Extensible datatypes

Andres Löh

Institut für Informatik III, Universität Bonn

DGP days Utrecht, 25 August 2005
Overview

1 Motivation
   - Lightweight generic programming
   - Other applications of extensible datatypes

2 Open datatypes and functions
   - Open datatypes
   - Open functions
   - Open problems

3 Related work
Overview

1 Motivation
- Lightweight generic programming
- Other applications of extensible datatypes

2 Open datatypes and functions
- Open datatypes
- Open functions
- Open problems

3 Related work
Generic programming via representation types

- Hinze and Cheney: *A lightweight implementation of generics and dynamics*
- Also: Hinze’s *Fun of Programming* chapter
- Idea: a value of type Rep a is a representation of type a; then use value-level pattern matching to define functions that require a type-level case construct
Implementation of representation types

1. pairs of isomorphisms

```haskell
data Iso :: * → * → * where I :: (a → b) → (b → a) → Iso a b
```

2. equality types

```haskell
data Eq :: * → * → * where P :: (∀f.f a → f b) → Eq a b
```

3. GADTs

```haskell
data Eq :: * → * → * where P :: Eq a a
```
Example

**data Rep :: * → * where**

- $R_{Unit} :: Rep ()$
- $R_{Int} :: Rep Int$
- $R_{Char} :: Rep Char$
- $R_{Either} :: Rep a → Rep b → Rep (Either a b)$
- $R_{(,)} :: Rep a → Rep b → Rep (a, b)$
Example

\begin{verbatim}
data Rep :: * → * where
  RUnit :: Rep ()
  RInt :: Rep Int
  RChar :: Rep Char
  REither :: Rep a → Rep b → Rep (Either a b)
  R(,) :: Rep a → Rep b → Rep (a, b)

eq :: Rep a → a → a → Bool
eq RUnit () () = True
eq RInt n1 n2 = n1 ≡ n2
eq RChar c1 c2 = c1 ≡ c2
eq (REither ra rb) (Left a1) (Left a2) = eq ra a1 a2
eq (REither ra rb) (Right b1) (Right b2) = eq rb b1 b2
eq (REither ra rb) _ _ = False
eq (R(,) ra rb) (a1, b1) (a2, b2) = eq ra a1 a2 ∧ eq rb b1 b2
\end{verbatim}
Evaluation of the approach

This approach seems to have a number of advantages over other generic programming approaches, such as Generic Haskell:

- Lightweight, only modest extensions required. Implemented in GHC.
- Value-level constructs can be reused (pattern matching, recursion).
- Generic functions are first-class.
Higher-order generic functions

\[ \textbf{type } \text{GT} = \forall a.\text{Rep } a \rightarrow a \rightarrow a \]

\[ b u :: \text{GT} \rightarrow \text{Rep } a \rightarrow a \rightarrow a \]

\[ bu \ g \ R_{\text{Unit}} \ () = g \ R_{\text{Unit}} \ () \]

\[ \ldots \]

\[ bu \ g \ (R(,) \ r_a \ r_b) \ (a, b) = g \ (R(,) \ r_a \ r_b) \ (bu \ g \ r_a \ a, bu \ g \ r_b \ b) \]
Higher-order generic functions

type GT = ∀a. Rep a → a → a

bu :: GT → Rep a → a → a

bu g RUnit () = g RUnit ()

... 

bu g (R(,) ra rb) (a, b) = g (R(,) ra rb) (bu g ra a, bu g rb b)

incAge :: GT

incAge RInt n = n + 1

incAge _ x = x
Higher-order generic functions

\textbf{type} \texttt{GT} = \forall a. \texttt{Rep} a \to a \to a

\texttt{bu} :: \texttt{GT} \to \texttt{Rep} a \to a \to a

\texttt{bu} \ g \ R_{\text{Unit}} () = g \ R_{\text{Unit}} ()

\ldots

\texttt{bu} \ g \ (R_{(\_)} r_a r_b) (a, b) = g \ (R_{(\_)} r_a r_b) (\texttt{bu} \ g \ r_a \ a, \texttt{bu} \ g \ r_b \ b)

\texttt{incAge} :: \texttt{GT}

\texttt{incAge} \ R_{\text{Int}} n = n + 1

\texttt{incAge} \quad x = x

\texttt{bu incAge} \ db
What if we want to apply a generic function to a new type that isn’t expressible in terms of Rep (such as \((\to)\) or IO or any other abstract type)?

the behaviour of a generic function on a specific datatype should not follow the generic pattern?
Extensibility?

What if

- we want to apply a generic function to a new type that isn’t expressible in terms of Rep (such as $(\to)$ or IO or any other abstract type)?
- the behaviour of a generic function on a specific datatype should not follow the generic pattern?

Two possibilities:

1. Define a new representation datatype.
2. Extend the Rep datatype.
Define a new representation datatype

- Probably shares a lot of code with the original Rep type.
- We need to convert between real datatypes and their representations for all representation types.
- Most generic functions will use different representation types.
- Higher-order generic functions are not feasible, because they’re tied to one particular representation type.
Extend the Rep datatype

- This is the solution usually taken in the papers.
- It is usually required to adapt the functions such as \textit{eq} and \textit{bu}, too.
Example

Extend the Rep type with a case $Embed$ to represent datatypes that can be encoded using the other constructors.

```haskell
data Rep :: * → * where
    ...
    Embed :: Iso a b → Rep b → Rep a
```

An example of such a type is $Bool$:

$rBool = Embed \text{isoBool} (R \text{Either} R \text{Unit} R \text{Unit})$

The $eq$ function can be extended to work with embedded types.

```haskell
eq :: Rep a → a → a → Bool
    eq ...
    eq (Embed (i a → b i a → b) r b) a1 a2 = eq r b (i a → b a1) (i a → b a2)
```
Example

Extend the Rep type with a case *Embed* to represent datatypes that can be encoded using the other constructors.

```haskell
data Rep :: * → * where
  ...
  Embed :: Iso a b → Rep b → Rep a
```

An example of such a type is *Bool*:

```
rBool = Embed isoBool (REither RUnit RUnit)
```
Example

Extend the Rep type with a case *Embed* to represent datatypes that can be encoded using the other constructors.

```haskell
data Rep :: * → * where
  ...
  Embed :: Iso a b → Rep b → Rep a
```

An example of such a type is *Bool*:

```haskell
rBool = Embed isoBool (REither RUnit RUnit)
```

The *eq* function can be extended to work with embedded types.

```haskell
eq :: Rep a → a → a → Bool
eq ... 

eq (Embed (l i_{a→b} i_{a→b}) r_b) a_1 a_2 = eq r_b (i_{a→b} a_1) (i_{a→b} a_2)
```
Example – continued

Define a new constructor for a specific datatype:

```haskell
data Rep :: * → * where
    ...
    R_Bool :: Rep Bool
```

Now we can give a specific behaviour of `eq` for `Bool`:

```haskell
eq Bool a 1 a 2 = False
```

We can also assign a default behaviour to `eq`:

```haskell
embed :: Rep a → Rep a
embed R_Bool = rBool

eq :: Rep a → a → a → Bool
    eq . . = eq (embed r a 1 a 2)
```

Andres Löh (U Bonn) Extensible datatypes DGP 2005
Example – continued

Define a new constructor for a specific datatype:

\[
data \text{ Rep} :: \ast \rightarrow \ast \ \text{where} \\
\ldots \\
R_{\text{Bool}} :: \text{Rep} \ \text{Bool}
\]

Now we can give a specific behaviour of \textit{eq} for \textit{Bool}:

\[
eq \text{Bool} \ a_1 \ a_2 = \text{False}
\]
Define a new constructor for a specific datatype:

```
data Rep :: * → * where
    ...
    R_Bool :: Rep Bool
```

Now we can give a specific behaviour of `eq` for `Bool`:

```
eq Bool a1 a2 = False
```

We can also assign a default behaviour to `eq`:

```
embed :: Rep a → Rep a
embed R_Bool = r_Bool

eq :: Rep a → a → a → Bool
eq ...
eq ra a1 a2 = eq (embed ra) a1 a2
```
Extend the Rep datatype – continued

- Not supported by Haskell, because datatypes are closed.
- We have to rewrite code that is scattered across multiple places and modules.
Not supported by Haskell, because datatypes are closed.

We have to rewrite code that is scattered across multiple places and modules.

As a result, it is not possible to

- define a library for generic programming in this style
- use this encoding as back-end for a language such as Generic Haskell, where we want to support separate compilation
Type classes?

Type classes are open.

However,

- defining one class per generic function leads to generic functions that are not first-class citizens anymore/again
- defining generic functions via Hinze’s *Generics for the masses* does not solve the extensibility problem
Generics for the masses

Definition of equality:

```haskell
newtype Equality a = Equality { applyEquality :: a → a → Bool }
instance Generic Equality where
  unit  = Poly (λ() () → True)
  int   = Poly (λn₁ n₂ → n₁ ≡ n₂)
  char  = ...
  either = ...
  pair  = ...

eq :: (Rep a) ⇒ a → a → Bool
eq = applyEquality rep
```
Generics for the masses

Definition of equality:

```haskell
newtype Equality a = Equality { applyEquality :: a → a → Bool }

instance Generic Equality where
    unit    = Poly (λ() () → True)
    int     = Poly (λn₁ n₂ → n₁ ≡ n₂)
    char    = . . .
    either  = . . .
    pair    = . . .

eq :: (Rep a) ⇒ a → a → Bool
eq                   = applyEquality rep
```

- Pro: One representation class.
- Contra: The “cases” of the generic function are the methods of the class. Classes cannot be extended with new methods.
Other applications: Compilers

\textbf{open} \texttt{Expr} :: * where
\begin{align*}
\texttt{Const} & :: \texttt{Int} \rightarrow \texttt{Expr} \\
\texttt{Add} & :: \texttt{Expr} \rightarrow \texttt{Expr} \rightarrow \texttt{Expr}
\end{align*}
\texttt{eval} :: \texttt{Expr} \rightarrow \texttt{Int}
\begin{align*}
\texttt{eval (Const } n \texttt{)} & = n \\
\texttt{eval (Add } x \, y \texttt{)} & = \texttt{eval } x + \texttt{eval } y
\end{align*}
Other applications: Compilers

\begin{align*}
\textbf{open} \ \textbf{Expr} &:: \ast \ \textbf{where} \\
& \quad \textit{Const} :: \text{Int} \rightarrow \text{Expr} \\
& \quad \textit{Add} :: \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \\
\textbf{eval} &:: \text{Expr} \rightarrow \text{Int} \\
\textbf{eval} \ (\textit{Const} \ n) &= n \\
\textbf{eval} \ (\textit{Add} \ x \ y) &= \textbf{eval} \ x + \textbf{eval} \ y
\end{align*}

\begin{align*}
\textbf{open} \ \textbf{Expr} &:: \ast \ \textbf{where} \\
& \quad \textit{Neg} :: \text{Expr} \rightarrow \text{Expr} \\
\textbf{eval} \ (\textit{Neg} \ x) &= \text{negate} \ (\textbf{eval} \ x)
\end{align*}

Again, we have to extend both the datatype and the function.
This style of programming with open types is similar to AG programming:

| data    | Expr | Const (n : Int) |
|         |      | Add (x : Expr) (y : Expr) |
| attr    | Expr | eval : syn Int |
| sem     | Expr | Const | lhs.eval = n |
|         |      | Add | lhs.eval = x.eval + y.eval |
Other applications: Compilers – continued

This style of programming with open types is similar to AG programming:

\[
\begin{align*}
\textbf{data} & \quad \text{Expr} & \quad | \quad \text{Const} \ (n : \text{Int}) \\
& \quad | \quad \text{Add} \ (x : \text{Expr}) \ (y : \text{Expr}) \\
\textbf{attr} & \quad \text{Expr} & \quad | \quad \text{eval} : \text{syn} \ \text{Int} \\
\textbf{sem} & \quad \text{Expr} & \quad | \quad \text{Const} \ \text{lhs} . \text{eval} = n \\
& \quad | \quad \text{Add} \ \text{lhs} . \text{eval} = x . \text{eval} + y . \text{eval} \\
\end{align*}
\]

\[
\begin{align*}
\textbf{data} & \quad \text{Expr} & \quad | \quad \text{Neg} \ (x : \text{Expr}) \\
\textbf{sem} & \quad \text{Expr} & \quad | \quad \text{Neg} \ \text{lhs} . \text{eval} = x . \text{eval} \\
\end{align*}
\]

There is, however, no direct correspondence for inherited attributes.
Other applications: Exceptions

open Exception

\[ \text{throwIO} :: \text{Exception} \rightarrow \text{IO} \ a \]

\[ \text{catch} :: \text{IO} \ a \rightarrow (\text{Exception} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a \]

Whenever a new form of exception is needed, we can add a new constructor to the Exception type.
Other applications: Exceptions

open Exception

\textit{throw}\textit{IO} :: \textit{Exception} \rightarrow \textit{IO} a

\textit{catch} :: \textit{IO} a \rightarrow (\textit{Exception} \rightarrow \textit{IO} a) \rightarrow \textit{IO} a

Whenever a new form of exception is needed, we can add a new constructor to the \textit{Exception} type.

The \textit{catch} construct is generally used as follows:

\textit{catch} (\textit{expr}) (\lambda e \rightarrow \textbf{case} e \textbf{ of}

\textit{SomeException} . . . \rightarrow . . .

_ \rightarrow \textit{throw} e)
Overview

1 Motivation
- Lightweight generic programming
- Other applications of extensible datatypes

2 Open datatypes and functions
- Open datatypes
- Open functions
- Open problems

3 Related work
Goals and problems

Goals:

- We want open datatypes and functions.
- We want extensibility across multiple modules.

Problems:

- How to deal with export/import restrictions?
- How to deal with pattern matching?
- What about type inference?
- How to implement open functions?
Goals and problems

Goals:
- We want open datatypes and functions.
- We want extensibility across multiple modules.

Problems:
- How to deal with export/import restrictions?
- How to deal with pattern matching?
- What about type inference?
- How to implement open functions?
Open datatypes

An *open datatype* is defined as follows:

```
open TypeName :: kind where
    Constructor_1 :: ...
    ...
```

- Multiple declarations for the same TypeName are possible.
- The name TypeName is in the same namespace as all other datatypes.
- The definition defines a new datatype if TypeName is not yet in scope, it extends the datatype if TypeName is already in scope.
- Implementation is probably less efficient than for closed datatypes, but not really problematic.
Open functions

Functions on open datatypes are open, too. Consider `eval`:

```
eval :: Expr → Int
eval (Const n) = n
eval (Add x y) = eval x + eval y
```
Open functions

Functions on open datatypes are open, too. Consider `eval`:

\[
\begin{align*}
\text{eval} & : \text{Expr} \rightarrow \text{Int} \\
\text{eval} \ (\text{Const} \ n) &= n \\
\text{eval} \ (\text{Add} \ x \ y) &= \text{eval} \ x + \text{eval} \ y \\
\text{eval} \ (\text{Neg} \ x) &= \text{negate} \ (\text{eval} \ x)
\end{align*}
\]
Open functions

Functions on open datatypes are open, too. Consider eval:

```haskell
eval :: Expr → Int
eval (Const n) = n
eval (Add x y) = eval x + eval y

eval (Neg x) = negate (eval x)
```

- Open functions require a type signature.
- If a type signature contains an open type, the function is open.
Haskell’s linear pattern matching is a problem for open functions, because it is hard to define a linear order between different places where the function is defined.

```haskell
module M where { f :: . . . }
module I where { import M; f . . . = . . . }
module J where { import M; f . . . = . . . }
module X where { import I; import J }
```

Even if we had a well-defined order, specifying a default case would effectively close the function.

```haskell
eval _ = 0
```
Best-fit rather than first-fit

The solution is similar to the approach taken for overlapping class instances.

- All branches of a function definition must have the same number of arguments (already the case in Haskell 98).
- The left-most best match is selected.

Therefore:

- Each partial definition of an open function contributes a list of cases/rules.
- The cases are combined/ordered using the above rules for pattern matching.
Example of best-fit pattern matching

\[ f :: [\text{Int}] \rightarrow \text{Either Int Char} \rightarrow \ldots \]

\[ f (x : xs) (\text{Left 1}) \]
\[ f y (\text{Right } a) \]
\[ f (0 : xs) (\text{Right } 'X') \]
\[ f [1] z \]
\[ f [0] z \]
\[ f [] z \]
\[ f [0] (\text{Left } b) \]
\[ f [0] (\text{Left } 2) \]
\[ f y z \]
\[ f [x] z \]
Example of best-fit pattern matching

\[ f :: [\text{Int}] \rightarrow \text{Either Int Char} \rightarrow \ldots \]

\[
\begin{align*}
  f & (x : xs) \quad (\text{Left 1}) \\
  f & y \quad (\text{Right a}) \\
  f & (0 : xs) \quad (\text{Right 'X'}) \\
  f & [1] \quad z \\
  f & [0] \quad z \\
  f & [] \quad z \\
  f & [0] \quad (\text{Left b}) \\
  f & [0] \quad (\text{Left 2}) \\
  f & y \quad z \\
  f & [x] \quad z
\end{align*}
\]

\[ f :: [\text{Int}] \rightarrow \text{Either Int Char} \rightarrow \ldots \]

\[
\begin{align*}
  f & [] \quad z \\
  f & [0] \quad (\text{Left 2}) \\
  f & [0] \quad (\text{Left b}) \\
  f & [0] \quad z \\
  f & (0 : xs) \quad (\text{Right 'X'}) \\
  f & [1] \quad z \\
  f & [x] \quad z \\
  f & (x : xs) \quad (\text{Left 1}) \\
  f & y \quad (\text{Right a}) \\
  f & y \quad z
\end{align*}
\]
Implementing open functions – recursion

- An open function is implicitly parametrized over the final closed version of the function.

\[
\text{eval :: (eval) } \Rightarrow \text{Expr} \rightarrow \text{Int}
\]

Intermediate code:

\[
\begin{align*}
\text{eval} & \quad \text{eval}' \quad (\text{Const} \ n) = n \\
\text{eval} & \quad \text{eval}' \quad (\text{Add} \ x \ y) = \text{eval}' x + \text{eval}' y
\end{align*}
\]

- Other functions that make use of open functions inherit these implicit parameters.

\[
\begin{align*}
\text{f} & \quad = \ldots \text{eval something} \ldots \\
\Rightarrow & \quad \\
\text{f eval'} & \quad = \ldots \text{eval'} something \ldots
\end{align*}
\]
Remaining problems

- Is best-fit pattern matching sufficient for all cases? (Should be for the given examples.)
- First-class rules might be an alternative for best-fit pattern matching.

Like instance declarations, open functions are difficult to deal with in conjunction with modules. Can cases be hidden (possibly by not exporting certain changes)? Can cases be overwritten (possibly if a clear order is recognisable)?

What about open datatypes and deriving/generic functions?

Can we define a transformation between type classes and extensible datatypes?

Open datatypes and functions are closed once, for the whole program. Would it be beneficial to allow to close them earlier, or multiple times? (Related to subtyping.)
Remaining problems

- Is best-fit pattern matching sufficient for all cases? (Should be for the given examples.)
- First-class rules might be an alternative for best-fit pattern matching.
- Like instance declarations, open functions are difficult to deal with in conjunction with modules. Can cases be hidden (possibly by not exporting certain changes)? Can cases be overwritten (possibly if a clear order is recognisable)?
- What about open datatypes and deriving/generic functions?
- Can we define a transformation between type classes and extensible datatypes?
- Open datatypes and functions are closed once, for the whole program. Would it be beneficial to allow to close them earlier, or multiple times? (Related to subtyping.)
Remaining problems

- Is best-fit pattern matching sufficient for all cases? (Should be for the given examples.)
- First-class rules might be an alternative for best-fit pattern matching.
- Like instance declarations, open functions are difficult to deal with in conjunction with modules. Can cases be hidden (possibly by not exporting certain changes)? Can cases be overwritten (possibly if a clear order is recognisable)?
- What about open datatypes and deriving/generic functions?
Remaining problems

- Is best-fit pattern matching sufficient for all cases? (Should be for the given examples.)
- First-class rules might be an alternative for best-fit pattern matching.
- Like instance declarations, open functions are difficult to deal with in conjunction with modules. Can cases be hidden (possibly by not exporting certain changes)? Can cases be overwritten (possibly if a clear order is recognisable)?
- What about open datatypes and deriving/generic functions?
- Can we define a transformation between type classes and extensible datatypes?
Remaining problems

- Is best-fit pattern matching sufficient for all cases? (Should be for the given examples.)
- First-class rules might be an alternative for best-fit pattern matching.
- Like instance declarations, open functions are difficult to deal with in conjunction with modules. Can cases be hidden (possibly by not exporting certain changes)? Can cases be overwritten (possibly if a clear order is recognisable)?
- What about open datatypes and \texttt{deriving}/generic functions?
- Can we define a transformation between type classes and extensible datatypes?
- Open datatypes and functions are closed once, for the whole program. Would it be beneficial to allow to close them earlier, or multiple times? (Related to subtyping.)
Related work

- Type classes.
- GADTs.
- First-class patterns and rules.
- Subtyping.
- Extensible records.