, leading to potentially unwanted variable binding. For example, when β-reducing $(\lambda xy.x)y \to_\beta (\lambda y.x)[x := y]$ we want to prevent simply replacing x by y, since then suddenly the binding of y changes from a free variable to bound by our λ. Instead, we want to rename our bound variable before substituting: $(\lambda y.x)[x := y] = (\lambda y'.x)[x := y] \to_\beta \lambda y'.y$. This is exactly what is done in `prepareSubstitution` for every potentially conflicting ('free') variable in the substitution target.

```haskell
class ΛCalculus λ where
    type Variable λ
    type VariableName λ
    type VariableName λ = String

    -- Some kind of constructors
    fromVar     :: Variable λ -> λ
    fromVarName :: VariableName λ -> λ
    fromΛ       :: Variable λ -> λ -> λ
    fromApp     :: λ -> λ -> λ

    -- Pretty printing intended just for the end user, including some
    -- equivalent show functions that will print to IO, taking care of
    -- unicode properly (Show on a unicode string will not print the
    -- unicode characters properly)
    prettyΛ, prettyLambda :: λ -> String
    prettyLambda = prettyΛ
    showΛ, showLambda :: λ -> IO ()
    showLambda = showΛ
    showΛ = putStrLn . prettyΛ

    αEquiv :: λ -> λ -> EquivalenceContext λ -> Bool

    βReductions, betaReductions :: λ -> [λ]
    betaReductions = βReductions

    isNormalForm :: λ -> Bool
    isNormalForm = null . βReductions

    -- If there is a normal form, then it can be achieved with repeated
    -- contraction of the leftmost redex.
    βReduceLeft :: λ -> λ
    βReduceLeft term
     | isNormalForm term = error "The λ-term is already in normal form"
     | otherwise         = head $ βReductions term

    normalForm :: λ -> λ
    normalForm term
     | isNormalForm term = term
     | otherwise         = (normalForm . βReduceLeft) term

    -- β-equivalence relation that we will identify with ===
    (===) :: λ -> λ -> Bool
    x === y = x == y || normalForm x == normalForm y
    infix 1 ===

type EquivalenceContext λ = [(VariableName λ, VariableName λ)]
```

Here we have defined the type class for a generic λ-calculus. We define two type families, `Variable λ` and `VariableName λ` that will have an instance for each instance of the type class. For the variable name, we provide a default instance with `String`. We define some default constructors that will be relevant for all calculi, and define and implement some pretty printing logic. The pretty printing is not intended to generate valid Haskell code, but should instead print something more human-readable. We also implement some show functions that will print directly to `IO`, as otherwise the unicode characters in the text will be encoded by `Show`.

Afterwards, we define the 'meat' of the any calculus: a notion of α-equivalence and β-reduction. We define a termination criterion for finding the β-normal form, as well as a rudimentary normal form finding that will simply only contract the leftmost redex until it reaches termination. For non-strongly normalising calculi, this function might not terminate. Lastly, we define `(===)` to be β-equivalence.

```
instance {-# OVERLAPPABLE #-} (ΛCalculus λ) ⇒ Eq λ where
    (==) :: λ -> λ -> Bool
    x == y = αEquiv x y []
```

For each instance of our `ΛCalculus` type class, we provide an instance of `Eq` that will identify two λ-terms when they are α-equivalent. This provides us with a versatile notion of equality that is intuitive to most users. Together with the previously-defined `(===)` operation, we cover most bases. Note that this instance is labelled `{-# OVERLAPPABLE #-}`. This indicates to GHC that even though this instance might overlap with other `Eq` instances, the other instance should be picked with priority. This prevents issues with the open-world assumption in GHC, where an instance `ΛCalculus Int` *might* exist, conflicting with the 'traditional' instance for `Int`.

```
class (ΛCalculus λ) ⇒ TypedΛCalculus λ where
    data Type λ

    prettyType :: Type λ -> String

    showType :: Type λ -> IO ()
    showType = putStrLn . prettyType

    showTermType :: λ -> IO ()
    showTermType term = putStrLn $ maybe "Impossible type" prettyType (typeOf term)

    typesEquivalent :: Type λ -> Type λ -> EquivalenceContext λ -> Bool
    typeOf :: λ -> Maybe (Type λ)

    deduceTypes  :: λ -> TypeMapping λ -> λ
    hasValidType :: λ -> TypeMapping λ -> Bool

type TypeMapping λ = Set (VariableName λ, Type λ)

instance {-# OVERLAPPABLE #-} (TypedΛCalculus λ) ⇒ Eq (Type λ) where
    (==) :: Type λ -> Type λ -> Bool
    σ == τ = typesEquivalent σ τ []
```

At the end of our typeclass adventure we define some extensions to `ΛCalculus` tailored to typed calculi. We define another type family, this time a family of `data` definitions. In contrast to `type` families, `data` families allow us to uniquely identify a single `Type λ` to each implementation of `TypedΛCalculus λ`. As a consequence, we know that if `Type λ1 == Type λ2`, then `λ1 == λ2`, which in turn enables us to define class instances for the `Type λ` - which is what we do for `Eq` again to provide a type equivalence.

2 Untyped Λ-calculus

We will now discuss the implementation of a basic untyped λ-calculus. We will implement the standard type class we defined before, and we will focus on developer-friendliness in the syntax. From now on, we will skip module headers for brevity.

```haskell
-- Main defintions of lambda terms
data Λ = Var (Variable Λ) | Λ (Variable Λ) Λ | App Λ Λ
    deriving (Show)
type Lambda = Λ
```

Here we define our main `Λ` data type. It will be either a variable, an application, or a λ-abstraction. We do not derive `Eq`, since that will be provided by the `ΛCalculus` typeclass, to make `Λ` an equivalence class under α-equivalence.

```haskell
instance Substitutable Λ String where
    -- Determining the set of free variables
    freeVariables :: Λ -> Set (Variable Λ)
    freeVariables (Var x)      = singleton x
    freeVariables (Λ x term) = delete x $ freeVariables term
    freeVariables (App x y)  = freeVariables x `union` freeVariables y

    -- Performing a substitution
    renameVariable :: Λ -> VariableName Λ -> VariableName Λ -> Λ
    renameVariable (Var x) old new
        | x == old  = Var new
        | otherwise = Var x
    renameVariable (Λ x term) old new
        | x == old  = Λ new $ renameVariable term old new
        | otherwise = Λ x   $ renameVariable term old new
    renameVariable (App x y) old new = App (renameVariable x old new) (renameVariable y old new)

    prepareSubstitution :: Λ -> VariableName Λ -> Λ
    prepareSubstitution (Λ x term) var
        | x /= var  = Λ x       $ prepareSubstitution term var
        | otherwise = Λ newName $ prepareSubstitution (renameVariable term x newName) var
        where newName = "_" ++ x
    prepareSubstitution (App x y) var = App (prepareSubstitution x var) (prepareSubstitution y var)
    prepareSubstitution var _ = var

    performSubstitution :: Λ -> Variable Λ -> Λ -> Maybe Λ
    performSubstitution (Var x) var term
        | x == var  = Just term
        | otherwise = Just $ Var x
    performSubstitution (Λ x t) var term
        | x == var  = Just $ Λ x t
        | otherwise = Λ x <$> performSubstitution t var term
    performSubstitution (App x y) var term = App <$> performSubstitution x var term <*> performSubstitution y var term
```

The first action is to implement a notion of substitution on our data type. The definitions are relatively straightforward. One point of interest is the substitution preparation: it is not perfect as it will only prepend a single underscore to the variable name, which in more advanced cases might still yield conflicts. Furthermore, the actual substitution implementation uses `Maybe`, since substitutions might fail in the generic case. For this implementation, that is not applicable.

```haskell
instance ΛCalculus Λ where
    type Variable Λ = String

    fromVar = Var
    fromVarName = Var
    fromΛ = Λ
    fromApp = App

    -- Pretty printing
    prettyΛ :: Λ -> String
    prettyΛ (Var x) = x
    prettyΛ (Λ x term@(Λ _ _))        = "λ" ++ x ++  tail (prettyΛ term)
    prettyΛ (Λ x term)                = "λ" ++ x ++ "." ++ prettyΛ term
    prettyΛ (App x@(Λ _ _) y@(Var _)) = "(" ++ prettyΛ x ++ ")"  ++ prettyΛ y
    prettyΛ (App x@(Λ _ _) y)         = "(" ++ prettyΛ x ++ ")(" ++ prettyΛ y ++ ")"
    prettyΛ (App x y@(Var _))         = prettyΛ x     ++     prettyΛ y
    prettyΛ (App x y)                 = prettyΛ x ++ "(" ++ prettyΛ y ++ ")"

    -- Defining the α-equivalence between pre-terms
    αEquiv :: Λ -> Λ -> [(Variable Λ, Variable Λ)] -> Bool
    αEquiv (Var x) (Var y) context          = x == y || (x, y) `elem` context
    αEquiv (Λ x xTerm) (Λ y yTerm) context  = notCrossBound && αEquiv xTerm yTerm ((x, y) : context)
        where
            yFreeInX = y `elem` freeVariables xTerm
            xFreeInY = x `elem` freeVariables yTerm
            notCrossBound = x == y || (not yFreeInX && not xFreeInY)
    αEquiv (App x1 x2) (App y1 y2) context  = αEquiv x1 y1 context && αEquiv x2 y2 context
    αEquiv _ _ _ = False

    -- Perform one of each possible β-redex in the lambda term
    βReductions :: Λ -> [Λ]
    βReductions (App (Λ x term) n) = [fromJust substitution | isJust substitution] ++ reduceTerm ++ reduceApp
        where
            reduceTerm = (\newTerm -> App (Λ x newTerm) n) <$> βReductions term
            reduceApp = App (Λ x term) <$> βReductions n
            substitution = substitute term x n
    βReductions (Var _)     = []
    βReductions (Λ x term) = Λ x <$> βReductions term
    βReductions (App x y)  = ((`App` y) <$> βReductions x) ++ (App x <$> βReductions y)
```

The implementation of our typeclass is not particularly interesting. It has some pretty printing functionality that aims to resemble mathematical notation, and implements the default α-equivalence relation. The only interesting aspect there is how to deal with variable names that are bound in one expression and free in the other.

The more complicated part is the implementation of the β-reduction itself. This function will produce a list of all possible reductions. In most cases this is a simple recursive tree operation - the interesting case is where we have an application operating on an abstraction, which is a β-redex. There, we try to reduce the term itself through a substitution (which will always succeed in an untyped setting), but we also include reductions where we do not reduce this particular redex, but recurse further down into the data structure.

2.1 Parsing

From a coding perspective, the most interesting aspect of this implementation of an untyped λ-calculus is the 'parsing' part we implement here. The objective is to develop a developer-friendly syntax for constructing λ-terms (instances of `Λ`) without the need for a parser and with the benefit of compile-time syntax checking. The objective is to move from a syntax of `Λ "x" (Λ "y" (App (Var "x") (Var "y")))` to the nicer `λ"x" "y" --> "x" $$ "y"`. We do this by using variadic functions.

```haskell
-- Helper functions for notation
class ΛParameters a where
    toΛparameters :: [Variable Λ] -> a

instance ΛParameters (Λ -> Λ) where
    toΛparameters [] = error "No Λ-parameters supplied"
    toΛparameters [x] = Λ x
    toΛparameters (x:xs) = Λ x . toΛparameters xs

instance (ΛParameters a) ⇒ ΛParameters (String -> a) where
    toΛparameters xs x = toΛparameters (xs ++ [x])

λ,l :: ΛParameters a ⇒ a
l = λ
λ = toΛparameters []
```

The idea behind the syntax introduced before is to let the arrow `(-->)` separate the two 'parts' of a λ-abstraction. The arrow will be an infix function that as its first argument accepts a 'partial λ-term' - essentially a function that will accept a λ-term serving as body, returning another λ-term representing the entire abstraction.

This 'partial λ-term' is implemented using a variadic function. Variadic functions in Haskell are implemented using recursive typeclasses - in this case `ΛParameters`. We have two instances for this typeclass, one of our desired 'return type' (`Λ -> Λ`), and one of the 'recursive case' (`String -> a` for `a` another instance of the typeclass). In addition, we have a 'seed function' `λ` that is just an arbitrary instance of the typeclass. So then, if we are writing `λ"x" "y" -->` … , GHC will deduce that everything to the left of the arrow will need to have type `(Λ -> Λ)`, and thus `λ` must be a function accepting two parameters and returning this function type. Since `λ` itself just needs to be an instance of our typeclass, GHC will apply the 'recursive' instance twice to obtain `λ :: ΛParameters [String -> (ΛParameters [String -> (ΛParameters [Λ -> Λ])])]` (square brackets added to indicate how the typeclasses combine). Note that without the presence of `(⟶)` we would need to manually set the type of the partial term - Haskell won't know whether you want the 'return type' or the 'recursive case'. A defaulting mechanism would help alleviate this ambiguity, but the presence of `(⟶)` like will be the case in practice, this is not needed.

Now that we have the typeclasses with a recursive, variadic behaviour, we can focus on the implementation of them. This is essentially a 'conversion' function from strings to functions `Λ -> Λ`, where the 'seed' will call it with an empty list, and the recursive typeclass instances will just append to this list and call the 'next implementation' - all the way until the 'return type' instance, that will convert the list of strings into a nested λ-abstraction that is just missing the 'body'.

```haskell
class ΛTerm a where
    toΛ :: a -> Λ

instance ΛTerm Λ      where toΛ = id
instance ΛTerm String where toΛ = fromVarName

(-->) :: (ΛTerm a) ⇒ (Λ -> Λ) -> a -> Λ
a --> b = a (toΛ b)
infixr 6 -->

($$) :: (ΛTerm a, ΛTerm b) ⇒ a -> b -> Λ
```

```
x $$ y = App (toΛ x) (toΛ y)
infixl 7 $$
```

This second part of parsing is a bit simpler: we now only need to construct some λ-term that is the body of our abstraction. To simplify notation, eliminating prefix calls to `App`, we define an infix function `($$)` that will do the same. To then further simplify notation, eliminating the need for `Var`, we create a typeclass representing a generic λ-term that has instances of a string and an actual `Λ` with a conversion function to always produce a `Λ`.

2.2 Example Terms

We implement some standard example terms in our untyped λ-calculus. These can be used for basic testing or exploration of the realms of λ-terms.

```
module UntypedLambdaTerms where

import UntypedLambda

-- Common lambda terms
λI, λK, λS, λΩ, λY :: Λ
λI = λ"x" --> "x"
λK = λ"x" "y" --> "x"
λS = λ"x" "y" "z" --> "x" $$ "z" $$ ("y" $$ "z")
λΩ = let λω = λ "x" --> "x" $$ "x" in λω $$ λω
λY = λ "f" --> (λ "x" --> "f" $$ ("x" $$ "x")) $$ (λ "x" --> "f" $$ ("x" $$ "x"))
```

Notably here, `λΩ` is non-normalising and always reduces to itself. Similarly, `λY` is non-normalising, but has an infinite reduction path to ever-different terms - it will never cycle back to itself. `λK` and `λS` correspond to combinators as they are defined in combinatory logic.

```
-- Boolean values
λtrue, λfalse :: Λ
λtrue = λ "x" "y" --> "x"
λfalse = λ "x" "y" --> "y"

-- Conditionals with nice "inline" syntax:
-- "if P then Q else R" ⟺ {λif P ? Q |: R} ⟺ {P $$ Q $$ R}
λif :: Λ -> (Λ -> Λ -> Λ)
λif p q r = p $$ q $$ r

(?) :: (Λ -> Λ -> Λ) -> Λ -> (Λ -> Λ)
(?) p' = p'

(|:) :: (Λ -> Λ) -> Λ -> Λ
(|:) q' = q'

-- Pairs and two pair accessors
λpair :: Λ
λpair = λ "x" "y" "f" --> "f" $$ "x" $$ "y"

λp1, λp2 :: Λ
λp1 = λ "p" --> "p" $$ (λ "x" "y" --> "x")
λp2 = λ "p" --> "p" $$ (λ "x" "y" --> "y")
```

We implement standard booleans and some syntax to more easily do a ternary if-statement; but it essentially just concatenates the three terms. In addition, we implement pairs and two accessors/deconstructors.

```haskell
-- Church numerals and various arithmetical operations
church :: Int -> Λ
church 0 = λ "f" "x" --> "x"
church n = λ "f" "x" --> fs n
    where
        fs :: Int -> Λ
        fs 1 = "f" $$ "x"
        fs m = "f" $$ fs (m - 1)

λsucc, λadd, λmult, λexp, λzero :: Λ
λsucc = λ "n" "f" "x" --> "f" $$ ("n" $$ "f" $$ "x")
λadd = λ "m" "n" "f" "x" --> "m" $$ "f" $$ ("n" $$ "f" $$ "x")
λmult = λ "m" "n" "f" "x" --> "m" $$ ("n" $$ "f") $$ "x"
λexp = λ "m" "n" "f" "x" --> "m" $$ "n" $$ "f" $$ "x" -- "other way around": λexp a b ⟺ b^a
λzero = λ "m" --> church 0

-- λit has the property that:
---- λit x y (church 0) ≡ x
---- λit x y (λsucc n) ≡ y $$ (λit x y n)
λit :: Λ
λit = λ "x" "y" "z" --> "z" $$ "y" $$ "x"

-- λiszero has the property that:
---- λiszero (church 0) ≡ λtrue
---- λiszero (λsucc n) ≡ λfalse
λiszero :: Λ
λiszero = λit $$ λtrue $$ (λ"x" --> λfalse)

-- Predecessor function
λpred :: Λ
λpred = λ "x" --> λp1 $$ (λQ $$ "x")
    where λQ = λit $$ (λpair $$ church 0 $$ church 0) $$ (λ"x" --> (λpair $$ (λp2 $$ "x") $$ (λsucc
    ↪  $$ (λp2 $$ "x"))))
```

Lastly, we implement Church numerals and various arithmetical operations in a canonical way. Only the predecessor is slightly different, taken from how it was introduced in the Type Theory course. These examples show the utility of the 'parser' functions we implemented, and how they drastically help code clarity. At the same time, we keep the analysis from GHC to ensure our syntax is correct, so we don't run into issues at runtime when it turns out we wrote an incorrect λ-term.

3 Typed Λ-calculus

Of all calculi implemented, this might be the least useful one: it is strongly constraint in expressivity due to the introduction of types, but since there is no quantification over types like in the polymorphic variant, it is not possible to write 'generic' pair functions that work for every type - instead, for every type a new implementation has to be made.

However, from a coding perspective, this is a nice intermediate step between untyped and polymorphic versions: we can focus on introducing types, while keeping them still strictly separated from terms themselves.

```haskell
data ΛVariable = (VariableName Λ) :-: (Type Λ)
    deriving (Show, Eq, Ord)
infixl 6 :-:

data Λ = Var ΛVariable | Λ ΛVariable Λ | App Λ Λ
    deriving (Show)
type Lambda = Λ
```

This is mostly the same as for the untyped case, except that variables now deserve their own dedicated data type and are no longer simply strings. A variable here is now a string with an associated type - the `Type` here is not a typeclass itself, but an instance of the type family inside the `TypedΛCalculus` typeclass. Its definition will be given further down. We also set the correct fixity of the variable constructor.

```haskell
instance Substitutable Λ String where
    renameVariable :: Λ -> VariableName Λ -> VariableName Λ -> Λ
    renameVariable (Var (x :-: σ)) old new
        | x == old  = Var (new :-: σ)
        | otherwise = Var (x   :-: σ)
    renameVariable (Λ (x :-: σ) term) old new
        | x == old  = Λ (new :-: σ) $ renameVariable term old new
        | otherwise = Λ (x   :-: σ) $ renameVariable term old new
    renameVariable (App x y) old new = App (renameVariable x old new) (renameVariable y old new)

    prepareSubstitution :: Λ -> VariableName Λ -> Λ
    prepareSubstitution (Λ (x :-: σ) term) var
        | x /= var  = Λ (x       :-: σ) $ prepareSubstitution term var
        | otherwise = Λ (newName :-: σ) $ prepareSubstitution (renameVariable term x newName) var
        where newName = "_" ++ x
    prepareSubstitution (App x y) var = App (prepareSubstitution x var) (prepareSubstitution y var)
    prepareSubstitution var _ = var

    performSubstitution :: Λ -> VariableName Λ -> Λ -> Maybe Λ
    performSubstitution (Var (x :-: σ)) var term
        | x /= var              = Just $ Var (x :-: σ)
        | typeOf term /= Just σ = Nothing
        | otherwise             = Just term
    performSubstitution (Λ (x :-: σ) t) var term
        | x /= var              = Λ (x :-: σ) <$> performSubstitution t var term
        | otherwise             = Just $ Λ (x :-: σ) t
    performSubstitution (App x y) var term = App <$> performSubstitution x var term <*> performSubstitution y var term
```

We have another implementation of `Substitutable` to start off with, where most functions are an almost direct copy of the untyped case, but with some types sprinkled in. Only actually performing a substitution has become more complicated: we have to check the types of the variable to be substituted and the target λ-term. If they are not equivalent (in the simply-typed case: identical), then we cannot perform the substitution and we return `Nothing`. Otherwise, the implementation is identical, and thus the `freeVariables` implementation has been left out for brevity.

```haskell
instance ΛCalculus Λ where
    type Variable Λ = ΛVariable

    fromVar = Var
    fromVarName name = Var (name :-: Null)
    fromΛ = Λ
    fromApp = App

    prettyΛ :: Λ -> String
    prettyΛ (Var (x :-: Null))          = x
    prettyΛ (Var (x :-: σ))             = "(" ++ x ++ ":" ++ prettyType σ ++ ")"
    prettyΛ (Λ (x :-: σ) term@(Λ _ _)) = "λ" ++ x ++ ":" ++ prettyType σ ++ "," ++ tail (prettyΛ term)
    prettyΛ (Λ (x :-: σ) term)          = "λ" ++ x ++ ":" ++ prettyType σ ++ "." ++ prettyΛ term
    prettyΛ (App x@(Λ _ _) y@(Var _))  = "(" ++ prettyΛ x ++ ")"  ++ prettyΛ y
    prettyΛ (App x@(Λ _ _) y)          = "(" ++ prettyΛ x ++ ")(" ++ prettyΛ y ++ ")"
    prettyΛ (App x y@(Var _))          = prettyΛ x     ++     prettyΛ y
    prettyΛ (App x y)                  = prettyΛ x ++ "(" ++ prettyΛ y ++ ")"

    αEquiv :: Λ -> Λ -> EquivalenceContext Λ -> Bool
    αEquiv (Var (x :-: σ)) (Var (y :-: τ)) context
        = σ == τ && (x == y || (x, y) `elem` context)

    αEquiv (Λ (x :-: σ) xTerm) (Λ (y :-: τ) yTerm) context
        = notCrossBound && σ == τ && αEquiv xTerm yTerm ((x, y) : context)
        where
            yFreeInX = y `elem` freeVariables xTerm
            xFreeInY = x `elem` freeVariables yTerm
            notCrossBound = x == y || (not yFreeInX && not xFreeInY)

    αEquiv (App x1 x2) (App y1 y2) context = αEquiv x1 y1 context && αEquiv x2 y2 context
    αEquiv _ _ _ = False

    βReductions :: Λ -> [Λ]
    βReductions (App (Λ (x :-: σ) term) n) = [fromJust substitution | isJust substitution] ++ reduceTerm ++ reduceApp
        where
            reduceTerm  = (\newTerm -> App (Λ (x :-: σ) newTerm) n) <$> βReductions term
            reduceApp   = App (Λ (x :-: σ) term) <$> βReductions n
            substitution = substitute term x n
    βReductions (Var _)     = []
    βReductions (Λ x term) = Λ x <$> βReductions term
    βReductions (App x y)  = ((`App` y) <$> βReductions x) ++ (App x <$> βReductions y)
```

The implementation of the `ΛCalculus` typeclass itself also sees little change, with minor adaptations to the pretty printing. The α-equivalence now checks for type equality as well, and β-reduction is unchanged since type checking is handled in the substitution implementation.

```haskell
infixr 7 :->
instance TypedΛCalculus Λ where
    data Type Λ = Pure (VariableName Λ) | (Type Λ) :-> (Type Λ) | Perp | Null
        deriving (Show, Eq, Ord)
```

Here we are getting to some more interesting aspects: the implementation of the type extensions in the `TypedΛCalculus` typeclass. As mentioned before, we implement a type for our calculus using the type family defined on the typeclass. We have four cases: a pure type of some string (coinciding with variable names), a function type with infix constructor (and appropriate fixity), `Perp` for inconsistent or impossible types, and `Null` for unknown types (that we can fill using `deduceTypes`).

```
prettyType :: Type Λ -> String
prettyType (Pure σ)           = σ
prettyType (σ@(Pure _) :-> τ) =       prettyType σ ++  "->" ++ prettyType τ
prettyType (σ :-> τ)          = "(" ++ prettyType σ ++ ")->" ++ prettyType τ
prettyType Null               = "?"
prettyType Perp               = "□"

typesEquivalent :: Type Λ -> Type Λ -> EquivalenceContext Λ -> Bool
typesEquivalent x y _ = x == y
```

Pretty printing is relatively straightforward, and type equivalence is just direct identity since we don't allow quantification over type variables - we defer that to the polymorphic case.

```
typeOf :: Λ -> Maybe (Type Λ)
typeOf (Var (_ :-: σ))     = Just σ
typeOf (Λ (_ :-: σ) term) = (σ :->) <$> typeOf term
typeOf (App x y)          = join $ functionType <$> typeOf x <*> typeOf y
    where
        functionType :: Type Λ -> Type Λ -> Maybe (Type Λ)
        functionType (σ :-> τ) υ | σ == υ = Just τ
        functionType _ _ = Nothing
```

The `typeOf` function is somewhat interesting, as it has to deduce compound types from just the types of the variables. In our implementation of λ-terms, the terms themselves do not carry a type, only variables do. Especially the function application case is somewhat tricky, since we need to ensure that function and argument types align. If they do not, we return `Nothing`.

```
deduceTypes :: Λ -> TypeMapping Λ -> Λ
deduceTypes (Var (x :-: Null)) types
    | isJust mapping = Var (x :-: fromJust mapping)
    | otherwise      = Var (x :-: Null)
    where mapping = lookupSet x types
deduceTypes (Var x) _ = Var x
deduceTypes (Λ (x :-: σ) t) types = Λ (x :-: σ) $ deduceTypes t $ insert (x, σ) types
deduceTypes (App xTerm (Var (x :-: Null))) types
    | not isFunction            = App deduceX (Var (x :-: Null))
    | isNothing mappedType      = App deduceX (Var (x :-: σ))
    | fromJust mappedType == σ  = App deduceX (Var (x :-: σ))
    | otherwise                 = App deduceX (Var (x :-: Null))
    where
        mappedType     = lookupSet x types
        functionType   = typeOf deduceX
        isFunction     = isJust functionType && case fromJust functionType of
                                                (_ :-> _) -> True
                                                _ -> False
        Just (σ :-> _) = functionType -- This will generate a warning but is explicitly safe here
        deduceX        = deduceTypes xTerm types

deduceTypes (App x y) types = App (deduceTypes x types) (deduceTypes y types)
```

The type deduction is probably the trickiest part of the entire implementation here. Our type deduction algorithm is focussed on 'filling holes' in a λ-term, i.e. to replace `Null` types with concrete ones. We use a `TypeMapping` for this, a set of variables and their associated types. The cases for variables and λ-abstractions are relatively straightforward, but function application with just a variable of unknown type is the trickiest. We try to deduce the type in two ways: we try to determine the type of the function, and we see if the variable exists in our type mapping. When both are known, we check the two are the same, and otherwise we pick the appropriate one.

This function might not be able to fill all the holes, and then it will leave them empty. One could decide to instead replace these holes with `Perp`, to indicate an inconsistent type. However, in our 'parsing' implementation down below, we call this function continuously throughout term construction, and thus a judgement of `Perp` might be premature. Leaving the holes empty allows us to repeatedly call this function to saturate the term more and more.

```haskell
    hasValidType :: Λ -> TypeMapping Λ -> Bool
    hasValidType (Var (x :-: σ)) vars    = (x, σ) `elem` vars
    hasValidType (Λ (x :-: σ) term) vars = hasValidType term (insert (x, σ) $ Data.Set.filter (\(y, _) -> x /= y) vars)
    hasValidType t@(App x y) vars        = hasValidType x vars && hasValidType y vars && isJust (typeOf t)
```

Lastly, we have a simple type validity check. It will check whether the type is consistent: whether all variables are always used with the correct type and whether function application is valid as well.

3.1 Parsing

Once again, we implement a developer-friendly 'parsing'. The setup is the same as for the untyped variant, except that variables are no longer only strings, but need to carry a type. The decision was made to always force a type for parameters in an abstraction, but to make types optional in the 'body' of a term, where they can typically be deduced by the code above.

```haskell
class Typeable a where
    toType :: a -> Type Λ

instance Typeable (Type Λ) where toType = id
instance Typeable String where toType = Pure

(==>) :: (Typeable a, Typeable b) => a -> b -> Type Λ
a ==> b = toType a :-> toType b
infixr 7 ==>

data TypeableVariable where
    (:::) :: Typeable a => VariableName Λ -> a -> TypeableVariable
infixl 6 :::

toVariable :: TypeableVariable -> Variable Λ
toVariable (x ::: σ) = x :-: toType σ
```

We first introduce a new typeclass, for parsing types more easily. Instead of having to write `Pure "p" :-> Pure "q" :-> Pure "r"`, we introduce a typeclass `Typeable` (in a similar fashion to `ΛTerm` below) that allows us to convert both types and strings to a type. Since our `ΛVariable` definition only accepts a `Type Λ` instance, and not a generic `Typeable` instance, we introduce a custom data type `TypeableVariable` that allows us to combine a variable name with a generic type, providing a function to convert it all into a proper instance of `Type Λ`. The rest of the parsing code is nearly identical to the untyped version, to we omit it for brevity. The main change in there is that upon every application of `$$`, we call `deduceTypes` to fill the type holes as we go.

Even though the parsing code is so similar to the untyped case, with many shared typeclasses and implementations, unfortunately the implementation could not be made more generic for any instance of `ΛCalculus` due to limitations in the Haskell typing system. Essentially, in an expression like `λ("x" ::: "σ") --> "x" $$ "y"`, the type of neither `(⟶)` nor `($$)` can be deduced: the arrow knows that it should return a `Λ`, but does not know what the term-type of its RHS will be (string or `Λ`), while the `$$` function does not know what the 'target calculus' is - it only knows what type of variables it deals with. Even though it seems like this should be solvable, it turns out to be impossible in Haskell.

3.2 Example Terms

Another set of example terms in simply-typed λ-calculus shows how constrained the calculus is: we can esseitally only work with variables of type `σ` to do any kind of combinations between operations. For a boolean to have a consistent signature, we must have both variables of the same type, and the same holds for pairs. The terms are all equivalent to their untyped counterparts.

```haskell
module TypedLambdaTerms where

import Lambda
import TypedLambda

-- Common lambda terms
λI :: Λ
λI = λ ("x" ::: "σ") --> "x"

-- Boolean values
λtrue, λfalse :: Λ
λtrue = λ ("x" ::: "σ") ("y" ::: "σ") --> "x"
λfalse = λ ("x" ::: "σ") ("y" ::: "σ") --> "y"

-- Conditionals with nice "inline" syntax:
-- "if P then Q else R" ⟺ {λif P ? Q |: R} ⟺ {P $$ Q $$ R}
λif :: Λ -> (Λ -> Λ -> Λ)
λif p q r = p $$ q $$ r

(?) :: (Λ -> Λ -> Λ) -> Λ -> (Λ -> Λ)
(?) p' = p'

(|:) :: (Λ -> Λ) -> Λ -> Λ
(|:) q' = q'

-- Pairs and two pair accessors
λpairType :: Type Λ
λpairType = "σ" ⟹ "σ" ⟹ "σ"

λpair :: Λ
λpair = λ ("x" ::: "σ") ("y" ::: "σ") ("f" ::: λpairType) --> "f" $$ "x" $$ "y"

λp1, λp2 :: Λ
λp1 = λ ("p" ::: λpairType ⟹ "σ") --> "p" $$ (λ ("x" ::: "σ") ("y" ::: "σ") --> "x")
λp2 = λ ("p" ::: λpairType ⟹ "σ") --> "p" $$ (λ ("x" ::: "σ") ("y" ::: "σ") --> "y")
```

4 Polymorphic Λ-calculus a.k.a. System F

The final implementation we provide is one for polymorphic λ-calculus, better known as System F. It allows for quantification over types as an extension of the simply-typed calculus we discussed above. Again, we will only focus on the interesting/unique aspects.

```haskell
data ΛVariable = (VariableName Λ) :-: (Type Λ)
  deriving (Show, Eq, Ord)
infixl 6 :-:

data Λ = Var ΛVariable | Λ ΛVariable Λ | ΛT (VariableName Λ) Λ | App Λ Λ
  deriving (Show)
type Lambda = Λ
```

These definitions are nearly identical to previous instances, except that our `Λ` data type now has a fourth constructor for a quantification over a type parameter. The type parameter is represented by just a name. Another approach would have been to represent quantification over types by a lambda function with a parameter of a `Type` type, but that would arguably lead to uglier code. In the current approach we do still need a `Type` type, so that type variable usages are possible.

```haskell
instance Substitutable Λ String where
    freeVariables :: Λ -> Set (VariableName Λ)
    freeVariables (Var (x :-: σ))     = insert x $ freeVariables σ
    freeVariables (Λ (x :-: _) term) = delete x $ freeVariables term
    freeVariables (ΛT p term)        = delete p $ freeVariables term
    freeVariables (App x y)          = freeVariables x `union` freeVariables y
```

Already in this simple function we see the first signs of what is the biggest complication of quantification over types: we are mixing types and terms. A type variable can occur inside a term, types now have free variables as well, so this greatly complicates our design. The types for our polymorphic calculus will therefore also be an instance of `Substitutable`, since we need to be able to perform substitutions there as well.

```haskell
    renameVariable :: Λ -> VariableName Λ -> VariableName Λ -> Λ
    renameVariable (Var (x :-: σ)) old new
        | x == old  = Var (new :-: renameVariable σ old new)
        | otherwise = Var (x   :-: renameVariable σ old new)
    renameVariable (Λ (x :-: σ) term) old new
        | x == old  = Λ (new :-: renameVariable σ old new) $ renameVariable term old new
        | otherwise = Λ (x   :-: renameVariable σ old new) $ renameVariable term old new
    renameVariable (ΛT p term) old new
        | p == old  = ΛT new $ renameVariable term old new
        | otherwise = ΛT p   $ renameVariable term old new
    renameVariable (App x y) old new = App (renameVariable x old new) (renameVariable y old new)

    prepareSubstitution :: Λ -> VariableName Λ -> Λ
    prepareSubstitution (Λ (x :-: σ) term) var
        | x /= var  = Λ (x       :-: prepareSubstitution σ var) $ prepareSubstitution term var
        | otherwise = Λ (newName :-: prepareSubstitution σ var) $ prepareSubstitution (renameVariable
        ↪  term x newName) var
        where newName = "_" ++ var
    prepareSubstitution (ΛT p term) var
        | p /= var  = ΛT p       $ prepareSubstitution term var
        | otherwise = ΛT newName $ prepareSubstitution (renameVariable term p newName) var
        where newName = "_" ++ var
    prepareSubstitution (App x y) var = App (prepareSubstitution x var) (prepareSubstitution y var)
```

```
    prepareSubstitution (Var (x :-: σ)) var = Var (x :-: prepareSubstitution σ var)

    performSubstitution :: Λ -> VariableName Λ -> Λ -> Maybe Λ
    performSubstitution (Var (x :-: σ)) var term
        | x /= var              = Just $ Var (x :-: σ)
        | typeOf term /= Just σ = Nothing
        | otherwise             = Just term
    performSubstitution (Λ (x :-: σ) t) var term
        | x /= var  = Λ (x :-: σ) <$> performSubstitution t var term
        | otherwise = Just $ Λ (x :-: σ) t
    performSubstitution (ΛT p t) var term
        | p /= var  = ΛT p <$> performSubstitution t var term
        | otherwise = Just $ ΛT p t
    performSubstitution (App x y) var term = App <$> performSubstitution x var term <*>
     ↪  performSubstitution y var term
```

Renaming here is still relatively straightforward - it just needs to be propagated to inside the types as well, the same holds for preparing a substitution. Performing one is a bit more challenging, since we now need to be more careful with the types. We also need to distinguish between applying a variable to a λ-term, or a Λ-term - the latter being quantification over types.

```
instance ΛCalculus Λ where
    type Variable Λ = ΛVariable
```

We omit pretty printing code for brevity

```
    αEquiv :: Λ -> Λ -> EquivalenceContext Λ -> Bool
    αEquiv (Var (x :-: σ)) (Var (y :-: τ)) context
        = typesEquivalent σ τ context && (x == y || (x, y) `elem` context)

    αEquiv (Λ (x :-: σ) xTerm) (Λ (y :-: τ) yTerm) context
        = notCrossBound && typesEquivalent σ τ context && αEquiv xTerm yTerm ((x, y) : context)
        where
            yFreeInX = y `elem` freeVariables xTerm
            xFreeInY = x `elem` freeVariables yTerm
            notCrossBound = x == y || (not yFreeInX && not xFreeInY)

    αEquiv (ΛT x xTerm) (ΛT y yTerm) context
        = notCrossBound && αEquiv xTerm yTerm ((x, y) : context)
        where
            yFreeInX = y `elem` freeVariables xTerm
            xFreeInY = x `elem` freeVariables yTerm
            notCrossBound = x == y || (not yFreeInX && not xFreeInY)

    αEquiv (App x1 x2) (App y1 y2) context = αEquiv x1 y1 context && αEquiv x2 y2 context
    αEquiv _ _ _ = False

    βReductions :: Λ -> [Λ]
    βReductions (App (Λ (x :-: σ) term) n) = [fromJust substitution | isJust substitution ] ++
     ↪  reduceTerm ++ reduceApp
        where
            reduceTerm   = (\newTerm -> App (Λ (x :-: σ) newTerm) n) <$> βReductions term
            reduceApp    = App (Λ (x :-: σ) term) <$> βReductions n
            substitution = substitute term x n
    βReductions (App (ΛT p term) (Var (q :-: Type))) = [fromJust substitution | isJust substitution]
     ↪  ++ reduceTerm
        where
```

```haskell
            reduceTerm   = (\newTerm -> App (ΛT p newTerm) (Var (q :-: Type))) <$> βReductions term
            substitution = substituteTypes term p (Pure q)
    βReductions (Var _)      = []
    βReductions (Λ x term)  = Λ x  <$> βReductions term
    βReductions (ΛT p term) = ΛT p <$> βReductions term
    βReductions (App x y)   = ((`App` y) <$> βReductions x) ++ (App x <$> βReductions y)
```

The α-equivalence check is a bit more extensive with an additional case to check (type quantification) and a more sophisticated check for type equivalence. β-reductions might be the trickiest in this section, with a newly added type quantification, but even that follows the existing patterns that we already know.

```haskell
instance Substitutable (Type Λ) String where
    freeVariables :: Type Λ -> Set (VariableName Λ)
    freeVariables (Pure σ)       = singleton σ
    freeVariables (σ :-> τ)      = freeVariables σ `union` freeVariables τ
    freeVariables (Forall p τ) = delete p $ freeVariables τ
    freeVariables Perp           = empty
    freeVariables Null           = empty
    freeVariables Type           = empty

    renameVariable :: Type Λ -> VariableName Λ -> VariableName Λ -> Type Λ
    renameVariable (Pure σ)      old new
        | σ /= old  = Pure σ
        | otherwise = Pure new
    renameVariable (σ :-> τ)     old new = renameVariable σ old new :-> renameVariable τ old new
    renameVariable (Forall p τ) old new
        | p /= old = Forall p $ renameVariable τ old new
        | p == old = Forall p τ
    renameVariable σ _ _ = σ

    prepareSubstitution :: Type Λ -> VariableName Λ -> Type Λ
    prepareSubstitution (σ :-> τ) var = prepareSubstitution σ var :-> prepareSubstitution τ var
    prepareSubstitution (Forall p τ) var
        | p /= var  = Forall p $ prepareSubstitution τ var
        | otherwise = Forall newName $ prepareSubstitution (renameVariable τ p newName) var
        where newName = "_" ++ var
    prepareSubstitution σ _ = σ

    performSubstitution :: Type Λ -> VariableName Λ -> Type Λ -> Maybe (Type Λ)
    performSubstitution (Pure σ)      var term
        | σ /= var  = Just $ Pure σ
        | otherwise = Just term
    performSubstitution (σ :-> τ)     var term = (:->) <$> performSubstitution σ var term <*> performSubstitution τ var term
    performSubstitution (Forall p t) var term
        | p /= var  = Forall p <$> performSubstitution t var term
        | otherwise = Just $ Forall p t
    performSubstitution σ _ _ = Just σ

substituteTypes :: Λ -> VariableName Λ -> Type Λ -> Maybe Λ
substituteTypes (Var (x :-: σ)) var term = Var <$> ((x :-:) <$> substitute σ var term)
substituteTypes (Λ (x :-: σ) t) var term = Λ   <$> ((x :-:) <$> substitute σ var term) <*> substituteTypes t var term
substituteTypes (ΛT p t) var term
 | p /= var  = ΛT p <$> substituteTypes t var term
 | otherwise = Just $ ΛT p t
```

```
substituteTypes (App x y) var term = App <$> substituteTypes x var term <*> substituteTypes y var term
```

Now we make sure that our type system (to be defined further down in the `TypedΛCalculus` instance) allows substitution as well. The function implementations follow the patterns and conventions we have used before, just with different constructors and semantics. We do introduce a special function 'bridging' between terms and types: allowing us to substitute a type directly everywhere in a term.

```
infixr 7 :->
instance TypedΛCalculus Λ where
    data Type Λ = Pure (VariableName Λ) | (Type Λ) :-> (Type Λ) | Forall (VariableName Λ) (Type Λ) | Perp | Null | Type
        deriving (Show, Ord)
```

The type system for our polymorphic calculus extends the simple types from before by adding a `Forall` constructor - serving as quantification over tyeps. We do not introduce special syntax or notation for it, since writing out 'forall' is already the most readable approach.

```
    prettyType :: Type Λ -> String
    prettyType (Pure σ)           = σ
    prettyType (σ@(Pure _) :-> τ) = prettyType σ ++ "->" ++ prettyType τ
    prettyType (σ :-> τ)          = "(" ++ prettyType σ ++ ")->" ++ prettyType τ
    prettyType (Forall p σ)       = "□" ++ p ++ "." ++ prettyType σ
    prettyType Null               = "?"
    prettyType Perp               = "□"
    prettyType Type               = error "Invalid"

    typesEquivalent :: Type Λ -> Type Λ -> EquivalenceContext Λ -> Bool
    typesEquivalent (Pure σ)     (Pure τ)     context = σ == τ || (σ, τ) `elem` context
    typesEquivalent (σ :-> σ')   (τ :-> τ')   context = typesEquivalent σ τ context && typesEquivalent σ' τ' context
    typesEquivalent (Forall p σ) (Forall q τ) context
        = notCrossBound && typesEquivalent σ τ ((p, q) : context)
        where
            qFreeInΣ = q `elem` freeVariables σ
            pFreeInτ = p `elem` freeVariables τ
            notCrossBound = p == q || (not qFreeInΣ && not pFreeInτ)
    typesEquivalent Perp Perp _ = True
    typesEquivalent Type Type _ = True
    typesEquivalent _    _    _ = False
```

Pretty printing works as expected, but in this case we need the type equivalence to be more sophisticated than a simple built-in equality - since we should identify types of the same structure but with different bound variable names ($\forall p.p \to p \equiv \forall q.q \to q$). The structure is the same as for α-equivalence of λ-terms.

```
    typeOf :: Λ -> Maybe (Type Λ)
    typeOf (Var (_ :-: σ))             = Just σ
    typeOf (Λ (_ :-: σ) term)          = (σ :->)  <$> typeOf term
    typeOf (ΛT p term)                 = Forall p <$> typeOf term
    typeOf (App x (Var (y :-: Type))) = forallType =<< typeOf x
        where
            forallType :: Type Λ -> Maybe (Type Λ)
            forallType (Forall p t) = substitute t p (Pure y)
```

```haskell
            forallType _            = Nothing

    typeOf (App x y)
        = join $ functionType <$> typeOf x <*> typeOf y
        where
            functionType :: Type Λ -> Type Λ -> Maybe (Type Λ)
            functionType (σ :-> τ) υ | σ == υ = Just τ
            functionType _ _ = Nothing
```

The typeOf function now has an additional code for quantification, where we need to check that it is only applied to 'type variables' and not to any ordinary variable.

```haskell
deduceTypes :: Λ -> TypeMapping Λ -> Λ
deduceTypes (Var (x :-: Null)) types
    | isJust mapping = Var (x :-: fromJust mapping)
    | otherwise      = Var (x :-: Null)
    where mapping = lookupSet x types
deduceTypes (Var x)            _      = Var x
deduceTypes (Λ (x :-: σ) t) types = Λ (x :-: σ) $ deduceTypes t $ insert (x, σ) types
deduceTypes (ΛT p t)        types = ΛT p $ deduceTypes t $ insert (p, Type) types
deduceTypes (App xTerm (Var (x :-: Null))) types
    | not isFunction && not isForall = App deduceX (Var (x :-: Null))
    | isFunction && isNothing mappedType = App deduceX (Var (x :-: σ))
    | isFunction && fromJust mappedType == σ = App deduceX (Var (x :-: σ))
    | isForall && isNothing mappedType = App deduceX (Var (x :-: Type))
    | isForall && fromJust mappedType == Type = App deduceX (Var (x :-: Type))
    | otherwise = App deduceX (Var (x :-: Null))
    where
        mappedType = lookupSet x types
        functionType = typeOf deduceX
        isFunction = isJust functionType && case fromJust functionType of
                                            (_ :-> _) -> True
                                            _ -> False
        Just (σ :-> _) = functionType -- This will generate a warning but is explicitly safe here
        isForall = isJust functionType && case fromJust functionType of
                                            Forall _ _ -> True
                                            _ -> False
        deduceX = deduceTypes xTerm types

deduceTypes (App x y) types = App (deduceTypes x types) (deduceTypes y types)

hasValidType :: Λ -> TypeMapping Λ -> Bool
hasValidType (Var (x :-: σ))    vars = (x, σ) `elem` vars
hasValidType (Λ (x :-: σ) term) vars = hasValidType term (insert (x, σ) $ Data.Set.filter (\(y, _) -> x /= y) vars)
hasValidType (ΛT p term)        vars = hasValidType term (insert (p, Type) $ Data.Set.filter (\(y, _) -> p /= y) vars)
hasValidType t@(App x y)        vars = hasValidType x vars && hasValidType y vars && isJust (typeOf t)
```

The type deduction system is once again the most complicated part, but still looks very similar to existing cases. The only difference is that for applications, we need to check whether we are applying to a function or a quantification, and check the types of the arguments differently. This also applies to hasValidType.

4.1 Parsing

The parsing is nearly entirely identical to the typed version - with the addition of an abstraction over types. For brevity, we will not repeat the code once again.

4.2 Example Terms

Once more, we show some example terms in polymorphic λ-calculus that highlight the strength of the polymorphic aspect and our developer-friendly syntax. We can easily store types in variables, to make type signatures easier to read, and this allows us to create comprehensive and short expressions.

```
module PolymorphicLambdaTerms where

import Lambda
import PolymorphicLambda

-- Boolean values
λboolean :: Type Λ
λboolean = Forall "p" ("p" ==> "p" ==> "p")

λtrue, λfalse :: Λ
λtrue = lT "p" --> l ("x" ::: "p") ("y" ::: "p") --> "x"
λfalse = lT "p" --> l ("x" ::: "p") ("y" ::: "p") --> "y"

λneg, λland :: Λ
λneg = l ("u" ::: λboolean) --> lT "q" --> l ("x" ::: "q") ("y" ::: "q") --> "u" $$ "q" $$ "y" $$ "x"
λland = l ("u" ::: λboolean) ("v" ::: λboolean) --> lT "q" --> l ("x" ::: "q") ("y" ::: "q")
        --> "u" $$ "q" $$ ("v" $$ "q" $$ "x" $$ "y") $$ ("v" $$ "q" $$ "y" $$ "y")
```

In addition to the standard boolean values, we have defined negation and logical and operations.

```
-- Trees (from the Type Theory exam)
λtree :: Type Λ
λtree = Forall "p" ((λboolean ==> "p") ==> (λboolean ==> "p" ==> "p" ==> "p") ==> "p")

λconstruct :: Λ -> Λ
λconstruct form = lT "p" --> l ("leaf" ::: λboolean ==> "p") ("node" ::: λboolean ==> "p" ==> "p" ==>
↪ "p") --> form

λjoin :: Λ
λjoin = l ("z" ::: λboolean) ("x" ::: λtree) ("y" ::: λtree)
        --> lT "p" --> l ("leaf" ::: λboolean ==> "p") ("node" ::: λboolean ==> "p" ==> "p" ==> "p")
        --> "node" $$ "z" $$ ("x" $$ "p" $$ "leaf" $$ "node") $$ ("y" $$ "p" $$ "leaf" $$ "node")

λfalseNode, λtrueNode, λsimpleFork :: Λ
λfalseNode = λconstruct $ "leaf" $$ λfalse
λtrueNode = λconstruct $ "leaf" $$ λtrue
λsimpleFork = λconstruct $ "node" $$ λtrue $$ ("leaf" $$ λfalse) $$ ("leaf" $$ λtrue)

λsampleJoin :: Λ
λsampleJoin = λjoin $$ λtrue $$ λfalseNode $$ λtrueNode
```