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

$$\mathbf{data}\ \mathsf{Iso} :: * \to * \to * \ \mathbf{where}\ I \ :: (a \to b) \to (b \to a) \to \mathsf{Iso}\ a\ b$$

2. equality types

$$\mathbf{data}\ \mathsf{Eq} :: * \to * \to * \ \mathbf{where}\ P :: (\forall f . f\ a \to f\ b) \qquad \to \mathsf{Eq}\ a\ b$$

3. GADTs

$$\mathbf{data}\ \mathsf{Eq} :: * \to * \to * \ \mathbf{where}\ P :: \qquad \mathsf{Eq}\ a\ a$$

Example

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Rep} :: * \to * \ \textbf{where}} \\
\quad R_{Unit} & :: & \mathsf{Rep}\ () \\
\quad R_{\mathsf{Int}} & :: & \mathsf{Rep\ Int} \\
\quad R_{\mathsf{Char}} & :: & \mathsf{Rep\ Char} \\
\quad R_{\mathsf{Either}} & :: \mathsf{Rep}\ a \to \mathsf{Rep}\ b \to & \mathsf{Rep}\ (\mathsf{Either}\ a\ b) \\
\quad R_{(,)} & :: \mathsf{Rep}\ a \to \mathsf{Rep}\ b \to & \mathsf{Rep}\ (a, b)
\end{array}
$$

Example

$$
\begin{array}{lllll}
\textbf{data}\ \mathrm{Rep} :: * \to * \ \textbf{where} \\
\quad R_{Unit} & :: & & & \mathrm{Rep}\ () \\
\quad R_{\mathrm{Int}} & :: & & & \mathrm{Rep\ Int} \\
\quad R_{\mathrm{Char}} & :: & & & \mathrm{Rep\ Char} \\
\quad R_{\mathrm{Either}} & :: & \mathrm{Rep}\ a \to & \mathrm{Rep}\ b \to & \mathrm{Rep}\ (\mathrm{Either}\ a\ b) \\
\quad R_{(,)} & :: & \mathrm{Rep}\ a \to & \mathrm{Rep}\ b \to & \mathrm{Rep}\ (a, b)
\end{array}
$$

$$
\begin{array}{llllll}
eq :: & \mathrm{Rep}\ a \to & a \to & a & \to & Bool \\
eq & R_{Unit} & () & () & = & True \\
eq & R_{\mathrm{Int}} & n_1 & n_2 & = & n_1 \equiv n_2 \\
eq & R_{\mathrm{Char}} & c_1 & c_2 & = & c_1 \equiv c_2 \\
eq & (R_{\mathrm{Either}}\ r_a\ r_b) & (Left\ a_1) & (Left\ a_2) & = & eq\ r_a\ a_1\ a_2 \\
eq & (R_{\mathrm{Either}}\ r_a\ r_b) & (Right\ b_1) & (Right\ b_2) & = & eq\ r_b\ b_1\ b_2 \\
eq & (R_{\mathrm{Either}}\ r_a\ r_b) & \_ & \_ & = & False \\
eq & (R_{(,)}\ r_a\ r_b) & (a_1, b_1) & (a_2, b_2) & = & eq\ r_a\ a_1\ a_2 \wedge eq\ r_b\ b_1\ b_2
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{type}\ \mathrm{GT} = \forall a. \mathrm{Rep}\ a \to a \to a \\
\begin{array}{lllll}
bu :: \mathrm{GT} \to & \mathrm{Rep}\ a \to & a & \to & a \\
bu\quad g & R_{Unit} & () & = & g\ R_{Unit}\ () \\
\ldots & & & & \\
bu\quad g & (R_{(,)}\ r_a\ r_b) & (a, b) & = & g\ (R_{(,)}\ r_a\ r_b)\ (bu\ g\ r_a\ a, bu\ g\ r_b\ b)
\end{array}
\end{array}
$$

Higher-order generic functions

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

$$\begin{array}{llll}
bu :: \mathsf{GT} \to \mathsf{Rep}\ a \to & a & \to a \\
bu\quad g\quad R_{Unit} & () & = g\ R_{Unit}\ () \\
\ldots & & \\
bu\quad g\quad (R_{(,)}\ r_a\ r_b) & (a, b) & = g\ (R_{(,)}\ r_a\ r_b)\ (bu\ g\ r_a\ a, bu\ g\ r_b\ b)
\end{array}$$

$$\begin{array}{l}
incAge :: \mathsf{GT} \\
incAge\ R_{\mathsf{Int}}\ n = n + 1 \\
incAge\ _\quad x = x
\end{array}$$

Higher-order generic functions

$$\textbf{type}\ \mathsf{GT} = \forall a.\mathsf{Rep}\ a \rightarrow a \rightarrow a$$

$$\begin{array}{llllll}
bu :: \mathsf{GT} \rightarrow & \mathsf{Rep}\ a \rightarrow & a & & \rightarrow a \\
bu\ \ g & R_{Unit} & () & & = g\ R_{Unit}\ () \\
\ldots & & & & \\
bu\ \ g & (R_{(,)}\ r_a\ r_b) & (a, b) & & = g\ (R_{(,)}\ r_a\ r_b)\ (bu\ g\ r_a\ a, bu\ g\ r_b\ b)
\end{array}$$

$$\begin{array}{l}
incAge :: \mathsf{GT} \\
incAge\ R_{\mathsf{Int}}\ n = n + 1 \\
incAge\ _\ \ \ \ x = x
\end{array}$$

$$bu\ incAge\ db$$

Extensibility?

What if

- we want to apply a generic function to a new type that isn't expressible in terms of Rep (such as $(\rightarrow)$ 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 $(\rightarrow)$ 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 *eq* and *bu*, too.

Example

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

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Rep} :: * \to * \ \textbf{where} \\
\quad\quad \ldots \\
\quad \mathit{Embed} :: \mathrm{Iso}\ a\ b \to \mathrm{Rep}\ b \to \mathrm{Rep}\ a
\end{array}
$$

Example

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

$$\textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where}$$
$$\quad \ldots$$
$$\quad \mathit{Embed} :: \mathrm{Iso}\ a\ b \rightarrow \mathrm{Rep}\ b \rightarrow \mathrm{Rep}\ a$$

An example of such a type is *Bool*:

$$\mathit{rBool} = \mathit{Embed}\ \mathit{isoBool}\ (R_{\mathrm{Either}}\ R_{\mathit{Unit}}\ R_{\mathit{Unit}})$$

Example

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

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where} \\
\quad \ldots \\
\quad \mathit{Embed} :: \mathrm{Iso}\ a\ b \rightarrow \mathrm{Rep}\ b \rightarrow \mathrm{Rep}\ a
\end{array}
$$

An example of such a type is *Bool*:

$$
\mathit{rBool} = \mathit{Embed}\ \mathit{isoBool}\ (R_{\mathrm{Either}}\ R_{\mathit{Unit}}\ R_{\mathit{Unit}})
$$

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

$$
\begin{array}{l}
\mathit{eq} :: \mathrm{Rep}\ a \rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\
\mathit{eq} \ldots \\
\mathit{eq}\ (\mathit{Embed}\ (I\ i_{a \rightarrow b}\ i_{a \rightarrow b})\ r_b)\ a_1\ a_2 = \mathit{eq}\ r_b\ (i_{a \rightarrow b}\ a_1)\ (i_{a \rightarrow b}\ a_2)
\end{array}
$$

Example – continued

Define a new constructor for a specific datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where} \\
\quad \ldots \\
\quad R_{Bool} :: \mathrm{Rep}\ \mathit{Bool}
\end{array}
$$

Example – continued

Define a new constructor for a specific datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where} \\
\qquad \ldots \\
\quad R_{Bool} :: \mathrm{Rep}\ Bool
\end{array}
$$

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

$$eq\ Bool\ a_1\ a_2 = False$$

Example – continued

Define a new constructor for a specific datatype:

$$\begin{array}{l} \textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where} \\ \quad \ldots \\ \quad R_{Bool} :: \mathrm{Rep}\ Bool \end{array}$$

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

$$eq\ Bool\ a_1\ a_2 = False$$

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

$$\begin{array}{l} embed :: \mathrm{Rep}\ a \rightarrow \mathrm{Rep}\ a \\ embed \quad R_{Bool} \ = rBool \\ eq :: \mathrm{Rep}\ a \rightarrow a \rightarrow a \rightarrow Bool \\ eq \ldots \\ eq\ r_a\ a_1\ a_2 = eq\ (embed\ r_a)\ a_1\ a_2 \end{array}$$

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.

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.

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:

$$
\begin{aligned}
&\textbf{newtype}\ \mathrm{Equality}\ a = \mathit{Equality}\ \{\mathit{applyEquality} :: a \to a \to \mathit{Bool}\}\\
&\textbf{instance}\ \mathrm{Generic}\ \mathrm{Equality}\ \textbf{where}\\
&\quad \mathit{unit} \quad = \mathit{Poly}\ (\lambda()\ ()\ \to \mathit{True})\\
&\quad \mathit{int} \quad\ \ = \mathit{Poly}\ (\lambda n_1\ n_2 \to n_1 \equiv n_2)\\
&\quad \mathit{char} \quad = \ldots\\
&\quad \mathit{either} = \ldots\\
&\quad \mathit{pair} \quad = \ldots\\
&\mathit{eq} :: (\mathrm{Rep}\ a) \Rightarrow a \to a \to \mathit{Bool}\\
&\mathit{eq} \qquad\qquad\quad = \mathit{applyEquality}\ \mathit{rep}
\end{aligned}
$$

Generics for the masses

Definition of equality:

$$
\begin{aligned}
&\textbf{newtype}\ \mathrm{Equality}\ a = \mathit{Equality}\ \{\mathit{applyEquality} :: a \to a \to \mathit{Bool}\}\\
&\textbf{instance}\ \mathrm{Generic\ Equality}\ \textbf{where}\\
&\quad \mathit{unit} \ \ \ = \mathit{Poly}\ (\lambda()\ ()\ \to \mathit{True})\\
&\quad \mathit{int} \ \ \ \ = \mathit{Poly}\ (\lambda n_1\ n_2 \to n_1 \equiv n_2)\\
&\quad \mathit{char} \ \ = \ldots\\
&\quad \mathit{either} = \ldots\\
&\quad \mathit{pair} \ \ = \ldots\\
&\mathit{eq} :: (\mathrm{Rep}\ a) \Rightarrow a \to a \to \mathit{Bool}\\
&\mathit{eq} \qquad\qquad\quad = \mathit{applyEquality}\ \mathit{rep}
\end{aligned}
$$

- 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

$$
\begin{array}{l}
\textbf{open}\ \mathsf{Expr} :: * \ \textbf{where} \\
\quad \mathit{Const} :: \mathsf{Int} \rightarrow \mathsf{Expr} \\
\quad \mathit{Add} :: \mathsf{Expr} \rightarrow \mathsf{Expr} \rightarrow \mathsf{Expr}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{eval} :: \mathsf{Expr} & \rightarrow & \mathsf{Int} \\
\mathit{eval}\ (\mathit{Const}\ n) & = & n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathit{eval}\ x + \mathit{eval}\ y
\end{array}
$$

Other applications: Compilers

$$
\begin{aligned}
&\textbf{open}\ \mathrm{Expr} :: * \ \textbf{where} \\
&\quad \mathit{Const} :: \mathrm{Int} \rightarrow \mathrm{Expr} \\
&\quad \mathit{Add} :: \mathrm{Expr} \rightarrow \mathrm{Expr} \rightarrow \mathrm{Expr} \\
&\mathit{eval} :: \mathrm{Expr} \rightarrow \mathrm{Int} \\
&\mathit{eval}\ (\mathit{Const}\ n) = n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{open}\ \mathrm{Expr} :: * \ \textbf{where} \\
&\quad \mathit{Neg} :: \mathrm{Expr} \rightarrow \mathrm{Expr} \\
&\mathit{eval}\ (\mathit{Neg}\ x) = \mathit{negate}\ (\mathit{eval}\ x)
\end{aligned}
$$

Again, we have to extend both the datatype and the function.

Other applications: Compilers – continued

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

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Expr} & | & \mathit{Const}\ (n : \mathsf{Int}) \\
 & | & \mathit{Add}\quad (x : \mathsf{Expr})\ (y : \mathsf{Expr}) \\
\textbf{attr}\ \mathsf{Expr} & | & \mathit{eval} : \textbf{syn}\ \mathsf{Int} \\
\textbf{sem}\ \mathsf{Expr} & | & \mathit{Const}\ \textbf{lhs}.\mathit{eval} = n \\
 & | & \mathit{Add}\quad \textbf{lhs}.\mathit{eval} = x.\mathit{eval} + y.\mathit{eval}
\end{array}
$$

Other applications: Compilers – continued

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

$$
\begin{array}{lll}
\textbf{data}\ \mathrm{Expr} & | & \mathit{Const}\ (n : \mathrm{Int}) \\
 & | & \mathit{Add}\quad (x : \mathrm{Expr})\ (y : \mathrm{Expr}) \\
\textbf{attr}\ \mathrm{Expr} & | & \mathit{eval} : \textbf{syn}\ \mathrm{Int} \\
\textbf{sem}\ \mathrm{Expr} & | & \mathit{Const}\ \textbf{lhs}.\mathit{eval} = n \\
 & | & \mathit{Add}\quad \textbf{lhs}.\mathit{eval} = x.\mathit{eval} + y.\mathit{eval}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathrm{Expr} & | & \mathit{Neg}\quad (x : \mathrm{Expr}) \\
\textbf{sem}\ \mathrm{Expr} & | & \mathit{Neg}\quad \textbf{lhs}.\mathit{eval} = x.\mathit{eval}
\end{array}
$$

There is, however, no direct correspondence for inherited attributes.

Other applications: Exceptions

open Exception

$$
\begin{aligned}
&\mathit{throwIO} :: \mathrm{Exception} \to \mathrm{IO}\ a \\
&\mathit{catch} \quad :: \mathrm{IO}\ a \to (\mathrm{Exception} \to \mathrm{IO}\ a) \to \mathrm{IO}\ a
\end{aligned}
$$

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

Other applications: Exceptions

$$\textbf{open}\ \text{Exception}$$

$$\begin{aligned} &\mathit{throwIO} :: \text{Exception} \to \text{IO}\ a \\ &\mathit{catch} \quad\ :: \text{IO}\ a \to (\text{Exception} \to \text{IO}\ a) \to \text{IO}\ a \end{aligned}$$

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

The *catch* construct is generally used as follows:

$$\begin{aligned} &\mathit{catch}\ (\mathit{expr})\ (\lambda e \to \textbf{case}\ e\ \textbf{of} \\ &\qquad\qquad\qquad\qquad \mathit{SomeException} \ldots \to \ldots \\ &\qquad\qquad\qquad\qquad \_ \qquad\qquad\qquad\quad\ \to \mathit{throw}\ e) \end{aligned}$$

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.

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**
$\quad$ $\mathit{Constructor}_1$ :: ...
$\quad$ ...

- 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*:

$$
\begin{aligned}
&\mathit{eval} :: \mathrm{Expr} && \rightarrow \mathrm{Int} \\
&\mathit{eval} \quad (\mathit{Const}\ n) && = n \\
&\mathit{eval} \quad (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}
$$

Open functions

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

$$
\begin{aligned}
&\mathit{eval} :: \mathrm{Expr} &&\to \mathrm{Int}\\
&\mathit{eval}\ \ (\mathit{Const}\ n) &&= n\\
&\mathit{eval}\ \ (\mathit{Add}\ x\ y) &&= \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}
$$

$$
\mathit{eval}\ \ (\mathit{Neg}\ x) = \mathit{negate}\ (\mathit{eval}\ x)
$$

Open functions

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

$$\begin{aligned}
&\mathit{eval} :: \mathsf{Expr} && \to \mathsf{Int} \\
&\mathit{eval}\ (\mathit{Const}\ n) && = n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

$$\mathit{eval}\ (\mathit{Neg}\ x) = \mathit{negate}\ (\mathit{eval}\ x)$$

- Open functions require a type signature.
- If a type signature contains an open type, the function is open.

Implementing open functions – pattern matching

- 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.

$$
\begin{array}{lll}
\textbf{module}\ M & \textbf{where} & \{f :: \ldots\} \\
\textbf{module}\ I & \textbf{where} & \{\textbf{import}\ M; f \ldots = \ldots\} \\
\textbf{module}\ J & \textbf{where} & \{\textbf{import}\ M; f \ldots = \ldots\} \\
\textbf{module}\ X & \textbf{where} & \{\textbf{import}\ I; \textbf{import}\ J\}
\end{array}
$$

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

$$eval\ \_ = 0$$

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

$$
\begin{array}{lll}
f :: [\mathrm{Int}] \to \mathrm{Either\ Int\ Char} \to \ldots \\
f & (x : xs) & (\mathit{Left}\ 1) \\
f & y & (\mathit{Right}\ a) \\
f & (0 : xs) & (\mathit{Right}\ \texttt{'X'}) \\
f & [1] & z \\
f & [0] & z \\
f & [\,] & z \\
f & [0] & (\mathit{Left}\ b) \\
f & [0] & (\mathit{Left}\ 2) \\
f & y & z \\
f & [x] & z
\end{array}
$$

Example of best-fit pattern matching

$$
\begin{array}{llll}
f & :: [\mathrm{Int}] \to \mathrm{Either\ Int\ Char} \to \ldots \\
f & (x : xs) & (\mathit{Left}\ 1) \\
f & y & (\mathit{Right}\ a) \\
f & (0 : xs) & (\mathit{Right}\ \texttt{'X'}) \\
f & [1] & z \\
f & [0] & z \\
f & [\,] & z \\
f & [0] & (\mathit{Left}\ b) \\
f & [0] & (\mathit{Left}\ 2) \\
f & y & z \\
f & [x] & z
\end{array}
$$

$$
\begin{array}{llll}
f & :: [\mathrm{Int}] \to \mathrm{Either\ Int\ Char} \to \ldots \\
f & [\,] & z \\
f & [0] & (\mathit{Left}\ 2) \\
f & [0] & (\mathit{Left}\ b) \\
f & [0] & z \\
f & (0 : xs) & (\mathit{Right}\ \texttt{'X'}) \\
f & [1] & z \\
f & [x] & z \\
f & (x : xs) & (\mathit{Left}\ 1) \\
f & y & (\mathit{Right}\ a) \\
f & y & z
\end{array}
$$

Implementing open functions – recursion

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

$$eval :: (eval) \Rightarrow \mathrm{Expr} \quad \to \mathrm{Int}$$

Intermediate code:

$$\begin{aligned} &eval \quad eval' \quad (Const\ n) &&= n \\ &eval \quad eval' \quad (Add\ x\ y) &&= eval'\ x + eval'\ y \end{aligned}$$

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

$$\begin{aligned} &f &&= \ldots eval \ \ something \ldots \\ &\rightsquigarrow \\ &f\ eval' &&= \ldots eval'\ something \ldots \end{aligned}$$

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.

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)?

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 **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.