Software System Design and Implementation

Existentially Quantified Types

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Scope of type variables

data Tree a
  = Leaf
  | Branch a (Tree a) (Tree a)

the type variable a is in scope here
Scope of type variables

we can only use type variables which are in scope

```
data Tree a
    = Leaf
    | Branch b (Tree b) (Tree b)
```

Not in scope: type variable ‘b’
Scope of type variables

but we don’t have to use them (phantom types):

data Length a = Length Double

data Kilometer

data Miles

addLength :: Length a -> Length a -> Length a
addLength (Length n) (Length m)
  = Length (n + m)
Scope of type variables

With GADT notation:

\[
\begin{align*}
\text{data Tree a where} \\
\text{Leaf :: Tree a} \\
\text{Branch :: a -> Tree a -> Tree a -> Tree a}
\end{align*}
\]

which is equivalent to:

\[
\begin{align*}
\text{data Tree a where} \\
\text{Leaf :: Tree a} \\
\text{Branch :: b -> Tree b -> Tree b -> Tree b}
\end{align*}
\]
Scope of type variables

Type variables are implicitly $\forall$-quantified:

```haskell
data Tree a where
  Leaf :: forall a. Tree a
  Branch :: forall a. a -> Tree a -> Tree a -> Tree a
```

```haskell
data Tree a where
  Leaf :: forall a. Tree a
  Branch :: forall b. b -> Tree b -> Tree b -> Tree b
```
Scope of type variables

- Type variables don’t have to appear in the result

```haskell
data M where
  MC :: a -> M
```

- or in non-GADT notation (needs language extension enabled)

```haskell
data M = forall a. MC a
```
Scope of type variables

data M where
    MC :: a -> M

• We can define a list of values of type M:

    xs :: [M]
    xs = [MC 5, MC True, MC "Why??"]

There is nothing we can do with values of type M!
Existential Types

• So, what is the actual type of unpackM?

  \( \text{unpackM :: M \rightarrow a} \)

• Recall that type variables in Haskell are implicitly \( \forall \)-quantified, so the above type is the same as

  \( \text{unpackM :: forall a. M \rightarrow a} \)

• But the real type of unpackM is (which can’t be expressed in Haskell):

  \( \text{unpackM :: \exists a. M \rightarrow a} \)

• This is why these types are called ‘existential types’

{-# LANGUAGE ExistentialQuantification #-}

data M = forall a. MC a
Existential Types

data N where
   NC :: Show a => a -> N

data P where
   PC :: (a -> String) -> a -> P

showNs :: [N] -> [String]
showNs ns = map show' ns
   where
      show' (NC x) = show x

showPs :: [P] -> [String]
showPs ps = map ((\(PC f p) -> f p) ps)
Example: Shapes

- Haskell:

```haskell
data Shape
    = Circle ...
    | Rectangle ...
    | Square ...

perimeter :: Shape -> Double
perimeter (Circle ...) = ...
perimeter (Rectangle ...) = ...

area :: Shape -> Double
...
```

- easy to add new functions on the Shape type, less so to add more variants
Example: Shapes

• In OO-languages
  • class Shape
  • Circle, Rectangle, Square extend the class
  • easy to add new variants, less so to add more functions

• Use classes and overloading to model this in Haskell?

class Shape a where
  perimeter :: a -> Double
  area      :: a -> Double

data Circle    = Circle …

instance Shape Circle where
  perimeter (Circle ...) = …
  area      (Circle ...) = …
Rank-n polymorphism

- Write a function which, given
  - a polymorphic list constructor function `a -> [a]`
  - and two values of possibly different types
  - applies this function to both values and returns the lists
- Is this function type correct?

```haskell
foo f a b = (f a, f b)
```

- **Problem:** we can write polymorphic functions in vanilla Haskell, but we can express the fact that we want a polymorphic function as argument
Rank-n polymorphism

- **Problem:** we can write polymorphic functions in vanilla Haskell, but we can’t express the fact that we want a polymorphic function as argument.

- Again, this is a scoping issue:

\[ \forall a. \forall b. (a \rightarrow [a]) \rightarrow a \rightarrow b \rightarrow ([a], [b]) \]

versus

\[ \forall a. \forall b. (\forall a. a \rightarrow [a]) \rightarrow a \rightarrow b \rightarrow ([a], [b]) \]
Rank-n polymorphism

- **Rank-n polymorphism** makes this possible

\[
\forall a. \forall b. (\forall a. a \rightarrow [a]) \rightarrow a \rightarrow b \rightarrow ([a], [b])
\]

rank-2 polymorphic function

- **Rank-n polymorphism** can be used to control what information a function has access to
Remember the ST monad?

```haskell
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
runST :: (forall s. ST s a) -> a
```
Existential Types and Rank-n types

- Note the difference:

\[
\text{data } M \text{ where } \\
\quad \text{MC} :: a \rightarrow M
\]

\[
\text{data } M \text{ where } \\
\quad \text{MC} :: \text{forall } a. a \rightarrow M
\]

\[
\text{data } M = \text{forall } a. \text{MC } a
\]

\[
\text{data } M \text{ where } \\
\quad \text{MC} :: (\text{forall } a.a) \rightarrow M
\]

\[
\text{data } M = \text{MC } (\text{forall } a. a)
\]
Error Handling

- Two types of errors:
  - **Fatal errors**: indicates serious problems that an application should not try to catch, as it requires external fix: program bug, stack overflow…
  - **Non-fatal errors**: conditions that an application should catch and handle.

- Further distinction
  - **Synchronous errors**: 
    - raised as a direct consequence by the program itself
  - **Asynchronous errors**: 
    - timeouts, user interrupt, resource exhaustion
Asynchronous error handling

• Asynchronous errors can happen at any time
• Can’t (in general) be prevented from occurring by checks in the program
• Sometimes necessary to mask such exceptions to ensure proper clean-up
Synchronous error handling

• If a function can trigger a non-fatal error, it should in general be reflected in the type:

\[
\text{read} :: \text{Read } a \Rightarrow \text{String} \to a \\
\text{readMaybe} :: \text{Read } a \Rightarrow \text{String} \to \text{Maybe a}
\]

• If the function has to be partial for some reason, raise an appropriate error, don’t just leave the patterns incomplete

• Compiler can detect incomplete patterns

\[-f\text{warn-incomplete-patterns}\]
Synchronous error handling

• How errors are handled depends on programming language:
  • programming language support?
  • possible to throw exceptions?
  • exceptions declared in the type of a function/method?
  • handling statically enforced?