WS 2024/2025

Advanced Functional Programming

Week 2 – Type-Checking and Type-Inference

René Thiemann

Department of Computer Science

Type-Checking and Type-Inference

Static and Dynamic Type-Checking

- every Haskell expression is type-checked
- static type-checking: ill-typed expressions are detected at compile time
- big advantage: well-typed programs cannot go wrong (w.r.t. typing errors)
  - evaluation cannot change the type of an expression
  - example: if `f :: String -> Int` and `e :: String`, then `f e :: Int`, independent of evaluation
  - conclusion: detect type-errors at compile-time, erase types at runtime
- alternative: dynamic type-checking (e.g., Python)
  - dynamic: types are determined at run-time
  - consider `f x = if x > 3.1415 then "foo" else 5`
  - now evaluate `f (approxPi 1000) - 2`
    - only after evaluation of `approxPi 1000` we can determine the Boolean `approxPi 1000 > 3.1415`
    - this Boolean decides whether `f (approxPi 1000)` evaluates to the string `"foo"` or to the number `5`
    - only then we know whether we will get a type error (`"foo" - 2`) or no type-error (`5 - 2`)

```python
def f(x): return ("foo" if x > 3.1415 else 5)

# pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - ...)
def approxPi(x):
  p = 1
  y = 3
  m = -1
  while (x > 0):
     x -= 1
     p += m/y
     y += 2
     m *= -1
  return (4 * p)

# question: do the following python functions lead to type-errors?
def test1(): return f(approxPi(1000)) - 2
def test2(): return f(approxPi(1001)) - 2
```

Static Type-Checking and Type-Inference

- type-checking: given expression `e`, context Γ and type `ty`, determine whether

  $$\Gamma \vdash \texttt{e :: ty} \qquad\qquad (\texttt{e}\text{ has type }\texttt{ty}\text{ in context }\Gamma)$$

  using some typing rules, e.g., the ones of Haskell, ML, ...
  - context Γ: stores types of previously defined variables, functions and constructors
    - Γ might contain `(:) :: a -> [a] -> [a]`, `True :: Bool`, `id :: a -> a`, ...
    - we often just write `e :: ty` instead of $\Gamma \vdash$ `e :: ty` if choice of Γ is clear
- type-inference: given expression `e` and context Γ,
  determine a most general type (aka principal type) of `e` or report non-typability
  - most general: `ty1` is more general than `ty2` if there is some type-substitution τ such that $\texttt{ty1}\tau = \texttt{ty2}$
    - `a` is more general than any type `ty`, choose $\tau := \{\texttt{a}/\texttt{ty}\}$
    - `a -> Int -> b` is more general than `[b] -> Int -> String`, take $\tau := \{\texttt{a}/\texttt{[b]}, \texttt{b}/\texttt{String}\}$
    - `a -> Int -> b` is not more general than `Char -> [Int] -> Char`
- Haskell performs type-inference where type-inference will be applied twice
  - on function definitions in Haskell programs
  - on each expression before it is evaluated in ghci

Non-Deterministic Type-Checking Algorithm

- note: we restrict to expressions built from variables, constants and applications
- algorithm to type-check new definition of `f p1 ... pn = rhs` in context Γ
  - guess a type for `f` of shape `ty1 -> ... -> tyn -> ty`
    (or take a user-defined type annotation for `f`)
  - guess a type for each variable `x` in the defining equation
  - for each constant `c` $\neq$ `f` that appears in the defining equation, guess an instance w.r.t. Γ
    - e.g., if `id :: a -> a` $\in \Gamma$, then each occurrence of `id` can choose a different substitution, e.g., `id :: Int -> Int` and `id :: Bool -> Bool`
  - define a local context Γ' that extends Γ by all guesses
  - type-check definition of `f` by checking $\Gamma' \vdash$ `f p1 ... pn :: ty` and $\Gamma' \vdash$ `rhs :: ty` by recursion on the expressions
    - $\Gamma' \vdash$ `xf :: t` if `xf :: t` $\in \Gamma'$ if `xf` is a variable or `xf = f`
    - $\Gamma' \vdash$ `c :: t` if `c :: t` $\in \Gamma'$ according to guessed instance for each constant `c` $\neq$ `f`
    - type-check all applications

$$\frac{\Gamma' \vdash \texttt{e1 :: t1 -> t2} \qquad \Gamma' \vdash \texttt{e2 :: t1}}{\Gamma' \vdash \texttt{e1 e2 :: t2}}$$

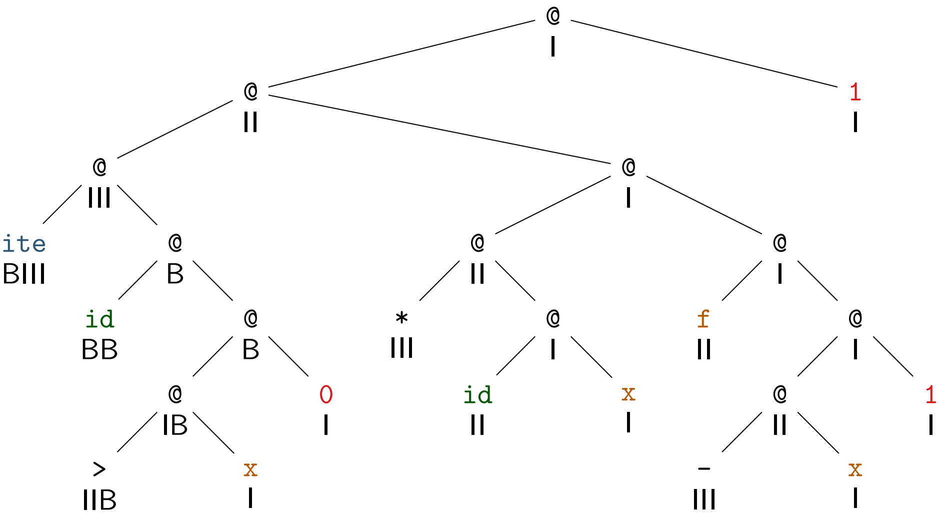
  - finally, store `f :: ty1 -> ... -> tyn -> ty` in Γ

Example

- `f x = if id (x > 0) then id x * f (x - 1) else 1`
  - guess `f :: Int -> Int`
  - guess `x :: Int`
  - instantiate `if-then-else :: Bool -> Int -> Int -> Int` (`if-then-else :: Bool -> a -> a -> a` $\in \Gamma$)
  - instantiate `id :: Bool -> Bool` (`id :: a -> a` $\in \Gamma$)
  - instantiate `(>) :: Int -> Int -> Bool`
  - instantiate `0 :: Int`
  - instantiate `id :: Int -> Int`
  - instantiate `* :: Int -> Int -> Int`
  - instantiate `- :: Int -> Int -> Int`
  - instantiate `1 :: Int`
  - instantiate `1 :: Int`
- on next slide, abbreviate `Int -> Int -> Bool` by `IIB`, etc.

Example Typing

```
f x = if id (x > 0) then id x * f (x - 1) else 1
```



Example Typing

`f x = if id (x > 0) then id x * f (x - 1) else 1`

- guesses work out
  - we assumed `f :: Int -> Int` and `x :: Int`
  - then lhs `f x :: Int` and rhs `if ... :: Int`
  - so `f :: Int -> Int` is added to Γ
- guesses might be too specific, but it is possible to guess most general type

Next Step – Type Inference

- avoid guesses
- compute most generic types instead
- algorithm of Hindley and Milner
  - use very generic types first (which might be too generic)
  - setup constraints
  - solve constraints and thereby specialize initial types

Example Typing – Inferring a Most General Type

```
map f [] = []
map f (x : xs) = f x : map f xs
```

- n-ary function gets type `a0 -> ... -> an` with type-variables `a0, ..., an`
  - `map :: a1 -> a2 -> a3`
- each variable in defining equation gets assigned fresh type-variable
  - `f :: a4` (in principle one could distinguish both `f`s)
  - `x :: a5`
  - `xs :: a6`
- instantiate all type-variables in type of constants by fresh type-variables
  - instantiate `[] :: [a7]`
  - instantiate `[] :: [a8]`
  - instantiate `(:) :: a9 -> [a9] -> [a9]`
  - instantiate `(:) :: a10 -> [a10] -> [a10]`

Example Typing – Setting Up Constraints

```
map f [] = []
map f (x : xs) = f x : map f xs
```

- setup from previous slide
  - `map :: a1 -> a2 -> a3`, `f :: a4`, `x :: a5`, `xs :: a6`
  - `[] :: [a7]`, `[] :: [a8]`, `(:) :: a9 -> [a9] -> [a9]`, `(:) :: a10 -> [a10] -> [a10]`
- further assign type-variables to all non-atomic subexpressions of patterns and rhss
  - `(:) x :: b1`, `x : xs :: b2`, `f x :: b3`, `(:) f x :: b4`, `map f :: b5`, `map f xs :: b6`, `f x : map f xs :: b7`
- finally add constraints to ensure applicability of typing rules
  - `a1 = a4`, first argument of `map` in lhss of equations
  - `a2 = [a7]`, `a2 = b2`, second argument of `map` in lhss of equations
  - `a3 = [a8]`, `a3 = b7`, return type of `map` equals type of rhss in both equations
  - consider each application `e1 e2`
    - lookup types for `e1 :: t1`, `e2 :: t2`, and `e1 e2 :: t3`
    - add constraint `t1 = t2 -> t3`

Example Typing – Final Constraints

```
map f [] = []
map f (x : xs) = f x : map f xs
```

- setup
  - `map :: a1 -> a2 -> a3`, `f :: a4`, `x :: a5`, `xs :: a6`
  - `[] :: [a7]`, `[] :: [a8]`, `(:) :: a9 -> [a9] -> [a9]`, `(:) :: a10 -> [a10] -> [a10]`
  - `(:) x :: b1`, `x : xs :: b2`, `f x :: b3`, `(:) f x :: b4`, `map f :: b5`, `map f xs :: b6`, `f x : map f xs :: b7`
- constraints
  - `a1 = a4`, `a2 = [a7],` `a2 = b2`, `a3 = [a8],` `a3 = b7`
  - `a9 -> [a9] -> [a9] = a5 -> b1`
  - `b1 = a6 -> b2`
  - `a4 = a5 -> b3`
  - `a10 -> [a10] -> [a10] = b3 -> b4`
  - `a1 -> a2 -> a3 = a4 -> b5`
  - `b5 = a6 -> b6`
  - `b4 = b6 -> b7`

Example Typing – Current State

```
map f [] = []
map f (x : xs) = f x : map f xs
```

- setup
  - `map :: a1 -> a2 -> a3`, ...
- constraints
  - $U = \{$`a1 = a4`, `a2 = [a7]`, `a2 = b2`, `a3 = [a8]`, `a3 = b7`, `a9 -> [a9] -> [a9] = a5 -> b1`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 -> [a10] -> [a10] = b3 -> b4`, `a1 -> a2 -> a3 = a4 -> b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`$\}$
- connection of constraints and types via substitution τ, mapping type-variables to types
  - theorem: ($s\tau = t\tau$ for all $s = t \in U$) iff `map :: (a1 -> a2 -> a3)`τ
  - task: find most general τ such $s\tau = t\tau$ for all $s = t \in U$ unification problem
  - such a most general unifier (mgu) τ yields the most general type for `map`
  - unification is decidable and a most general unifier can be computed
  - unification is the core algorithm for type-inference (unification works on terms, and indeed types are terms where `[.]` is unary symbol, `. -> .` is binary symbol, `Bool` and `Int` are constants, etc.)

Unification Algorithm of Martelli & Montanari

Transform unification problem U until no further rules are applicable

- $\{s = s\} \uplus U \hookrightarrow U$ (delete)
- $\{f(s_1,\dots,s_n) = f(t_1,\dots,t_n)\} \uplus U \hookrightarrow \{s_1 = t_1,\dots,s_n = t_n\} \cup U$ (decompose)
- $\{f(\dots) = g(\dots)\} \uplus U \hookrightarrow \bot$, if $f \neq g$ (clash)
- $\{f(\dots) = x\} \uplus U \hookrightarrow \{x = f(\dots)\} \cup U$ (swap)
- $\{x = t\} \uplus U \hookrightarrow \{x = t\} \cup U\{x/t\}$, if $x \in \mathit{Vars}(U) \setminus \mathit{Vars}(t)$ (substitute)
- $\{x = t\} \uplus U \hookrightarrow \bot$, if $x \in \mathit{Vars}(t)$ and $x \neq t$ (occurs check)

Properties

- $\hookrightarrow$ terminates
- if $U \hookrightarrow V$ then U and V have the same unifiers ($\bot$ has no unifiers)
- if $U \hookrightarrow^! V$ ($U \hookrightarrow^* V$ and there is no $\hookrightarrow$-step possible on V) then either
    - $V = \bot$ and U has no unifier, or
    - $V = \{x_1 = t_1,\dots,x_n = t_n\}$ encodes a substitution τ, where the list $x_1,\dots,x_n$ is distinct and $\{x_1,\dots,x_n\} \cap (\mathit{Vars}(t_1) \cup \dots \cup \mathit{Vars}(t_n)) = \emptyset$; moreover τ is an mgu of U

Example: Unification to Determine Type of Map

- `a1 = a4`, `a2 = [a7]`, `a2 = b2`, `a3 = [a8]`, `a3 = b7`, `a9 -> [a9] -> [a9] = a5 -> b1`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 -> [a10] -> [a10] = b3 -> b4`, `a1 -> a2 -> a3 = a4 -> b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- decompose: `a1 = a4`, `a2 = [a7]`, `a2 = b2`, `a3 = [a8]`, `a3 = b7`, `a9 = a5`, `[a9] -> [a9] = b1`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `a2 -> a3 = b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- substitute: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `a3 = b7`, `a9 = a5`, `[a9] -> [a9] = b1`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `[a7] -> a3 = b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- substitute: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = b1`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `[a7] -> [a8] = b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`

Example: Unification to Determine Type of Map

- `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = b1`, <u>`b1 = a6 -> b2`</u>, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `[a7] -> [a8] = b5`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- substitute: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = a6 -> b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `[a7] -> [a8] = b5`, <u>`b5 = a6 -> b6`</u>, `b4 = b6 -> b7`
- substitute: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = a6 -> b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b4`, `[a7] -> [a8] = a6 -> b6`, `b5 = a6 -> b6`, <u>`b4 = b6 -> b7`</u>
- substitute: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = a6 -> b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b6 -> b7`, `[a7] -> [a8] = a6 -> b6`, `b5 = a6 -> b6`, `b4 = b6 -> b7`

Example: Unification to Determine Type of Map

- `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] -> [a9] = a6 -> b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] -> [a10] = b6 -> b7`, `[a7] -> [a8] = a6 -> b6`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- decompose: `a1 = a4`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a9] = a6`, `[a9] = b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[a10] = b6`, `[a10] = b7`, `[a7] = a6`, `[a8] = b6`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- substitute: `a1 = a5 -> b3`, `a2 = [a7]`, `[a7] = b2`, `a3 = [a8]`, `[a8] = b7`, `a9 = a5`, `[a5] = a6`, `[a5] = b2`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `[b3] = b6`, `[b3] = b7`, `[a7] = a6`, `[a8] = b6`, `b5 = a6 -> b6`, `b4 = b6 -> b7`
- swap: `a1 = a5 -> b3`, `a2 = [a7]`, `b2 = [a7]`, `a3 = [a8]`, `b7 = [a8]`, `a9 = a5`, `a6 = [a5]`, `b2 = [a5]`, `b1 = a6 -> b2`, `a4 = a5 -> b3`, `a10 = b3`, `b6 = [b3]`, `b7 = [b3]`, `a6 = [a7]`, `b6 = [a8]`, `b5 = a6 -> b6`, `b4 = b6 -> b7`

Example: Unification to Determine Type of Map

- a1 = a5 -> b3, a2 = [a7], b2 = [a7], a3 = [a8], b7 = [a8], a9 = a5, a6 = [a5], b2 = [a5], b1 = a6 -> b2, a4 = a5 -> b3, a10 = b3, b6 = [b3], b7 = [b3], a6 = [a7], b6 = [a8], b5 = a6 -> b6, b4 = b6 -> b7
- substitute: a1 = a5 -> b3, a2 = [a7], b2 = [a7], a3 = [a8], b7 = [a8], a9 = a5, a6 = [a5], [a7] = [a5], b1 = [a5] -> [a7], a4 = a5 -> b3, a10 = b3, b6 = [b3], [a8] = [b3], [a5] = [a7], [b3] = [a8], b5 = [a5] -> [b3], b4 = b6 -> [a8]
- decompose: a1 = a5 -> b3, a2 = [a7], b2 = [a7], a3 = [a8], b7 = [a8], a9 = a5, a6 = [a5], a7 = a5, b1 = [a5] -> [a7], a4 = a5 -> b3, a10 = b3, b6 = [b3], a8 = b3, a5 = a7, b3 = a8, b5 = [a5] -> [b3], b4 = b6 -> [a8]
- substitute: a1 = a5 -> b3, a2 = [a5], b2 = [a5], a3 = [b3], b7 = [b3], a9 = a5, a6 = [a5], a7 = a5, b1 = [a5] -> [a5], a4 = a5 -> b3, a10 = b3, b6 = [b3], a8 = b3, a5 = a5, b3 = b3, b5 = [a5] -> [b3], b4 = [b3] -> [b3]
- delete: a1 = a5 -> b3, a2 = [a5], b2 = [a5], a3 = [b3], b7 = [b3], a9 = a5, a6 = [a5], a7 = a5, b1 = [a5] -> [a5], a4 = a5 -> b3, a10 = b3, b6 = [b3], a8 = b3, b5 = [a5] -> [b3], b4 = [b3] -> [b3]

Example: Unification to Determine Type of Map

- final result of unification algorithm: mgu τ
  `a1 = a5 -> b3`, `a2 = [a5]`, `b2 = [a5]`, `a3 = [b3]`, `b7 = [b3]`, `a9 = a5`, `a6 = [a5]`, `a7 = a5`, `b1 = [a5] -> [a5]`, `a4 = a5 -> b3`, `a10 = b3`, `b6 = [b3]`, `a8 = b3`, `b5 = [a5] -> [b3]`, `b4 = [b3] -> [b3]`
- most general type of `map`: `(a1 -> a2 -> a3)`τ, i.e.,

```
(a5 -> b3) -> [a5] -> [b3]
```

Remarks

- we introduced fresh variables for every variable, for every argument of the function, and every non-atomic subexpression
  - this provides a systematic way (algorithm) to setup constraints
  - when doing type-inference manually, one often immediately sees certain connections and uses less variables and less constraints
- failures when running the unification algorithm correspond to type-errors of Haskell programs
  - clash appears on type-inference for function `f xs = True ++ xs`:
    constant `(++) :: [a] -> [a] -> [a]`, but fist argument `True :: Bool`;
    this results in clash of equation $[](a) = \texttt{Bool}$
  - occurs check appears on type-inference for function `f x = x : x`:
    subexpression `(:) x :: [a] -> [a]`, but the next argument `x :: a`;
    this results in occurs check of equation $[a] = a$

Extensions of the Type-Inference System

- extend expressions, e.g., by allowing `let` and `\ x -> e` (exercises)
- integrate type-classes
  - several functions are defined in type-classes or have type-class constraints
    - `fromEnum :: Enum a => a -> Int`
    - `sort :: Ord a => [a] -> [a]`
    - `5 :: Num a => a`
  - these constraints have to be collected in addition to the equalities in the unification algorithm
  - whenever the variables in type-class constraints get instantiated, one needs to look into the type-class instances to check the instantiation
  - examples are given on the next slide, without providing a full algorithm

Extensions of the Type-Inference System

- example 1
  - we know `map :: (a -> b) -> [a] -> [b]` and `show :: Show c => c -> String`
  - type-inference on `map show` works as follows
    - `map show :: ([a] -> [b])`τ, for τ being mgu of
      $U = \{$`(a -> b) = (c -> String)`$\}$ for constraints $C = \{$`Show c`$\}$
    - $U \hookrightarrow \{$`a = c`, `b = String`$\}$ and C remains unchanged
    - result: `map show :: Show c => [c] -> [String]` where C is added as constraint
- example 2
  - type-inference on `f x = map show [(x, True, 'c')]` works as follows
    - assume `x :: a`
    - `map show :: Show b => [b] -> [String]`
    - `[(x, True, 'c')] :: [(a, Bool, Char)]`
    - unification leads to `b = (a, Bool, Char)`
    - now `Show b` is instantiated to `Show (a, Bool, Char)` and simplified to `Show a`, since
      - `instance (Show a, Show b, Show c) => Show (a, b, c)`
      - `instance Show Bool`
      - `instance Show Char`
    - result: `f :: Show a => a -> [String]`

Limits of Type-Inference in the Presence of Type-Classes

- consider `f = if 2 * 2^62 < 0 then "overflow" else "okay"`
  - question: which number-type is chosen for the comparison?
    `Int` or `Integer` or `Float` or `Double`
  - type-inference is of no help, e.g., `(<) (2 * 2^62) :: (Num a, Ord a) => a -> Bool`,
    i.e., `2 * 2^62 < 0 :: Bool` for any suitable `a`
- default rule
  - for numeric types, Haskell uses a default rule: choose `Integer` as default, or switch to `Double` if fractional computations are involved (`2.0 < 4`)
  - if one does not want to use default types, provide explicit type annotation
  - note: defaults can be overwritten, e.g. by line `default (Int, Float)`
- examples
  - `f` evaluates to `"okay"`
  - `g = if 2 * 2^62 < (0 :: Int) then "overflow" else "okay"` yields `"overflow"`
  - `[]` in ghci is `show ([] :: [a])` which evaluates to string `[]` after defaulting `a` to `Integer`
  - `[] :: String` in ghci is `show ([] :: String)` which evaluates to string `""`

Limits of Default Rule

- built-in default rule is restricted to built-in numeric type classes
- consider function definition

```haskell
f :: String -> Bool
f xs = show (read xs) == xs
```

- function `f` takes input `xs`, parses it into an element, which is then converted back to a string via `show` and compared to the input
  - `read xs :: Read a => a`
  - `show (read xs) :: (Show a, Read a) => String`
    where `a` is the type for the intermediate result of `read xs`
- it is completely unclear, which type `a` should be: `Int`, `Bool`, `[Double]`, ...
- ghc complains about ambiguous type variables at this point
- solution: provide explicit type annotation, e.g.

```haskell
f xs = show (read xs :: [(Int, Bool)]) == xs
```

Exercises – Task 1 (5 points)

Consider the following definition of a `fold` function on lists:

```
fold f [] e = e
fold f (x : xs) e = f x (fold f xs e)
```

1. Construct constraints to determine the most generic type of `fold`, similarly to Slide 12.
2. Encode the constraints in Haskell, and use the provided implementation of the unification algorithm to obtain the most generic type of `fold`. Compare the computed type to the type-inference algorithm of ghc. The latter can be invoked as follows:

   ```
   cabal repl
   ghci> :m Exercise02
   ghci> :t fold
   ```

Exercises – Task 2 (5 points)

1. Extend the type-inference algorithm so that it can handle λ-abstractions of the form `\ x -> e`, where `x` is a variable and `e` some expression. What will be the constraints for type-inference of `function`?

   ```
   function = \ x -> x x
   ```

2. Compare the difference of type-inference of Haskell when treating λ and **`let`**. To this end, invoke ghc on the following two functions. Try to explain the observed difference.

   ```
   polymorphicLet :: (Bool, String)
   polymorphicLet =
     let f = id
     in (f True, f "hello")

   polymorphicLambda :: (Bool, String)
   polymorphicLambda =
     (\ f -> (f True, f "hello")) id
   ```

Literature

- Simon Thompson, The Craft of Functional Programming, Second Edition, Addison–Wesley, Chapter 13: "Checking Types"
- J. Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society, volume 146, pages 29—60. https://doi.org/10.2307%2F1995158
- Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, volume 17(3), pages 348–375. https://doi.org/10.1016%2F0022-0000%2878%2990014-4