



Advanced Functional Programming

Week 3 – Type-Inference in Haskell, Kinds and Explicit Foralls

René Thiemann

Department of Computer Science

Type-Inference in Haskell

Last Week: Type-Inference Algorithm of Hindley and Milner

- assign **one** type variable to term variables and to arguments of newly defined function
 - $f :: a1 \rightarrow a2 \rightarrow a3, x :: a4, \dots$
- **each** type variable in previously defined function in Γ is replaced by fresh one
 - $(:) :: a5 \rightarrow [a5] \rightarrow [a5]$ for one occurrence of $(:)$, and
 - $(:) :: a6 \rightarrow [a6] \rightarrow [a6]$ for another occurrence of $(:)$
- assign type variable to each non-atomic subterm
 - $(:) x :: b1, \dots$
- constraints ensure that applications are well-typed
 - $a5 \rightarrow [a5] \rightarrow [a5] = a4 \rightarrow b1$ for application $(:) x, \dots$
- unification detects type-problem or results in most general type
 $f :: (a1 \rightarrow a2 \rightarrow a3)\tau$
- finally, $f :: (a1 \rightarrow a2 \rightarrow a3)\tau$ is added to context Γ
- **simplified presentation**: original algorithm merges constraint generation and unification

When to Instantiate Type Variables?

- after a function `f :: ty` has been type checked, future uses of `f` can instantiate the type variables: `f :: ty τ`
`twice :: a -> [a]`
`twice x = [x, x]`
`test1 = (twice True, twice (twice (5 :: Int)))`
- this also happens for locally defined functions within a `let`
`test2 = let twiceLocal x = [x, x] in (twiceLocal True, twiceLocal 'c')`
- in contrast, variables in λ 's or variables in lhss are restricted to one type during type checking; therefore the following functions are not well-typed
`createGen1 p = (p True, p 'c')` `-- not allowed`
`createGen2 = \ p -> (p True, p 'c')` `-- not allowed`
`test3 = createGen1 twice`
`test4 = createGen2 twice`

Instantiations with Explicit Type Annotations

- consider the following program

```
typingTest 0 x y = x
typingTest n x y =
  if typingTest (n - 1) True (x > y)
  then y
  else typingTest (n - 1) y x
```

- using type inference without given type for `typingTest`
 - third line: `typingTest` takes Booleans as second and third argument
 - inferred type: $(Eq\ a, Num\ a) \Rightarrow a \rightarrow Bool \rightarrow Bool \rightarrow Bool$

- using explicit type annotation

```
typingTest :: (Eq a, Num a, Ord b) => a -> b -> b -> b
```

- in each occurrence of rhs, `typingTest` can be instantiated differently
 - line 3: `b = Bool`
 - line 5: `b = b`
- overall, get more general type than by type inference alone
- sometimes, typing is not possible without explicit annotation

Order of Type-Inference

- consider program containing definitions of functions f_1, f_2, \dots
- for type-inference, first a call-graph is constructed and call-dependencies are tracked
- whenever f_i calls f_j , but not vice-versa (directly or indirectly), then f_j is type-checked before f_i
 - consequence: type of f_j can be instantiated when type-checking f_i
- mutually recursive functions are type-checked at the same time

- example

```
f x = if x == 0 then g h else g (g (f (x - 1)))  -- f calls g,h
h = g (f 5)                                     -- h calls f,g
g x = x . x
j c = if c == 'a' then 'z' else i c            -- j calls i
i c = pred c
```

- order: $i < j$ and $g < \{f, h\}$

Types, Type-Expressions and Kinds

Expressions Revisited

- grammar describes building rules of expressions
 - variables are expressions
 - if f is n -ary function and e_1, \dots, e_n are expressions, then so is $f e_1 \dots, e_n$ (first order)
 - function names are expressions (higher order)
 - if e_1 and e_2 are expressions then so is $e_1 e_2$ (higher order)
 - if x is a variable and e is an expression then so is $\lambda x \rightarrow e$ (higher order)
- restriction to **well-typed expressions** `expr :: ty`

Types Revisited

- grammar describes building rules of types
 - variables are types
 - if c is n -ary type-constructor and t_1, \dots, t_n are types, then so is $c t_1 \dots, t_n$ (first order)
- example: `Either (Int, Bool, Char) [[String]]`
- missing: generalization to **higher-order types**
- missing: what are **well-typed types**? `ty :: ???`

Type-Expressions: Higher-Order Types

- generalize grammar for types to higher-order: partial application
- types (or type-expressions) are build as follows
 - variables are type-expressions
 - type-constructors are type-expressions
 - whenever te_1 and te_2 are type-expressions, then so is $te_1 te_2$
- there is no λ for type-expressions
- examples
 - `Either (Int, Bool, Char)` is a type expression
(recall: `data Either a b = Left a | Right b`)
 - `a b` is a type expression
 - `(a b){a/Either (Int, Bool, Char)}` is the type `Either (Int, Bool, Char) b`
- notion used in this course
 - type expressions: as defined by grammar above
 - types: type expressions without partial application
 - clarification: next slides

Kinds: The Type of Type Expressions

- **kinds** are used to describe the structure of types
- kind-inference determines whether some type(-expression) t has kind k , written $t :: k$
- kinds themselves are formed as follows
 - $*$ is a kind, representing types, but not partially applied type expressions
 - if k_1 and k_2 are kinds, then so is $k_1 \rightarrow k_2$
- examples
 - $t :: *$ means that t is a type
 - `Int`, `[Bool]`, `a -> a`, etc.
 - $t :: * \rightarrow *$ means that t is expecting one argument (a type) to become a type
 - `Maybe`, `[]`, `(->) Int`, `Either Bool`
 - $t :: * \rightarrow * \rightarrow *$ (identical to: $* \rightarrow (* \rightarrow *)$)
 - t is a type-expression that expects two types to deliver a type
 - `Either :: * -> * -> *`
 - `(->) :: * -> * -> *`, the function type constructor needs two types

Kinds Continued

- often n -ary type constructor have kind $* \rightarrow * \rightarrow \dots \rightarrow * \rightarrow *$ where there are n many \rightarrow
- example type constructors
 - arity 0: `Int`, `Integer`, `Char`, `Double`, `()`, `Bool`, `Ty` where `data Ty = ...`
 - arity 1: `Maybe`, `Set`, `[]` (the list-type constr.), `Ty` where `data Ty a = ...`
 - arity 2: `Either`, `Map`, `(->)` (the function-type constr.), `Ty` where `data Ty a b = ...`
- rule for determining kinds using some context Γ
 - whenever $\Gamma \vdash \text{ty1} :: k1 \rightarrow k2$ and $\Gamma \vdash \text{ty2} :: k1$ then $\Gamma \vdash \text{ty1 ty2} :: k2$
 - $\Gamma \vdash a :: k$ whenever $a :: k \in \Gamma$ for every type variable a and kind k
- example: `Either Int :: * -> *`, since `Either :: * -> (* -> *)` and `Int :: *`
- example: `try :: k Either Maybe` in `ghci`
- remark: also type-classes have a kind, using the special kind `Constraint`
 - `Show`, `Num`, `Eq`, `Ord` `:: * -> Constraint`
 - classes of shape `class C a b where ...` often have kind `* -> * -> Constraint`

Higher-Order Kinds

- consider `data Ty a b = ...`
- often such a datatype definition results in `Ty :: * -> * -> *`
- however, this is not always the case, since we might have **higher-order kinds**
- example: `data Ty a b = Con (a b)`
 - here, `a :: * -> *`, since `a` is applied on `b`
 - this is automatically inferred using **kinds-inference**
 - consequently: `Ty :: (* -> *) -> * -> *`
 - `Ty` takes a unary type constructor (or more precisely: any type expression of kind `* -> *`) and a type to deliver a type
 - example: `Ty Maybe Int :: *` and `Con (Just 5) :: Ty Maybe Int`
- we will see that even **standard libraries utilize higher-order kinds**
- example: `:t minimum`, `:k Foldable`

Example using Higher-Order Kinds

- week 1 already used types and type-classes with higher-order kinds (docu arrays)

```
class IArray a e where
  bounds :: Ix i => a i e -> (i,i)
  ...
```

- from the type `a i e` we infer

- `a :: * -> * -> *`
- `i :: *`
- `e :: *`

- consequently, class `IArray` requires as arguments

- something of kind `* -> * -> *`, e.g., a binary type constructor `a` (the array constructor),
- and an element type `e`

- in total: `IArray :: (* -> * -> *) -> * -> Constraint`

- example instantiation:

```
instance IArray Array e
```

(the type-constructor `Array` implements `IArray` for every element type `e`)

- starting next week, we will see further type-classes using higher-order kinds

Language Extensions Involving Explicit Forall

Quantification of Type Variables

- consider polymorphic function, e.g., `map :: (a -> b) -> [a] -> [b]`
- type variables are implicitly **universally quantified**
 - we may substitute `a` and `b` by **all** types
- some Haskell extension allows us to make universal application **explicit** by keyword **forall**
`map :: forall a b. (a -> b) -> [a] -> [b]`

Language Extensions

- supported by GHC, extend the Haskell standard
- need to be activated explicitly
 - activation at the beginning of a Haskell file
`{-# LANGUAGE ExplicitForAll, ... #-}`
 - or project-wide activation in cabal-file
default-extensions: `ExplicitForAll, ...`
- just for type-system, there are several extensions

https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/types.html

Extension of Scoped Type Variables

- consider the following Haskell code

```
sortRev :: forall a. Ord a => [a] -> [(a, a)]
sortRev xs = zip sorted reversed where
  sorted :: [a]
  sorted = sort xs
  reversed :: [a]
  reversed = reverse xs
```

- this program does not type check without suitable language extensions
- reason: in each of the three type annotations, the `a` is implicitly quantified, so it is equivalent to use annotations `sorted :: [a1]`, `reversed :: [a2]`
- using extension `ScopedTypeVariables`, the `forall a` binds all `a`'s in the function body, including the `where`-blocks
- then the above code compiles, since all type annotation refer to the same type variable `a`
- remark: activating `ScopedTypeVariables` implicitly activates `ExplicitForAll`

Existential Types

- consider a polymorphic (**universally quantified**) function such as
`map :: forall a b. (a -> b) -> [a] -> [b]`
- view point from user of `map`
 - polymorphism: ability to substitute `a` and `b` by more concrete types
 - the more generic the type is, the more flexible it can be used
- view point from implementation of `map`
 - type variables `a` and `b` cannot be instantiated, represent unknown types
 - the more generic the type is, the less one can perform
 - example: there are only two functions of type `a -> a`
- **existentially quantified types**: change role of user and implementation
 - implementation can instantiate type variables
 - user needs to provide polymorphic input

Example Application: Generic Logger

- consider application, that processes many different kinds of data
- certain events should be logged, each event might have different type
- logging method should be parametric, e.g., log to stdout, log to file, no logging, etc.
- application in Haskell

```
appl :: ??? -> IO ()
```

```
appl log = do
```

```
  inputs <- readFile "inputs.txt"
```

```
  ...
```

```
  log ("init DB access")
```

```
  ...
```

```
  log ("login failed", user, timeStamp)
```

```
  ...
```

```
  log (Transaction client1 amount client2)
```

```
  ...
```

- current type-inference algorithm will fail;
 three incompatible input arguments: a string, a triple, and a custom datatype

Towards Existential Types in Haskell

- consider first order logic
 1. $P(a) \rightarrow Q(f(b))$
 2. $\forall b a. (P(a) \rightarrow Q(f(b)))$
 3. $\forall b. ((\forall a. P(a)) \rightarrow Q(f(b)))$
- formulas 1 and 2 are equivalent (using implicit universal quantification of free variables)
- formula 3 is different, since the \forall is put to the left of an implication
- in fact, formula 3 is equivalent to formula 4 with an existential quantifier
 4. $\forall b. \exists a. (P(a) \rightarrow Q(f(b)))$
- observation: using \forall inside left argument of an implication leads to an \exists on top-level
- formulas 1 and 2 have quantifier alternation depth 1
- formulas 3 and 4 have quantifier alternation depth 2
- now, let us do the same as in formula 3 in Haskell to obtain **existentially quantified types**
 - $\forall = \text{forall}$ and $\rightarrow = \text{->}$
 - quantifier alternation depth = **rank**

Rank

- types without type-variables have rank 0
- types with (implicitly) universally quantified variables have rank 1
- type has rank 2 if it contains a rank-1 type on the left of \rightarrow
- type has rank 3 if it contains a rank-2 type on the left of \rightarrow
- ...
- examples
 - rank 1: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ and `forall a. Show a => a -> a -> String`
 - rank 2: `Ord b => [b] -> (forall a. Show a => a -> String) -> Int -> b`
 - rank 3: `(forall b. (forall a. a -> c -> a) -> [b] -> c) -> Maybe c -> c`
- extension `RankNTypes` enables user to **specify** types with arbitrary rank
- **type-inference with arbitrary ranks is undecidable**
- rule of thumb
 - whenever an explicit `forall` is required, then type has to be user-provided
 - automatic type-inference only works if inferred types have rank of at most 1

Free Usage of forall in Type Definitions of Haskell: **Existential Types**

- swaps role of implementation and user of function
- example assumes type `fun :: (forall a. Ord a => [a] -> a) -> ... -> ...`
 - implementation of `fun` can instantiate `a`, e.g.,
`fun g x = ... g [True, b] ... g [1,7,3]`
 - user of `fun` needs to pass polymorphic function, e.g.,
`... fun minimum ...` or `... fun head ...` or `... fun (\ xs -> xs !! 5) ...`
 - the following invocations are not possible
 - `... fun and ...` `and :: [Bool] -> Bool` is not generic enough
 - `... fun sum ...` `sum :: Num a => [a] -> a` is not generic enough
- generic logger on slide 18 is now typable:
`appl :: (forall a. Show a => a -> IO ()) -> IO ()`
- also `createGen1` and `createGen2` on slide 4 are typable

Generic Logger – Finalized

- type of application

```
appl :: (forall a. Show a => a -> IO ()) -> IO ()
```

- implement different logger algorithms in Haskell

```
noLog x = return ()
```

```
logToFile f x = appendFile f (show x ++ "\n")
```

```
logStdOut x = putStrLn $ "log: " ++ show x
```

- combine application with logger at the very end

```
main1, main2, main3 :: IO ()
```

```
main1 = appl noLog
```

```
main2 = appl (logToFile "log.txt")
```

```
main3 = appl logStdOut
```

Typing of createGen1 and createGen2

```
createGen1, createGen2 :: (forall a. a -> b a) -> (b Bool, b Char)
```

```
createGen1 p = (p True, p 'c')
```

```
createGen2 = \ p -> (p True, p 'c')
```

```
testList1, testList2 :: ([Bool], [Char]) -- b = []
```

```
testList1 = createGen1 (\ x -> [x,x])
```

```
testList2 = createGen2 (\ x -> [x,x])
```

```
testMaybe :: (Maybe Bool, Maybe Char) -- b = Maybe
```

```
testMaybe = createGen1 Just
```

Limits of Higher-Order Type Expressions in Haskell

```
createGen1 :: (forall a. a -> b a) -> (b Bool, b Char)
createGen1 p = (p True, p 'c')
```

```
testList = createGen1 (\ x -> [x, x])
testMaybe = createGen1 Just
testInt = createGen1 (\ _ -> 42)
```

- the output type of `p` may depend on `a`: indicated by using `b a`
- there is no λ -abstraction and β -reduction on types, but just partial application; result: we cannot instantiate `b = \ _ -> Int` and simplify `(\ _ -> Int) a` to `Int`
- consequently, `testInt` is not typable, as `(\ _ -> 42) :: forall a. a -> Int` which is not compatible with `forall a. a -> b a`, no matter how we choose `b`
- workaround via `Const` type

```
newtype Const a b = Const a -- predefined in module Data.Functor.Const
testInt :: (Const Int Bool, Const Int Char)
testInt = createGen1 (\ _ -> Const 42) -- b = Const Int
```


Exercises

- Task 1 (8 points): See `Exercise03.hs` in the sources.
- Task 2 (2 points): Provide a type assignment for `f`, so that the following defining equation is typeable:

```
f x = x x True
```

Literature

- Christopher Allen and Julie Moronuki: Haskell Programming from first principles, Chapter 11.4: “Type constructors and kinds”
- Simon Thompson, The Craft of Functional Programming, Second Edition, Addison–Wesley, Chapter 13: “Checking Types”
- Haskell Report 2010, Chapters 4.5 and 4.6,
<https://www.haskell.org/onlinereport/haskell2010/>
- https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/types.html
- <https://serokell.io/blog/universal-and-existential-quantification>