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
Consider a small language of expressions:

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

Providing both directions of extensibility is known as the expression problem.

How do programming languages support these different forms of program evolution?
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 as the **expression problem**.

How do programming languages support these different forms of program evolution?
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 as 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 as the expression problem.

How do programming languages support these different forms of program evolution?
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
class Expr where
    eval :: Result
    simplify :: Expr
    pprint :: String
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 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 = ...
Adding a new construct to the expression language:

```haskell
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.
Adding a new construct to the expression language:

```haskell
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.
Adding a new operation on expressions:

- change class `Expr` to add the new operation as a method
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
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
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
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.
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.
data Expr where
  Num :: Int → Expr
  Sum :: Expr → Expr → Expr
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

```haskell
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 ++ ")"
```
Adding a new operation on expressions:

\[
\text{simplify} :: \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{simplify} (\text{Sum } e_1 e_2) = \text{let } s_1 = \text{simplify } e_1 \\
\hspace{1cm} s_2 = \text{simplify } e_2 \\
\hspace{1cm} \text{in case } (s_1, s_2) \\
\hspace{2cm} \text{of } (\text{Num } 0, \_ ) \rightarrow \text{Sum } s_2 \\
\hspace{2cm} (\_ , \text{Num } 0) \rightarrow \text{Sum } s_1 \\
\hspace{2cm} \_ \_ \rightarrow \text{Sum } s_1 s_2
\]

\[
\text{simplify } e = e
\]
Adding a new operation on expressions:

\[
\text{simplify} :: \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{simplify} (\text{Sum } e_1 e_2) = \text{let } s_1 = \text{simplify } e_1 \\
\hspace{1cm} s_2 = \text{simplify } e_2 \\
\hspace{1cm} \text{in } \text{case } (s_1, s_2) \text{ of }\\
\hspace{2cm} (\text{Num } 0, _) \rightarrow \text{Sum } s_2 \\
\hspace{2cm} (_, \text{Num } 0) \rightarrow \text{Sum } s_1 \\
\hspace{2cm} _- \rightarrow \text{Sum } s_1 s_2
\]

\[
\text{simplify } e = e
\]

This is \textbf{easy}, because it is \textbf{modular}: there is no need to change code that has already been written.
Adding a new construct to the expression language:
Adding a new construct to the expression language:

- change data type `Expr` to add a new data constructor
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
Adding a new construct to the expression language:

- change data type \texttt{Expr} to add a new data constructor
- change function \texttt{eval} to add an equation for the new constructor
- change function \texttt{pprint} 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 \texttt{Expr} class cannot be shipped as a library.
Adding a new construct to the expression language:

- change data type \texttt{Expr} to add a new data constructor
- change function \texttt{eval} to add an equation for the new constructor
- change function \texttt{pprint} to add an equation for the new constructor
- change function \texttt{simplify} to add an equation for the new constructor
Adding a new construct to the expression language:

- change data type \texttt{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 \textbf{difficult}, because the changes are non-local and have to be made in code that has already been written. In particular, the \texttt{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:

```haskell
class ExprVisitor a where
  visitNum :: Num → a
  visitSum :: Sum → a
  visitProd :: Prod → a

class Expr where
  accept :: ExprVisitor a → a

class Num implements Expr where
  val :: Int
  accept v = v.visitNum self

class Sum implements Expr where
  e1, e2 :: Expr
  accept v = v.visitSum self

class Prod implements Expr where
  e1, e2 :: Expr
  accept v = v.visitProd self
```
The visitor pattern, continued

class EvalVisitor implements ExprVisitor where
    visit Num x = x.val
    visit Sum x = x.e1.accept self + x.e2.accept self
    visit Prod x = x.e1.accept self * x.e2.accept self
class `EvalVisitor` implements `ExprVisitor` where

visit\_Num x = x.val
visit\_Sum x = x.e\_1.accept self + x.e\_2.accept self
visit\_Prod x = x.e\_1.accept self * x.e\_2.accept self

class `SimplifyVisitor` implements `ExprVisitor` where

simplify\_Num . . .
simplify\_Sum . . .
simplify\_Prod . . .
Using type classes, we can simulate the OO program in a functional language:

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

```haskell
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:

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

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

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

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
Syntax: open data types

open Expr :: *
Syntax: open data types

open Expr :: *

Num :: Int → 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 data types

open Expr :: *

Num :: Int → Expr

Sum :: Expr → Expr → Expr
Syntax: open data types

| open Expr :: * |
| Num :: Int → Expr |
| Sum :: Expr → Expr → Expr |
| Prod :: Expr → Expr → Expr |
Syntax: open data types

open Expr :: *

Num :: Int → Expr

Sum :: Expr → Expr → Expr

Prod :: Expr → Expr → 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

eval :: Expr → Int

\[
eval \ (\text{Num} \ n) = n \\
eval \ (\text{Sum} \ e_1 \ e_2) = e_1 + e_2
\]
open eval :: Expr → Int

eval (Num n) = n

eval (Sum e₁ e₂) = e₁ + e₂
Syntax: open functions

\textbf{open} eval :: \textit{Expr} \rightarrow \textit{Int}\\
eval (\textit{Num} n) = n\\
eval (\textit{Sum} e_1 e_2) = e_1 + e_2\\

\textbf{eval} (\textit{Prod} e_1 e_2) = e_1 * e_2
Syntax: open functions

\textbf{open} eval :: Expr \rightarrow \text{Int}

\begin{align*}
\text{eval (Num } n) &= n \\
\text{eval (Sum } e_1 e_2) &= e_1 + e_2
\end{align*}

\text{eval (Prod } e_1 e_2) &= e_1 * e_2

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

```haskell
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 \( a \rightarrow a \rightarrow a \rightarrow \text{Bool} \)

- \( \text{eq } \text{Int } x \ y = x == y \) -- use built-in
- \( \text{eq } \text{Char } x \ y = x == y \) -- use built-in
- \( \text{eq } (\text{Pair } a \ b) \ (x_1, x_2) \ (y_1, y_2) = \text{eq } a \ x_1 \ x_2 \land \text{eq } b \ y_1 \ y_2 \)
- \( \text{eq } (\text{Either } a \ b) \ (\text{Left } x) \ (\text{Left } y) = \text{eq } a \ x \ y \)
- \( \text{eq } (\text{Either } a \ b) \ (\text{Right } x) \ (\text{Right } y) = \text{eq } b \ x \ y \)
- \( \text{eq } (\text{Either } a \ b) \_ \_ = \text{False} \)
- \( \text{eq } (\text{List } a) \ xs \ ys = \text{and } (\text{zipWith } (\text{eq } a) \ xs \ ys) \)

Let us turn this function into a generic function:

- \( \text{eq } a \times y = \text{case } \text{view } a \ \text{of } \text{View } a' \ \text{from } \text{to } \ightarrow \text{eq } a' \ (\text{from } x) \ (\text{from } y) \)

**data View ::** \( \ast \rightarrow \ast \) **where**

- \( \text{View :: } a' \rightarrow (a \rightarrow a') \rightarrow (a' \rightarrow a) \rightarrow \text{View } a \)
Viewing a type as its structural representation

The function view is another overloaded open function:

```haskell
open view :: Type a → View a
```

How to view lists as a sum of a product:

```haskell
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 \text{eq} :: \text{Type} a \rightarrow a \rightarrow a \rightarrow \text{Bool}

\text{eq Int} \quad x \quad y \quad = \ x == y \quad -- \text{use built-in}

\text{eq Char} \quad x \quad y \quad = \ x == y \quad -- \text{use built-in}

\text{eq (Pair} a \ b) \quad (x_1, x_2) \quad (y_1, y_2) \quad = \ \text{eq} a \ x_1 \ x_2 \land \text{eq} b \ y_1 \ y_2

\text{eq (Either} a \ b) \quad (\text{Left} \ x) \quad (\text{Left} \ y) \quad = \ \text{eq} a \ x \ y

\text{eq (Either} a \ b) \quad (\text{Right} \ x) \quad (\text{Right} \ y) \quad = \ \text{eq} b \ x \ y

\text{eq (Either} a \ b) \quad _ \quad _ \quad = \ \text{False}

\text{eq (List} a) \quad xs \quad ys \quad = \ \text{and} \ (\text{zipWith} \ (\text{eq} a) \ xs \ ys)

\text{eq a x y} = \text{case} \ \text{view} \ a \ \text{of} \ \text{View} \ a' \ \text{from} \ \text{to} \rightarrow \ \text{eq} a' \ (\text{from} \ x) \ (\text{from} \ y)
Generic equality, again

\textbf{open eq :: Type} \ a \ → \ a \ → \ a \ → \ \text{Bool}

\text{eq Int \ x \ y = \ x == \ y} \quad \text{-- use built-in}
\text{eq Char \ x \ y = \ x == \ y} \quad \text{-- use built-in}

\text{eq (Pair} \ a \ b) (x_1, x_2) (y_1, y_2) = \text{eq} \ a \ x_1 \ x_2 \land \text{eq} \ b \ y_1 \ y_2

\text{eq (Either} \ a \ b) (\text{Left} \ x) (\text{Left} \ y) = \text{eq} \ a \ x \ y
\text{eq (Either} \ a \ b) (\text{Right} \ x) (\text{Right} \ y) = \text{eq} \ b \ x \ y
\text{eq (Either} \ a \ b) _ _ = \text{False}

\text{eq (List} \ a) \ xs \ ys = \text{and (zipWith} \ \text{eq} \ a \ xs \ ys
\text{eq} \ a \ x \ y = \text{case view} \ a \ \text{of View} \ a' \ \text{from} \ \text{to} \ → \ \text{eq} \ a' \ \text{(from} \ x) \ \text{(from} \ y)

\begin{itemize}
  \item The case for \text{List} is now subsumed by the generic case.
  \item We can add more data types, because the definitions are open \ldots
\end{itemize}
Viewing Booleans

Add a new constructor for representations of Booleans:

```plaintext
| Bool :: Type Bool |
```
Viewing Booleans

Add a new constructor for representations of Booleans:

```plaintext
   Bool :: Type Bool
```

Add a new equation to the definition of view:

```plaintext
data Bool :: * where
   False :: Bool
   True :: Bool

type Bool' a = Either () ()
fromBool :: Bool → Bool'
fromBool False = Left ()
fromBool True = Right ()
toBool :: Bool' → Bool
toBool (Left ()) = False
toBool (Right ()) = True
view (Bool a) = View (Either Unit Unit) fromBool toBool
```
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

\[
\text{throw :: } \text{Exception} \rightarrow a \\
\text{catch :: } \text{IO } a \rightarrow (\text{Exception} \rightarrow \text{IO } a) \rightarrow \text{IO } a
\]
An interface for exceptions

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

- In Haskell, the type \textit{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

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

- In Haskell, the type \text{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 → Exception

Raising the exception:

lookup k fm = . . . throw (KeyNotFound k) . . .

Catching the exception:

catch (. . .) (λ e → case e of KeyNotFound k → . . . → return (throw e))

Note: We have to re-raise the exception at the end of the handler.
An open data type for exceptions

| open data Exception :: *

Declaring a new exception:

| KeyNotFound :: Key → Exception

Raising the exception:

| lookup k fm = . . . throw (KeyNotFound k) . . .
An open data type for exceptions

**open data** Exception :: *

Declaring a new exception:

**KeyNotFound** :: Key → Exception

Raising the exception:

lookup k fm = ... throw (KeyNotFound k) ...

Catching the exception:

catch (...) 
  (λe → case e of 
    KeyNotFound k → ... 
    _ → return (throw e))

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

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

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

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

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

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

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

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
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.