Open data types and open functions

Andres Löh and Ralf Hinze

April 10, 2006

Overview

1. Motivation
   - Directions of extensibility
   - Encoding extensibility?
2. Syntax of open data types and open functions
3. Example applications
   - Generic programming
   - Exceptions
4. Semantics
5. Implementation
6. Conclusions

Motivation: The expression problem

Consider a small language of expressions:

- numbers
- addition
- equality
- conditionals (if-statements)

Motivation: The expression problem

Consider a small language of expressions:

- numbers
- addition
- equality
- conditionals (if-statements)

It is easy to write an evaluator for this expression language in nearly any programming language, be it imperative, object-oriented, or functional.

Programs evolve

There are different possibilities to extend the program:

- add new constructs to the expression language
  - multiplication
  - comparisons
  - operations on booleans
  - new base types
  - . . .

Programs evolve

There are different possibilities to extend the program:

- add new constructs to the expression language
  - multiplication
  - comparisons
  - operations on booleans
  - new base types
  - ...
- add more operations next to the evaluator
  - a pretty-printer
  - a simplifier/optimizer
  - an editor
  - ...

Programs evolve

There are different possibilities to extend the program:

- add new constructs to the expression language
  - multiplication
  - comparisons
  - operations on booleans
  - new base types
  - ...
- add more operations next to the evaluator
  - a pretty-printer
  - a simplifier/optimizer
  - an editor
  - ...

Providing both directions of extensibility is known ad the **expression problem**.

Programs evolve

There are different possibilities to extend the program:

- add new constructs to the expression language
    - multiplication
    - comparisons
    - operations on booleans
    - new base types
    - ...
- add more operations next to the evaluator
    - a pretty-printer
    - a simplifier/optimizer
    - an editor
    - ...

Providing both directions of extensibility is known ad the **expression problem**.

How do programming languages support these different forms of program evolution?

OO languages

In object-oriented languages, this is an idiomatic way to model the problem:

- there is a **class** of expressions,
- different constructs of the expression language are **instances** of the class,
- the operations on expressions (such as evaluation, pretty-printing, ...) are **methods** of the class

OO languages, continued

```
class Expr where
  eval     :: Result
  simplify :: Expr
  pprint   :: String
```

OO languages, continued

```
class Expr where
  eval     :: Result
  simplify :: Expr
  pprint   :: String
```

```
class Num implements Expr
  where
  -- specific to Num:
  val       :: Int
  -- Expr interface:
  eval     = self.val
  simplify = ...
  pprint   = ...
```

OO languages, continued

```
class Expr where
  eval     :: Result
  simplify :: Expr
  pprint   :: String
```

```
class Num implements Expr
  where
  -- specific to Num:
  val      :: Int
  -- Expr interface:
  eval     = self.val
  simplify = ...
  pprint   = ...
```

```
class Sum implements Expr
  where
  -- specific to Sum:
  e1       :: Expr
  e2       :: Expr
  -- Expr interface:
  eval     = e1.eval + e2.eval
  simplify = ...
  pprint   = ...
```

OO languages, continued

Adding a new construct to the expression language:

```
class Prod implements Expr
  where
  -- specific to Prod:
  e1       :: Expr
  e2       :: Expr
  -- Expr interface:
  eval     = e1.eval * e2.eval
  simplify = ...
  pprint   = ...
```

OO languages, continued

Adding a new construct to the expression language:

```
class Prod implements Expr
  where
  -- specific to Prod:
  e1       :: Expr
  e2       :: Expr
  -- Expr interface:
  eval     = e1.eval * e2.eval
  simplify = ...
  pprint   = ...
```

This is **easy**, because it is **modular**: there is no need to change code that has already been written.

OO languages, continued

Adding a new operation on expressions:

- change class Expr to add the new operation as a method

OO languages, continued

Adding a new operation on expressions:

- change class Expr to add the new operation as a method
- change class Num to add the new operation and its implementation

OO languages, continued

Adding a new operation on expressions:

- change class Expr to add the new operation as a method
- change class Num to add the new operation and its implementation
- change class Sum to add the new operation and its implementation

OO languages, continued

Adding a new operation on expressions:

- change class Expr to add the new operation as a method
- change class Num to add the new operation and its implementation
- change class Sum to add the new operation and its implementation
- change class Prod to add the new operation and its implementation

OO languages, continued

Adding a new operation on expressions:

- change class Expr to add the new operation as a method
- change class Num to add the new operation and its implementation
- change class Sum to add the new operation and its implementation
- change class Prod to add the new operation and its implementation

This is **difficult**, because the changes are non-local and have to be made in code that has already been written. In particular, the Expr class cannot be shipped as a library.

FP languages

In functional programming languages, this is an idiomatic way to model the problem:

- there is a **data type** of expressions,
- different constructs of the expression language are **data constructors** of the data type,
- the operations on expressions (such as evaluation, pretty-printing, ...) are **functions** the process values of the data type

FP languages, continued

data Expr **where**
  Num :: Int → Expr
  Sum :: Expr → Expr → Expr

FP languages, continued

```
data Expr where
  Num :: Int → Expr
  Sum :: Expr → Expr → Expr
```

```
eval :: Expr → Int
eval (Num n)       = n
eval (Sum e₁ e₂) = e₁ + e₂
```

FP languages, continued

```
data Expr where
  Num :: Int → Expr
  Sum :: Expr → Expr → Expr
```

```
eval :: Expr → Int
eval (Num n)       = n
eval (Sum e₁ e₂) = e₁ + e₂
```

```
pprint :: Expr → String
pprint (Num n)       = show n
pprint (Sum e₁ e₂) = "(" ++ pprint e₁ ++ " + " ++ pprint e₂ ++ ")"
```

FP languages, continued

Adding a new operation on expressions:

```
simplify :: Expr → Expr
simplify (Sum e₁ e₂) = let s₁ = simplify e₁
                           s₂ = simplify e₂
                       in  case (s₁, s₂)
                           of   (Num 0, _    ) → Sum s₂
                                (_    , Num 0) → Sum s₁
                                _              → Sum s₁ s₂
simplify e             = e
```

FP languages, continued

Adding a new operation on expressions:

$$
\begin{array}{l}
\mathrm{simplify} :: \mathrm{Expr} \rightarrow \mathrm{Expr} \\
\mathrm{simplify}\ (\mathrm{Sum}\ e_1\ e_2) = \textbf{let}\ s_1 = \mathrm{simplify}\ e_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s_2 = \mathrm{simplify}\ e_2 \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{case}\ (s_1, s_2) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{of}\ \ (\mathrm{Num}\ 0, \_) \rightarrow \mathrm{Sum}\ s_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\_, \mathrm{Num}\ 0) \rightarrow \mathrm{Sum}\ s_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \rightarrow \mathrm{Sum}\ s_1\ s_2 \\
\mathrm{simplify}\ e = e
\end{array}
$$

This is **easy**, because it is **modular**: there is no need to change code that has already been written.

FP languages, continued

Adding a new construct to the expression language:

FP languages, continued

Adding a new construct to the expression language:

- change data type Expr to add a new data constructor

FP languages, continued

Adding a new construct to the expression language:

- change data type Expr to add a new data constructor
- change function eval to add an equation for the new constructor

FP languages, continued

Adding a new construct to the expression language:

- change data type Expr to add a new data constructor
- change function eval to add an equation for the new constructor
- change function pprint to add an equation for the new constructor

FP languages, continued

Adding a new construct to the expression language:

- change data type Expr to add a new data constructor
- change function eval to add an equation for the new constructor
- change function pprint to add an equation for the new constructor
- change function simplify to add an equation for the new constructor

FP languages, continued

Adding a new construct to the expression language:

- change data type Expr to add a new data constructor
- change function eval to add an equation for the new constructor
- change function pprint to add an equation for the new constructor
- change function simplify to add an equation for the new constructor

This is **difficult**, because the changes are non-local and have to be made in code that has already been written. In particular, the Expr class cannot be shipped as a library.

Intermediate summary

- OO languages support extension of data, but not of functionality.
- FP languages support extension of functionality, but not of data.

Intermediate summary

- OO languages support extension of data, but not of functionality.
- FP languages support extension of functionality, but not of data.

It seems to be difficult to support both directions of extension described in the expression problem at the same time.

Intermediate summary

- OO languages support extension of data, but not of functionality.
- FP languages support extension of functionality, but not of data.

It seems to be difficult to support both directions of extension described in the expression **problem** at the same time.

The visitor pattern

Using the visitor pattern, we can simulate the functional program in an OO language:

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Expr}_{\mathrm{Visitor}}\ a\ \textbf{where} \\
\quad \mathrm{visit}_{\mathrm{Num}}\ ::\ \mathrm{Num} \rightarrow a \\
\quad \mathrm{visit}_{\mathrm{Sum}}\ ::\ \mathrm{Sum} \rightarrow a \\
\quad \mathrm{visit}_{\mathrm{Prod}}\ ::\ \mathrm{Prod} \rightarrow a
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Expr}\ \textbf{where} \\
\quad \mathrm{accept}\ ::\ \mathrm{Expr}_{\mathrm{Visitor}}\ a \rightarrow a \\
\textbf{class}\ \mathrm{Num}\ \textbf{implements}\ \mathrm{Expr}\ \textbf{where} \\
\quad \mathrm{val}\ ::\ \mathrm{Int} \\
\quad \mathrm{accept}\ v = v.\mathrm{visit}_{\mathrm{Num}}\ \textbf{self} \\
\textbf{class}\ \mathrm{Sum}\ \textbf{implements}\ \mathrm{Expr}\ \textbf{where} \\
\quad e_1, e_2\ ::\ \mathrm{Expr} \\
\quad \mathrm{accept}\ v = v.\mathrm{visit}_{\mathrm{Sum}}\ \textbf{self} \\
\textbf{class}\ \mathrm{Prod}\ \textbf{implements}\ \mathrm{Expr}\ \textbf{where} \\
\quad e_1, e_2\ ::\ \mathrm{Expr} \\
\quad \mathrm{accept}\ v = v.\mathrm{visit}_{\mathrm{Prod}}\ \textbf{self}
\end{array}
$$

The visitor pattern, continued

class $\text{Eval}_{\text{Visitor}}$ **implements** $\text{Expr}_{\text{Visitor}}$ **where**

$\text{visit}_{\text{Num}}\ x = x.\text{val}$

$\text{visit}_{\text{Sum}}\ x = x.e_1.\text{accept}\ \textbf{self} + x.e_2.\text{accept}\ \textbf{self}$

$\text{visit}_{\text{Prod}}\ x = x.e_1.\text{accept}\ \textbf{self} * x.e_2.\text{accept}\ \textbf{self}$

The visitor pattern, continued

$\textbf{class}\ \text{Eval}_{\text{Visitor}}\ \textbf{implements}\ \text{Expr}_{\text{Visitor}}\ \textbf{where}$
$\quad \text{visit}_{\text{Num}}\ x = x.\text{val}$
$\quad \text{visit}_{\text{Sum}}\ x = x.e_1.\text{accept}\ \textbf{self} + x.e_2.\text{accept}\ \textbf{self}$
$\quad \text{visit}_{\text{Prod}}\ x = x.e_1.\text{accept}\ \textbf{self} * x.e_2.\text{accept}\ \textbf{self}$

$\textbf{class}\ \text{Simplify}_{\text{Visitor}}\ \textbf{implements}\ \text{Expr}_{\text{Visitor}}\ \textbf{where}$
$\quad \text{simplify}_{\text{Num}} \ldots$
$\quad \text{simplify}_{\text{Sum}} \ldots$
$\quad \text{simplify}_{\text{Prod}} \ldots$

Type classes

Using type classes, we can simulate the OO program in a functional language:

```
class Expr a where
  eval     :: a → Result
  simplify :: a → Expr
  pprint   :: a → String
```

```
data Num = Num Int
instance Expr Num
where
  eval     (Num val) = val
  simplify ...
  pprint   ...
```

Type classes

Using type classes, we can simulate the OO program in a functional language:

```
class Expr a where
  eval     :: a → Result
  simplify :: a → Expr
  pprint   :: a → String
```

```
data Num = Num Int
instance Expr Num
where
  eval     (Num val) = val
  simplify ...
  pprint   ...
```

```
data Sum a b = Suma b
instance(Expr a, Expr b) ⇒
        Expr (Sum a b)
where
  eval     e1 e2 = eval e1 + eval e2
  simplify ...
  pprint   ...
```

Expression problem, again

If the direction of extensibility is not supported by our language of choice, there is usually an encoding of our program that supports the other direction, but

- it again provides only one direction of extensibility (now the other) at the time,
- it is somewhat non-idiomatic (but: design patterns),
- it is more verbose,
- we have to decide in the very beginning which form of extensibility is desired.

Proper solutions

There are, by now, many solutions to the expression problem:

- most for OO languages, some for FP languages
- varying degrees of complexity
- often require language extensions
- support available in some modern languages
- no light-weight, readily available solution for FP languages

Goals

- Add open data types to Haskell (or possibly other FP languages).
- Open functions are also required.
- As simple as possible.
- Inspiration from Haskell type classes.

Overview

Syntax: open data types

open Expr :: $*$

Syntax: open data types

$$\textbf{open}\ \mathsf{Expr} :: *$$

$$\mathsf{Num} :: \mathsf{Int} \rightarrow \mathsf{Expr}$$

Syntax: open data types

open Expr :: $*$

Num :: Int $\to$ Expr

Sum :: Expr $\to$ Expr $\to$ Expr

Syntax: open data types

open Expr :: $*$

Num :: Int $\rightarrow$ Expr

Sum :: Expr $\rightarrow$ Expr $\rightarrow$ Expr

Prod :: Expr $\rightarrow$ Expr $\rightarrow$ Expr

Syntax: open data types

$\textbf{open}\ \mathsf{Expr} :: *$

$\mathsf{Num} :: \mathsf{Int} \to \mathsf{Expr}$

$\mathsf{Sum} :: \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}$

$\mathsf{Prod} :: \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}$

- Additional constructors can be added at any time and any place of the program.
- Once we have open data types, we need open functions, too. (Question: why?)

Syntax: open functions

$$
\begin{aligned}
&\mathsf{eval} :: \mathsf{Expr} \rightarrow \mathsf{Int}\\
&\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad = \mathsf{n}\\
&\mathsf{eval}\ (\mathsf{Sum}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 + \mathsf{e}_2
\end{aligned}
$$

Syntax: open functions

$$
\begin{array}{l}
\textbf{open}\ \mathsf{eval} :: \mathsf{Expr} \to \mathsf{Int} \\
\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad = \mathsf{n} \\
\mathsf{eval}\ (\mathsf{Sum}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 + \mathsf{e}_2
\end{array}
$$

Syntax: open functions

$$
\begin{aligned}
&\textbf{open}\ \mathsf{eval} :: \mathsf{Expr} \rightarrow \mathsf{Int} \\
&\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad = \mathsf{n} \\
&\mathsf{eval}\ (\mathsf{Sum}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 + \mathsf{e}_2
\end{aligned}
$$

$$
\mathsf{eval}\ (\mathsf{Prod}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 * \mathsf{e}_2
$$

Syntax: open functions

$$
\begin{aligned}
&\textbf{open}\ \mathsf{eval} :: \mathsf{Expr} \rightarrow \mathsf{Int} \\
&\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad = \mathsf{n} \\
&\mathsf{eval}\ (\mathsf{Sum}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 + \mathsf{e}_2
\end{aligned}
$$

$$
\mathsf{eval}\ (\mathsf{Prod}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{e}_1 * \mathsf{e}_2
$$

- Additional equations can be added at any time and any place of the program.

Overview

A type of type representations

$$
\begin{array}{ll}
\textbf{open data}\ \mathsf{Type} :: * \to * & \\
\mathsf{Int} & :: \mathsf{Type\ Int} \\
\mathsf{Char} & :: \mathsf{Type\ Char} \\
\mathsf{Unit} & :: \mathsf{Type}\ () \\
\mathsf{Pair} & :: \mathsf{Type\ a} \to \mathsf{Type\ b} \to \mathsf{Type}\ (\mathsf{a}, \mathsf{b}) \\
\mathsf{Either} & :: \mathsf{Type\ a} \to \mathsf{Type\ b} \to \mathsf{Type}\ (\mathsf{Either\ a\ b}) \\
\mathsf{List} & :: \mathsf{Type\ a} \to \mathsf{Type}\ [\mathsf{a}]
\end{array}
$$

A type of type representations

```
open data Type :: * → *
Int    :: Type Int
Char   :: Type Char
Unit   :: Type ()
Pair   :: Type a → Type b → Type (a, b)
Either :: Type a → Type b → Type (Either a b)
List   :: Type a → Type [a]
```

```
data Either :: * → * → * where
  Left  :: a → Either a b
  Right :: b → Either a b
data [] :: * → * where
  []    :: [a]
  (:)   :: a → [a] → [a]
```

A type of type representations

```
open data Type :: * → *
Int    :: Type Int
Char   :: Type Char
Unit   :: Type ()
Pair   :: Type a → Type b → Type (a, b)
Either :: Type a → Type b → Type (Either a b)
List   :: Type a → Type [a]
```

```
data Either :: * → * → * where
  Left  :: a → Either a b
  Right :: b → Either a b
data [] :: * → * where
  []    :: [a]
  (:)   :: a → [a] → [a]
```

Note: The data type Type is a **generalized algebraic data type**.

An overloaded equality function

```
open eq :: Type a → a → a → Bool
eq Int            x          y          = x == y   -- use built-in
eq Char           x          y          = x == y   -- use built-in
eq (Pair a b)     (x1, x2)   (y1, y2)   = eq a x1 x2 ∧ eq b y1 y2
eq (Either a b)   (Left x)   (Left y)   = eq a x y
eq (Either a b)   (Right x)  (Right y)  = eq b x y
eq (Either a b)   _          _          = False
eq (List a)       xs         ys         = and (zipWith (eq a) xs ys)
```

An overloaded equality function

```
open eq :: Type a → a → a → Bool
eq Int          x          y          = x == y   -- use built-in
eq Char         x          y          = x == y   -- use built-in
eq (Pair a b)   (x₁, x₂)   (y₁, y₂)   = eq a x₁ x₂ ∧ eq b y₁ y₂
eq (Either a b) (Left x)   (Left y)   = eq a x y
eq (Either a b) (Right x)  (Right y)  = eq b x y
eq (Either a b) _          _          = False
eq (List a)     xs         ys         = and (zipWith (eq a) xs ys)
```

Let us turn this function into a generic function:

```
eq a x y = case view a of View a′ from to → eq a′ (from x) (from y)
data View :: ∗ → ∗ where
  View :: a′ → (a → a′) → (a′ → a) → View a
```

Viewing a type as its structural representation

The function view is another overloaded open function:

```
open view :: Type a → View a
```

How to view lists as a sum of a product:

```
data [] :: ∗ → ∗ where
  []  :: [a]
  (:) :: a → [a] → [a]
type List′ a = Either () (a, [a])
fromList :: [a] → List′ a
fromList []       = Left ()
fromList (x : xs) = Right (x, xs)
toList :: List′ a → [a]
toList (Left ())        = []
toList (Right (x, xs)) = x : xs
view (List a) = View (Either Unit (Pair a (List a))) fromList toList
```

Generic equality, again

```
open eq :: Type a → a → a → Bool
eq Int          x          y          = x == y   -- use built-in
eq Char         x          y          = x == y   -- use built-in
eq (Pair a b)   (x₁, x₂)   (y₁, y₂)   = eq a x₁ x₂ ∧ eq b y₁ y₂
eq (Either a b) (Left x)   (Left y)   = eq a x y
eq (Either a b) (Right x)  (Right y)  = eq b x y
eq (Either a b) _          _          = False
eq (List a)     xs         ys         = and (zipWith (eq a) xs ys)
eq a x y = case view a of View a' from to → eq a' (from x) (from y)
```

Generic equality, again

```
open eq :: Type a → a → a → Bool
eq Int          x          y          = x == y   -- use built-in
eq Char         x          y          = x == y   -- use built-in
eq (Pair a b)   (x₁, x₂)   (y₁, y₂)   = eq a x₁ x₂ ∧ eq b y₁ y₂
eq (Either a b) (Left x)   (Left y)   = eq a x y
eq (Either a b) (Right x)  (Right y)  = eq b x y
eq (Either a b) _          _          = False
eq (List a)     xs         ys         = and (zipWith (eq a) xs ys)
eq a x y = case view a of View a′ from to → eq a′ (from x) (from y)
```

- The case for List is now subsumed by the generic case.
- We can add more data types, because the definitions are open . . .

Viewing Booleans

Add a new constructor for representations of Booleans:

```
Bool :: Type Bool
```

Viewing Booleans

Add a new constructor for representations of Booleans:

$$\mathsf{Bool} :: \mathsf{Type}\ \mathsf{Bool}$$

Add a new equation to the definition of view:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Bool} :: * \ \textbf{where} \\
\quad \mathsf{False} :: \mathsf{Bool} \\
\quad \mathsf{True}\ \ :: \mathsf{Bool} \\
\textbf{type}\ \mathsf{Bool}'\ \mathsf{a} = \mathsf{Either}\ ()\ () \\
\mathsf{fromBool} :: \mathsf{Bool} \rightarrow \mathsf{Bool}' \\
\mathsf{fromBool}\ \mathsf{False} = \mathsf{Left}\ () \\
\mathsf{fromBool}\ \mathsf{True}\ \ = \mathsf{Right}\ () \\
\mathsf{toBool} :: \mathsf{Bool}' \rightarrow \mathsf{Bool} \\
\mathsf{toBool}\ (\mathsf{Left}\ ())\ \ \ = \mathsf{False} \\
\mathsf{toBool}\ (\mathsf{Right}\ ()) = \mathsf{True} \\
\mathsf{view}\ (\mathsf{Bool}\ \mathsf{a}) = \mathsf{View}\ (\mathsf{Either}\ \mathsf{Unit}\ \mathsf{Unit})\ \mathsf{fromBool}\ \mathsf{toBool}
\end{array}
$$

Intermediate summary

- With an open type of type representations, we can add a new constructor for each data type.
- With an open view function, we can add a way to view each data type as its structural representation.
- Then **all** generic functions automatically work for the added data type.
- If the generic functions are also open, we can add new specific behaviour (if a data type has a non-standard definition of equality, for example).

An interface for exceptions

$$\begin{aligned} \mathsf{throw} &:: \mathsf{Exception} \to \mathsf{a} \\ \mathsf{catch} &:: \mathsf{IO\ a} \to (\mathsf{Exception} \to \mathsf{IO\ a}) \to \mathsf{IO\ a} \end{aligned}$$

An interface for exceptions

$$
\begin{aligned}
\text{throw} &:: \text{Exception} \rightarrow \text{a} \\
\text{catch} &:: \text{IO a} \rightarrow (\text{Exception} \rightarrow \text{IO a}) \rightarrow \text{IO a}
\end{aligned}
$$

- In Haskell, the type Exception is a library type with several predefined constructors for frequent errors.
- If an application-specific error arises (for example: an illegal key is passed to a finite map lookup), we must try to find a close match among the predefined constructors.
- OCaml has a special construct for extensible exceptions, and extensible exceptions have been proposed multiple times for Haskell, too.

An interface for exceptions

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

- In Haskell, the type Exception is a library type with several predefined constructors for frequent errors.
- If an application-specific error arises (for example: an illegal key is passed to a finite map lookup), we must try to find a close match among the predefined constructors.
- OCaml has a special construct for extensible exceptions, and extensible exceptions have been proposed multiple times for Haskell, too.
- With open data types, there is no need for a special construct.

An open data type for exceptions

open data Exception :: $*$

Declaring a new exception:

KeyNotFound :: Key $\rightarrow$ Exception

An open data type for exceptions

open data Exception :: $*$

Declaring a new exception:

KeyNotFound :: Key $\rightarrow$ Exception

Raising the exception:

lookup k fm $=$ $\ldots$ throw (KeyNotFound k) $\ldots$

An open data type for exceptions

$\textbf{open data}\ \text{Exception} :: *$

Declaring a new exception:

$\text{KeyNotFound} :: \text{Key} \rightarrow \text{Exception}$

Raising the exception:

$\text{lookup k fm} = \ldots \text{throw}\ (\text{KeyNotFound k}) \ldots$

Catching the exception:

$$\begin{array}{l}
\text{catch}\ (\ldots) \\
\quad (\lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad \text{KeyNotFound k} \rightarrow \ \ldots \\
\qquad\qquad _ \qquad\qquad\qquad\ \rightarrow \text{return (throw e))}
\end{array}$$

Note: We have to re-raise the exception at the end of the handler.

Overview

Semantics: basic idea

- Collapse everything into a single module.
- Basically the same as we would have written in a closed setting.

Problems

- Local functions?
- Module system?
- Pattern matching?

Solutions

“Learn” from type classes.

- Local functions?
  Local open functions are not allowed.
- Module system?
  Open functions cannot be hidden selectively.
- Pattern matching?
  Best-fit pattern matching for open functions.

Pattern matching

The function view is very nice, because it has non-overlapping patterns.
What if we extend a function that has overlapping patterns?

Pattern matching

The function view is very nice, because it has non-overlapping patterns. What if we extend a function that has overlapping patterns?

- a variable pattern is a worse fit than a constructor pattern
- use the best fit (not the first)
- for multiple patterns, use a left-to-right bias
- this allows the programmer to add default equations early (such as the general case in eq)

Example of best-fit pattern matching

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

Example of best-fit pattern matching

```
f :: [Int] → Either Int Char → ...
f (x : xs) (Left 1)
f y        (Right a)
f (0 : xs) (Right 'X')
f [1]      z
f [0]      z
f []       z
f [0]      (Left b)
f [0]      (Left 2)
f y        z
f [x]      z
```

```
f :: [Int] → Either Int Char → ...
f []       z
f [0]      (Left 2)
f [0]      (Left b)
f [0]      z
f (0 : xs) (Right 'X')
f [1]      z
f [x]      z
f (x : xs) (Left 1)
f y        (Right a)
f y        z
```

Overview

A naïve implementation

- Like semantics: collapse program into a single module.
- Advantage: easy to implement, correct by construction.
- Big disadvantage: no separate compilation; inefficient compilation for large programs.
- Resulting programs are still efficient.

Implementation with separate compilation

- All open data types, and the pattern match logic of open functions are placed into a special module Closure.
- The module Closure must be recompiled whenever any open data type or open function changes.
- The rest of the program is translated module by module. Each module imports Closure, but only uses a small part of it (made explicit in an interface). Only if the interface or the module itself changes, the module has to be recompiled.
- Advantage: allows separate compilation (mostly).
- Disadvantage: slightly trickier to implement (but only a small extension to GHC would be required).

Overview

Conclusions

- Very simple solution: no changes to the type system, no deep semantics.
- Flagging a data type or a function as open is not a wide-reaching design decision, but a minor local syntactic change.
- One easy implementation, one relatively efficient implementation.
- Lots of related work, but most aim at solving a more complex problem.
- Our approach applies to many interesting examples.
- Many properties of type classes used (some restrictions, too).
- More properties of type classes could be transferred:
  - Partial evaluation of pattern matching.
  - Automatic inference of uniquely determined values.