Solving an existential crisis in Haskell

Michael B. Gale

Computer Laboratory, University of Cambridge
michael.gale@cl.cam.ac.uk

10 April 2014
Today’s question

How can we add an elimination rule for existential types to Haskell’s type system?
Things that make Haskell great

- Powerful type system
- Extensions include ideas from dependently-typed programming
- Great for embedding other languages
- Entirely static: no types at run-time
Data types in Haskell

\[
\textbf{data} \quad \text{Either } a \, b \quad = \quad \text{Left } a \mid \text{Right } b
\]
Data types in Haskell

```haskell
data Either a b  =  Left a | Right b
```

```
GHCi> :t Left
Left :: a -> Either a b
GHCi> :t Right
Right :: b -> Either a b
```
Data types in Haskell

\[ \textbf{data} \quad \text{Either } a \ b \quad = \quad \text{Left } a \ | \ \text{Right } b \]

\[
\text{GHCi> :t Left} \\
\text{Left :: a -> Either a b}
\]

\[
\text{GHCi> :t Right} \\
\text{Right :: b -> Either a b}
\]

\[
\text{GHCi> :t Left True} \\
\text{Either Bool b}
\]
Pattern matching

\[
\begin{align*}
\text{foo} &:: \text{ Either Bool Int } \to \text{ String} \\
\text{foo} \ (\text{Left} \ x) &\ = \ \text{if } x \ \text{then } "x \ \text{is true}" \ \text{else } "x \ \text{is false}" \\
\text{foo} \ (\text{Right} \ y) &\ = \ \text{if } y == 42 \ \text{then } "y \ \text{is the answer}" \\
&\quad \text{else } "y \ \text{is not the answer}" 
\end{align*}
\]
Existential types

data EEither = ∀a.ELeft a | ∀b.ERight b
Existential types

\[
\text{data } \text{EEither } = \forall a. \text{ELeft } a \mid \forall b. \text{ERight } b
\]

GHCi> :t ELeft
ELeft :: a -> EEither

GHCi> :t ERight
ERight :: b -> EEither
Existential types

\[
\textbf{data} \ E\text{Either} \ = \ \forall a. E\text{Left} \ a \mid \forall b. E\text{Right} \ b
\]

GHCi> :t ELeft
ELeft :: a -> EEither

GHCi> :t ERight
ERight :: b -> EEither

GHCi> :t ELeft True
EEither
Pattern matching revisited

\[
\begin{align*}
  \text{foo} & \quad :: \quad \text{EEither} \to \text{String} \\
  \text{foo} \ (\text{ELeft} \ x) & \quad = \quad ??? \\
  \text{foo} \ (\text{ERight} \ y) & \quad = \quad ???
\end{align*}
\]
Pattern matching revisited

\[ \text{foo} :: \text{Either} \rightarrow \text{String} \]
\[ \text{foo} (\text{Left } x) = ??? \]
\[ \text{foo} (\text{Right } y) = ??? \]

Our \textit{Either} type is completely useless!
Making existentials useful

\[
\textbf{data } \textit{EEither} \quad = \quad \forall a. \textit{ELeft} (a \to \textit{String}) \ a \mid \forall b. \textit{ERight} (b \to \textit{String}) \ b
\]
Making existentials useful

\[
\text{data } \text{EEither } = \forall a. \text{EVal } (a \rightarrow \text{String}) \ a
\]
\[
\text{testVal } :: \text{EEither }
\]
\[
\text{testVal } = \text{EVal } (\lambda y \rightarrow \text{if } y == 42 \\
\quad \text{then } "y is the answer"
\quad \text{else } "y is not the answer") 23
\]
Making existentials useful

\[
\begin{align*}
\text{foo} & \quad :: \quad \text{EEither} \rightarrow \text{String} \\
\text{foo} (\text{EVal}\ f\ x) & \quad = \quad f\ x
\end{align*}
\]
Subtyping

Circle "is a" Shape

Solving an existential crisis in Haskell

11 BCTCS'14
Haskell encoding

• A data type for each class:

  \[
  \begin{align*}
  \textbf{data} & \quad \text{Shape} & = & \quad \text{ShapeSelf} & \text{Int} & \text{Int} \\
  \textbf{data} & \quad \text{Circle} & = & \quad \text{CircleSelf} & \text{Int} & \text{Int} & \text{Int}
  \end{align*}
  \]

• Encoding of methods is not important for this talk
Casting to a parent class

```haskell
data Shape = ShapeSelf Int Int
            | ∀a. ShapeWrapper a

data Circle = CircleSelf Int Int Int
```

Solving an existential crisis in Haskell
Casting to a parent class

- Ask for the type of *ShapeWrapper* in GHCi:
  
  GHCi> :t ShapeWrapper
  ShapeWrapper :: a -> Shape

- (In practice we will probably want to place a constraint on *a*)
Casting to a parent class

- Casting from a Circle to a Shape is just more specialised:

  \[
  \text{upcast} :: \text{Circle} \rightarrow \text{Shape} \\
  \text{upcast} = \text{ShapeWrapper}
  \]

- Quite elegant!
Casting to a child class?

```haskell
data Maybe a = Just a | Nothing

downcast :: Shape → Maybe Circle

downcast (ShapeWrapper x) = ???
downcast _ = Nothing
```

- No way to find out what the type of \( x \) is here
- Only place where we have this information is in \( upcast \)
Solution?

data Shape = ShapeSelf Int Int
  | ∀a. ShapeWrapper (a → Maybe Circle) a

data Circle = CircleSelf Int Int Int

upcast :: Circle → Shape
upcast = ShapeWrapper Just

downcast :: Shape → Maybe Circle
downcast (ShapeWrapper f x) = f x
Solution?

- One downcast function per sub-class (finite and closed)
- Neat because Haskell’s type system will ensure that only one is set to *Just*:

\[ \forall a. \text{ShapeWrapper} \ (a \rightarrow \text{Maybe Circle}) \ (a \rightarrow \text{Maybe Triangle}) \ a \]

```
GHCI> :t ShapeWrapper Just Just
Couldn’t match type ‘Circle’ with ‘Triangle’
in the second argument of ‘ShapeWrapper’
```
Solving an existential crisis in Haskell
Open vs Closed

- Set of types for $a$ is open and infinite (can use any type)
- Solution so far provides “unwrapping” functions for a closed and finite subset
- We want this set to be open
Open vs Closed

• We need something along the lines of:

\[
\text{data } Shape = \begin{cases} 
\text{ShapeSelf Int Int} \\
\forall a. \text{ShapeWrapper} \ (\forall b. a \rightarrow \text{Maybe } b) \ a 
\end{cases}
\]

• This function would know what \(a\) is, but would have to work for all \(b\)

• Placing a constraint on the two combined is not useful:

\[
\forall a. \text{ShapeWrapper} \ (\forall b. \text{Castable } a \ b \Rightarrow a \rightarrow \text{Maybe } b) \ a
\]
Cheating the system

- Use tags and promise to choose unique ones for different types:

```haskell
data Shape = ShapeSelf Int Int
  | ∀a. ShapeWrapper String a

data Circle = CircleSelf Int Int Int

upcast :: Circle → Shape
upcast = ShapeWrapper "Circle"
```
Cheating the system

• Now we can cheat in `downcast`:

\[
\text{unsafeCoerce} :: a \to b
\]

\[
\text{downcast} :: \text{Shape} \to \text{Maybe Circle}
\]

\[
\text{downcast} (\text{ShapeWrapper "Circle" x}) = \text{unsafeCoerce x}
\]
Cheating the system

- Safe as long as we don’t break our promise
- But if we make a mistake, the type system won’t catch it
- Some Haskell libraries\(^1,2\) use a similar approach

\(^1\)http://hackage.haskell.org/package/base-4.3.1.0/docs/Data-Typeable.html
\(^2\)http://hackage.haskell.org/package/base-4.3.1.0/docs/Data-Dynamic.html
• Provides the following:\(^3\):

```haskell
class Typeable a where
    ofType :: a → TypeRep
    cast :: ∀a b.(Typeable a, Typeable b) ⇒ a → Maybe b
```

\(^3\)Simplified
Data.Typeable

- Idea: `typeOf` returns the same `TypeRep` value for all values of some type.
- If two `TypeDef` values are the same for two values of two (unknown) types, then they must have the same type.
Data.Typeable

```haskell
data Shape = ShapeSelf Int Int
            | \forall a. Typeable a \Rightarrow ShapeWrapper a

data Circle = CircleSelf Int Int Int
              deriving Typeable

downcast :: Shape \rightarrow Maybe Circle
downcast (ShapeWrapper x) = cast x
downcast _ = Nothing
```

Solving an existential crisis in Haskell
Data.Typeable

- Implementation of cast
  
  \[
  \text{cast} :: \forall a b.(\text{Typeable } a, \text{Typeable } b) \Rightarrow a \rightarrow \text{Maybe } b
  \]
  
  \[
  \text{cast } x = \begin{cases} 
  \text{Just } (\text{unsafeCoerce } x) & \text{ if } \text{typeOf } x == \text{typeOf } (\text{undefined } :: b) \\
  \text{Nothing} & \text{else}
  \end{cases}
  \]

Solving an existential crisis in Haskell
Teaching the type system a new trick

Idea: teach the type system how to enforce our “promises”

- Construction: tags should correspond to the right types
- Elimination: restore the right type from a tag
• Declare how values should specialise types:

\[
\text{tagged} \ Shape\text{Wrapper} \ \text{where}
\]

\[
\begin{align*}
\text{ShapeWrapper } "\text{Circle}" & \quad :: \quad \text{Circle} \to \text{Shape} \\
\text{ShapeWrapper } "\text{Rectangle}" & \quad :: \quad \text{Rectangle} \to \text{Shape}
\end{align*}
\]

• \textbf{tagged} declarations are open: we can have another one for \textit{ShapeWrapper} in another module

• May use most types as tags as long as we can pattern-match on the values

• Patterns may not overlap (across all modules)
Construction

- Type system will check that tag and type match
- This is fine:
  \[ \text{ShapeWrapper "Circle" (CircleSelf 5 5 42)} \]

- These are not:
  \[ \text{ShapeWrapper "Cake" (CircleSelf 5 5 42)} \]
  \[ \text{ShapeWrapper "Circle" 8} \]

- Only known tags may be used
- Values must be known at compile-time
Elimination

• Use the tag to refine the type of the constructor:

\[
downcast :: \text{Shape} \to \text{Maybe Circle} \\
downcast (\text{ShapeWrapper "Circle" x}) = \begin{cases} 
\text{Just } x & \text{if } x :: \text{Circle} \\
\text{Nothing} & \text{otherwise}
\end{cases}
\]

• Can pattern-match on any tag value
Why not both?

- Haskell already has the machinery to generate tags
- Combine both!
- Trade-off: store tags or generate when needed?
Future work

- Specifying tag rules is a little annoying: can type inference help?
- Tags need to be known at compile-time: is there a sane way to propagate this information through the type system?
- Can we let tags serve multiple purposes?
Summary

- Easy if we domain is closed
- We restrict the domain to a finite, but open one
- Each type is associated with a tag
- Ensure that tags match types
- Only requires tags where we need them
- Bridge between static / dynamic types
Fin

Email: michael.gale@cl.cam.ac.uk
Web: http://www.cl.cam.ac.uk/~mbg28/