Open data types and open functions

Andres Löh and Ralf Hinze

July 11, 2006

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

Data types in 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.

Data types in Haskell – continued

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.

Syntax: open data types

```
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).

Syntax: open data types

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 $\rightarrow$ Expr

Syntax: open data types

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 $\rightarrow$ Expr

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

Syntax: open data types

```
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
```

Syntax: open data types

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

- 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).

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

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

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

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

Syntax: open data types

```
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 referr 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 what if we extend a function that has overlapping patterns?

```
open simplify :: Expr → Expr
. . .
simplify (Sum (Num 0) e₂) = simplify e₂
. . .
simplify e                = e
```

Pattern matching

The function eval is very nice so far, because it has non-overlapping patterns. What 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:

```
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
```

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:

```
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
```

Summary

- 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

Expression problem

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.

Lightweight generic programming

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 :: * → *
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]
```

- 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

```
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]
```

- 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

```
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)
```

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 :: Type a′ → (a → a′) → (a′ → a) → View a
```

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

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

```
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.
- 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
```

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

Translating to Haskell

- 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

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

A direct, naïve implementation

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

```
module M₂ where
import M₁
C :: ··· → X

open f :: X → ...
f (C ...) = ... f ...
```

A direct, naïve implementation

```
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 = ...
```

A direct, naïve implementation

```
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 = . . .
```

A direct, naïve implementation

```
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 . . .
```

A direct, naïve implementation

```
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 ...
```

```
module Main where

data X where
   C :: ··· → X
   D :: ··· → X

f :: X → ...
f (C ...) = ... f ...
f (D ...) = ... f ...

g = ...

f' = ...

main = ... f ... f' ...
```

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.

Splitting code and case selection

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

Splitting code and case selection

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

```
eval (Num n)       = eval₁ n
eval (Sum e₁ e₂)  = eval₂ e₁ e₂
eval (Prod e₁ e₂) = eval₃ e₁ e₂
eval₁ n            = n
eval₂ e₁ e₂        = eval e₁ + eval e₂
eval₃ e₁ e₂        = eval e₁ ∗ eval e₂
```

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 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 . . .
```

```
module Closure where
import M1
import M2
import M3

data X where
  C :: · · · → X
  D :: · · · → X

f (C . . . ) = f1 . . .
f (D . . . ) = f2 . . .
```

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 . . .
```

```
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 :: ∗)
```

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 ...
```

```
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 M₂ (module M₂, module Closure) where
import Closure (C :: ··· → X, f :: X → ...)
import M₁

f₁ ... = ... 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

main = ... M2.f ... M4.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 M2 (module M2, module Closure) where
import Closure (C :: ··· → X, f :: X → ...)
import M1

f1 ... = ... f ...
```

```
module M3 (module M3, module Closure) where
import Closure (D :: ··· → X)
import M1
import M2

f2 ... = ... f ...

g = ...
```

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 . . .
```

```
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 . . .
```

```
module Closure where
import M_1
import M_2
import M_3

data X where
  C :: · · · → X
  D :: · · · → X

f (C . . . ) = f_1 . . .
f (D . . . ) = f_2 . . .
```

```
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 = . . .
```

Implementation using several modules

```
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 ...) = ... f ...
```

```
module $M_3$ where
import $M_1$
import $M_2$

D :: $\cdots \rightarrow$ 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 ...
```

```
module Closure where
import $M_1$
import $M_2$
import $M_3$

data X where
   C :: $\cdots \rightarrow$ X
   D :: $\cdots \rightarrow$ X

f (C ...) = $f_1$ ...
f (D ...) = $f_2$ ...
```

```
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 :: $\cdots \rightarrow$ X, f :: X $\rightarrow \ldots$)
import $M_1$

$f_1$ ... = ... f ...
```

```
module $M_3$ (module $M_3$, module Closure) where
import Closure (D :: $\cdots \rightarrow$ 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 – 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

Relation to type classes

- 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.

In the paper

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.