This talk

- Add open data types and open functions to Haskell.
- Keep it as simple as possible!
- Many of the design decisions (and restrictions) are inspired by Haskell’s type classes.
Overview

1 The language extension
   • Data types
   • Functions
   • Pattern matching

2 Applications
   • Expression problem
   • Generic programming
   • Exceptions

3 Implementation
   • Directly
   • Separate compilation

4 Conclusions
Data types in Haskell

```haskell
data Expr =
    Num Int
  | Sum Expr Expr
  | Prod Expr Expr
  | Neg Expr
```

- Each data type comprises a number of constructors.
- Each constructor can have arguments.
- All constructors have to be declared at once, while declaring the data type.
Generalized algebraic data types in Haskell use the following syntax:

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

- Equivalent to the declaration on the previous slide.
- The signature of the data type is given.
- Each constructor is accompanied by its type signature.
open data \texttt{Expr} :: ∗

This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where \texttt{Expr} is in scope).
open data Expr :: *

This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where Expr is in scope).

Num :: Int → Expr
open data Expr :: *

This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where Expr is in scope).

Num :: Int → Expr

Sum :: Expr → Expr → Expr
open data \texttt{Expr} :: \(*\)

- This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where \texttt{Expr} is in scope).

\begin{itemize}
  \item \texttt{Num} :: \texttt{Int} → \texttt{Expr}
  \item \texttt{Prod} :: \texttt{Expr} → \texttt{Expr} → \texttt{Expr}
  \item \texttt{Sum} :: \texttt{Expr} → \texttt{Expr} → \texttt{Expr}
\end{itemize}
open data Expr :: *

This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where Expr is in scope).

Num :: Int → Expr
Sum :: Expr → Expr → Expr
Prod :: Expr → Expr → Expr
Neg :: Expr → Expr
open data Expr :: *

- This declares a new open data type of the given kind. No constructors need to be given at the point of declaration, but can instead appear anywhere in the program (where Expr is in scope).

Num :: Int → Expr
Sum :: Expr → Expr → Expr
Prod :: Expr → Expr → Expr
Neg :: Expr → Expr

- The result type indicates the data type the constructor belongs to.
- The program behaves as if the data type had been defined closed, in a single place.
- We can now grow the expression language in multiple steps.
- Once we have open data types, we need open functions, too . . .
Open functions

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

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

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

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

If we now extend Expr, we have to extend eval as well:

Prod :: Expr → Expr → Expr
Open functions

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

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

If we now extend Expr, we have to extend eval as well:

Prod :: Expr → Expr → Expr

eval (Prod e₁ e₂) = eval e₁ * eval e₂
Open functions

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

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

If we now extend Expr, we have to extend eval as well:

Prod :: Expr → Expr → Expr

eval (Prod e₁ e₂) = eval e₁ * eval e₂

The open keyword declares a function to be open.
New equations can be given at any point in the program.
The program behaves as if the function had been defined closed, in a single place (in particular, recursive calls always refer to the full function).
What about pattern matching?
Pattern matching

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

```haskell
open simplify :: Expr → Expr

... simplify (Sum (Num 0) e₂) = simplify e₂
...

simplify e = e
```

In Haskell, the order of function equations is significant, this is not suitable for open functions.
The function eval is very nice so far, because it has non-overlapping patterns. What if we extend a function that has overlapping patterns?

```
open simplify :: Expr -> Expr

simplify (Sum (Num 0) e2) = simplify e2

simplify e = e
```

- in Haskell, the order of function equations is significant
- this is not suitable for open functions
Best-fit pattern matching

- inspired by Haskell’s resolution of overlapping instances
- 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
- order of equations is no longer significant
- default equations (such as for simplify can be added early)
Example of best-fit pattern matching

The equations of a function can always be reordered such that first-fit and best-fit pattern matching semantics coincide:

\[
\begin{align*}
  f :: [\text{Int}] &\rightarrow \text{Either Int Char} &\rightarrow \ldots \\
  f (x:xs) &\rightarrow \text{Left 1} \\
  f y &\rightarrow \text{Right a} \\
  f (0:xs) &\rightarrow \text{Right 'X'} \\
  f (1:[]) &\rightarrow z \\
  f (0:[]) &\rightarrow z \\
  f [] &\rightarrow z \\
  f (0:[]) &\rightarrow \text{Left b} \\
  f (0:[]) &\rightarrow \text{Left 2} \\
  f y &\rightarrow z \\
  f (x:[]) &\rightarrow z
\end{align*}
\]
The equations of a function can always be reordered such that first-fit and best-fit pattern matching semantics coincide:

\[
\begin{align*}
  f :: [\text{Int}] \to \text{Either Int Char} \to \ldots \\
  f (x : xs) & \xrightarrow{\text{Left 1}} \\
  f y & \xrightarrow{\text{Right a}} \\
  f (0 : xs) & \xrightarrow{\text{Right 'X'}} \\
  f (1 : []) & \xrightarrow{z} \\
  f (0 : []) & \xrightarrow{z} \\
  f [] & \xrightarrow{z} \\
  f (0 : []) & \xrightarrow{\text{Left b}} \\
  f (0 : []) & \xrightarrow{\text{Left 2}} \\
  f y & \xrightarrow{z} \\
  f (x : []) & \xrightarrow{z} \\
  f (x : xs) & \xrightarrow{\text{Left 1}} \\
  f y & \xrightarrow{\text{Right a}} \\
  f y & \xrightarrow{z}
\end{align*}
\]
Open data types and open functions are tagged with the `open` keyword, but otherwise the syntax is almost the same as for normal declarations.

Type signatures for open functions are required, and only top-level functions can be open.

The meaning of programs containing open data types and open functions is simple, as if all the open entities had been declared closed, in one place.

There is no way to address different “versions” of a data type or a function.

Open entities can be exported or hidden via the module system, but only as a whole.
Overview

1. The language extension
   - Data types
   - Functions
   - Pattern matching

2. Applications
   - Expression problem
   - Generic programming
   - Exceptions

3. Implementation
   - Directly
   - Separate compilation

4. Conclusions
As we have seen as a running example, we can apply open data types to solve the expression problem.

- We can add new sorts of data by declaring a new constructor.
- We can add new operations by defining a new function.
- We can extend existing functions to new sorts of data by providing additional equations.
- There is no need to change existing code.
- The solution is as type-safe as Haskell is. Usually, pattern match failures occur at runtime, but patterns can be checked for exhaustiveness at compile time (resulting in a warning).
- We will consider separate compilation when we discuss the implementation.
Defining operations that are parameterized by a type argument and can access the structure of data types:

- structural equality
- pretty printing and parsing
- traversals and queries

Lightweight generic programming:

- within the language (as opposed to having a special-purpose extension)
- type reflection via type classes or representation types
- generic functions are type class members or functions that match on a type representation
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 :: * \(\rightarrow\) *
--- | ---
Int | Type Int
Char | Type Char
Unit | Type ()
Pair | Type a \(\rightarrow\) Type b \(\rightarrow\) Type (a, b)
Either | Type a \(\rightarrow\) Type b \(\rightarrow\) Type (Either a b)
List | Type a \(\rightarrow\) Type [a]

- The type () is Haskell’s “unit”-type with only one element, the type Either represents binary choice in Haskell, and [] is the built-in data type of homogeneous lists.
A type of type representations

<table>
<thead>
<tr>
<th>open data</th>
<th>Type :: * → *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Type Int</td>
</tr>
<tr>
<td>Char</td>
<td>Type Char</td>
</tr>
<tr>
<td>Unit</td>
<td>Type ()</td>
</tr>
<tr>
<td>Pair</td>
<td>Type a → Type b → Type (a, b)</td>
</tr>
<tr>
<td>Either</td>
<td>Type a → Type b → Type (Either a b)</td>
</tr>
<tr>
<td>List</td>
<td>Type a → Type [a]</td>
</tr>
</tbody>
</table>

- The type () is Haskell’s “unit”-type with only one element, the type Either represents binary choice in Haskell, and [] is the built-in data type of homogeneous lists.
- **Note:** The data type Type is a **generalized** algebraic data type.
- A value of type Type a is a representation of type a.
An overloaded equality function

**open** eq :: Type a → a → a → Bool

eq Int x y = x == y -- use built-in equality

eq Char x y = x == y -- use built-in equality

eq Unit () () = True

eq (Pair a b) (x1, x2) (y1, y2) = eq a x1 y1 ∧ eq b x2 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

\[ \textbf{open } \text{eq} :: \text{Type } a \rightarrow a \rightarrow a \rightarrow \text{Bool} \]

\[
\begin{align*}
\text{eq } \text{Int} & \quad x \quad y \quad = \ x \ == \ y \quad -- \ use \ built-in \ equality \\
\text{eq } \text{Char} & \quad x \quad y \quad = \ x \ == \ y \quad -- \ use \ built-in \ equality \\
\text{eq } \text{Unit} & \quad () \quad () \quad = \ True \\
\text{eq } (\text{Pair } a \ b) & \quad (x_1, x_2) \quad (y_1, y_2) \quad = \ \text{eq } a \ x_1 \ y_1 \ \land \ \text{eq } b \ x_2 \ y_2 \\
\text{eq } (\text{Either } a \ b) & \quad (\text{Left } x) \quad (\text{Left } y) \quad = \ \text{eq } a \ x \ y \\
\text{eq } (\text{Either } a \ b) & \quad (\text{Right } x) \quad (\text{Right } y) \quad = \ \text{eq } b \ x \ y \\
\text{eq } (\text{Either } a \ b) & \quad _ \quad _ \quad = \ False \\
\text{eq } (\text{List } a) & \quad xs \quad ys \quad = \ \text{and } (\text{zipWith } (\text{eq } a) \ xs \ ys)
\end{align*}
\]

Let us turn this function into a generic function:

\[
\begin{align*}
\text{eq } a \ x \ y & \quad = \ \textbf{case } \text{view } a \ \textbf{of} \ \text{View } a' \ \text{from to} \rightarrow \ \text{eq } a' \ (\text{from } x) \ (\text{from } y)
\end{align*}
\]

\[
\begin{align*}
\textbf{data } \text{View} :: \ast \rightarrow \ast \ \textbf{where} \\
\quad \text{View} :: \text{Type } a' \rightarrow (a \rightarrow a') \rightarrow (a' \rightarrow a) \rightarrow \text{View } a
\end{align*}
\]
generic equality – continued

open eq :: Type a → a → a → Bool

eq Int x y = x == y -- use built-in equality

eq Char x y = x == y -- use built-in equality

eq Unit () () = True

eq (Pair a b) (x₁, x₂) (y₁, y₂) = eq a x₁ y₁ ∧ eq b x₂ 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 – continued

open eq :: Type a → a → a → Bool

eq Int    x    y = x == y -- use built-in equality

eq Char   x    y = x == y -- use built-in equality

eq Unit   ()   () = True

eq (Pair a b) (x₁, x₂) (y₁, y₂) = eq a x₁ y₁ ∧ eq b x₂ 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)

- Using view, other data types can be mapped to (nearly) isomorphic types built from Unit, Pair and Either.

- The case for List is then subsumed by the generic case.

- For each new data type, the representation type Type must be extended, but usually not the generic functions.
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

throw :: Exception → a

catch :: IO a → (Exception → IO a) → IO a

- 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

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

- 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

```haskell
open data Exception :: *
```

Declaring a new exception:

```haskell
KeyNotFound :: Key → Exception
```

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. The language extension
   - Data types
   - Functions
   - Pattern matching

2. Applications
   - Expression problem
   - Generic programming
   - Exceptions

3. Implementation
   - Directly
   - Separate compilation

4. Conclusions
In the following, we sketch two possibilities to implement open data types and open functions by giving mappings to (plain) Haskell.

The first approach is a direct implementation of the semantics.

The second approach relies crucially on mutually recursive modules, but supports separate compilation.
A direct, naïve implementation

```haskell
module M1 where
open data X :: *
```

...
module $M_1$ where
open data $X :: *$

module $M_2$ where
import $M_1$
C :: · · · → $X$
open f :: $X → · · ·$
f (C · · ·) = · · · f · · ·

module $M_3$ where
import $M_1$
import $M_2$
D :: · · · → $X$
f (D · · ·) = · · · f · · ·
g = · · ·
module $M_1$ where
open data $X :: *$

module $M_2$ where
import $M_1$
$C :: \cdots \rightarrow X$
open $f :: X \rightarrow \ldots$
$f (C \ldots) = \ldots f \ldots$

module $M_3$ where
import $M_1$
import $M_2$
$D :: \cdots \rightarrow X$
$f (D \ldots) = \ldots f \ldots$
$g = \ldots$
A direct, naïve implementation

module $M_1$ where
open data $X :: *$

module $M_2$ where
import $M_1$
$C :: \ldots \to X$
open $f :: X \to \ldots$
$f (C \ldots) = \ldots f \ldots$

module $M_3$ where
import $M_1$
import $M_2$
$D :: \ldots \to X$
$f (D \ldots) = \ldots f \ldots$
g = \ldots

module $M_4$ where
$f = \ldots$
A direct, naïve implementation

module $M_1$ where
open data $X :: *$

module $M_2$ where
import $M_1$
$C :: \ldots \rightarrow X$
open $f :: X \rightarrow \ldots$
$f (C \ldots) = \ldots f \ldots$

module $M_3$ where
import $M_1$
import $M_2$
$D :: \ldots \rightarrow X$
$f (D \ldots) = \ldots f \ldots$
g = \ldots

module $M_4$ where
f = \ldots

module Main where
import $M_1$
import $M_1$
import $M_2$
import $M_3$
import $M_4$
main = \ldots M_2.f \ldots M_4.f \ldots
A direct, naïve implementation

module \( M_1 \) where
open data \( X \) :: *

module \( M_2 \) where
import \( M_1 \)
\( C :: \cdots \rightarrow X \)
open \( f :: X \rightarrow \cdots \)
\( f (C \cdots) = \cdots f \cdots \)

module \( M_3 \) where
import \( M_1 \)
import \( M_2 \)
\( D :: \cdots \rightarrow X \)
\( f (D \cdots) = \cdots f \cdots \)
g = \cdots

module \( M_4 \) where
\( f = \cdots \)

module Main where

\( \text{main} = \cdots f \cdots f' \cdots \)
A direct, naïve implementation – continued

- Like semantics: collapse program into a single module.
- All open data types become closed data types.
- All open functions become closed functions, the equations are reordered to respect best-fit pattern matching.
- Advantage: easy to implement, certainly correct.
- Big disadvantage: no separate compilation; inefficient compilation for large programs.
- No performance problems for the resulting programs, however.
open eval :: Expr → Int

eval (Num n) = n

eval (Sum e₁ e₂) = eval e₁ + eval e₂

eval (Prod e₁ e₂) = eval e₁ * eval e₂
open eval :: Expr → Int

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

\[
\begin{align*}
\text{eval (Num } n) & = \text{eval}_1 n \\
\text{eval (Sum } e_1 e_2) & = \text{eval}_2 e_1 e_2 \\
\text{eval (Prod } e_1 e_2) & = \text{eval}_3 e_1 e_2 \\
\text{eval}_1 n & = n \\
\text{eval}_2 e_1 e_2 & = \text{eval } e_1 + \text{eval } e_2 \\
\text{eval}_3 e_1 e_2 & = \text{eval } e_1 \times \text{eval } e_2
\end{align*}
\]
module $M_1$ where
open data X :: *

module $M_2$ where
import $M_1$
C :: · · · → X
open f :: X → · · ·
f (C . . .) = · · ·f . . .

module $M_3$ where
import $M_1$
import $M_2$
D :: · · · → X
f (D . . .) = · · ·f . . .
g = · · ·

module $M_4$ where
f = · · ·

module Main where
import $M_1$
import $M_2$
import $M_3$
import $M_4$
main = · · ·M_2.f . . .M_4.f . . .
Implementation using several modules

module M₁ where
open data X :: *

module M₂ where
import M₁
C :: · · · → X
open f :: X → · · ·
f (C · · · ) = · · · f · · ·

module M₃ where
import M₁
import M₂
D :: · · · → X
f (D · · · ) = · · · f · · ·
g = · · ·

module M₄ where
f = · · ·

module Closure where
import M₁
import M₂
import M₃
data X where
C :: · · · → X
D :: · · · → X
f (C · · · ) = f₁ · · ·
f (D · · · ) = f₂ · · ·

module Main where
import M₁
import M₂
import M₃
import M₄
main = · · · M₂.f · · · M₄.f · · ·
Implementation using several modules

module M₁ where
open data X :: *

module M₂ where
import M₁
C :: ⋯ → X
open f :: X → ⋯
f (C ⋯) = ⋯f ⋯

module M₃ where
import M₁
import M₂
D :: ⋯ → X
f (D ⋯) = ⋯f ⋯
g = ⋯

module M₄ where
f = ⋯

module Closure where
import M₁
import M₂
import M₃
data X where
  C :: ⋯ → X
  D :: ⋯ → X
f (C ⋯) = f₁ ⋯
f (D ⋯) = f₂ ⋯

module M₁ (module M₁, module Closure) where
import Closure (data X :: *)

module Main where
import M₁
import M₂
import M₃
import M₄
main = ⋯M₂.f ⋯M₄.f ⋯
Implementation using several modules

module M₁ where
  open data X :: *

module M₂ where
  import M₁
  C :: · · · → X
  open f :: X → · · ·
  f (C . . .) = . . . f . . .

module M₃ where
  import M₁
  import M₂
  D :: · · · → X
  f (D . . .) = . . . f . . .
  g = . . .

module M₄ where
  f = . . .

module Main where
  import M₁
  import M₂
  import M₃
  import M₄
Implementation using several modules

module M1 where
open data X :: *

module M2 where
import M1
C :: · · · → X
open f :: X → · · ·
f (C · · ·) = · · · f · · ·

module M3 where
import M1
import M2
D :: · · · → X
f (D · · ·) = · · · f · · ·
g = · · ·

module M4 where
f = · · ·

module Main where
import M1
import M2
import M3
import M4
main = · · · M2.f · · · M4.f · · ·
Implementation using several modules

module M_1 where
open data X :: *

module M_2 where
import M_1
C :: · · · → X
open f :: X → · · ·
f (C · · ·) = · · · f · · ·

data X where
  C :: · · · → X
  D :: · · · → X
f (C · · ·) = f_1 · · ·
f (D · · ·) = f_2 · · ·

module M_3 where
import M_1
import M_2
D :: · · · → X
f (D · · ·) = · · · f · · ·
g = · · ·

module M_4 where
f = · · ·

module Closure where
import M_1
import M_2
import M_3

module M_1 (module M_1, module Closure) where
import Closure (data X :: *)

module M_2 (module M_2, module Closure) where
import Closure (C :: · · · → X, f :: X → · · ·)
import M_1
f_1 · · · = · · · f · · ·

module M_3 (module M_3, module Closure) where
import Closure (D :: · · · → X)
import M_1
import M_2
f_2 · · · = · · · f · · ·
g = · · ·

module M_4 where
f = · · ·

module Main where
import M_1
import M_2
import M_3
import M_4
main = · · · M_2.f · · · M_4.f · · ·
Implementation using several modules

module M₁ where
open data X :: *

module M₂ where
import M₁
C :: · · · → X
open f :: X → · · ·
f (C · · ·) = · · · f · · ·

data X where
  C :: · · · → X
  D :: · · · → X
  f (C · · ·) = f₁ · · ·
f (D · · ·) = f₂ · · ·

module M₃ where
import M₁
import M₂
D :: · · · → X
f (D · · ·) = · · · f · · ·
g = · · ·

module M₄ where
f = · · ·

module Main where
import M₁
import M₂
import M₃
import M₄
main = · · · M₂.f · · · M₄.f · · ·
Implementation using several modules – continued

- 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. The language extension
   - Data types
   - Functions
   - Pattern matching

2. Applications
   - Expression problem
   - Generic programming
   - Exceptions

3. Implementation
   - Directly
   - Separate compilation

4. Conclusions
Type classes allow to encode extensible data types, but one cannot use pattern matching to define functions, and the decision has to be made in the beginning (changing the code later is non-trivial).

Many properties and restrictions of type classes used:
- One global meaning for open entities.
- Limited possibilities to hide the visibility.
- Only top-level open entities.
- Overlapping instances resolution corresponds to best-fit pattern matching.

More properties of type classes could be transferred:
- Partial evaluation of pattern matching.
- Automatic inference of uniquely determined values.
More about Haskell peculiarities:

- Interaction with type classes
- Full pattern language
- Interaction with `deriving`
- ...

More about related work:

- Lots of related work, but most approaches try to solve a more complex problem.
Conclusions

- Very simple solution: no changes to the type system, no deep semantics.
- Open data types and open functions are syntactically similar to their closed counterparts.
- One easy implementation, one relatively efficient implementation.
- Our approach applies to many interesting examples.