Certified Software Development with Dependent Types in Idris

Lecture 6. Interfaces, Modules, Namespaces

Vitaly Bragilevsky (bravit@sfedu.ru)

I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science
Southern Federal University, Rostov-on-Don, Russia
part of Erasmus+ teaching mobility agreement with University of Twente

Content

Content

Types, Interfaces, and Implementations

Type

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Types, Interfaces, and Implementations

Type

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Interface

```
interface Show a where
  show : a -> String
```

Types, Interfaces, and Implementations

Type

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Interface

```
interface Show a where
  show : a -> String
```

Implementation

```
Show NPair where
  show (MkNPair n m) = "(" ++ show n ++ "," ++ show m ++ ")"
```

Types, Interfaces, and Implementations

Type

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Interface

```
interface Show a where
  show : a -> String
```

Implementation

```
Show NPair where
  show (MkNPair n m) = "(" ++ show n ++ "," ++ show m ++ ")"
```

Usage

```
Idris> :type show
show : Show a => a -> String
Idris> show $ MkNPair 5 10
"(5,10)" : String
```

Types, Interfaces, and Implementations

Type

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Interface

```
interface Show a where
  show : a -> String
```

Implementation

```
Show NPair where
  show (MkNPair n m) = "(" ++ show n ++ "," ++ show m ++ ")"
```

Usage

```
Idris> :type show
show : Show a => a -> String
Idris> show $ MkNPair 5 10
"(5,10)" : String
```

- Terminology: type implements interface.

Interface Eq

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

- Checking equality
- Default definitions
- Many types implement Eq

Interface Ord

```
interface Eq a => Ord a where
  compare : a -> a -> Ordering
  (<) : a -> a -> Bool
  (>) : a -> a -> Bool
  (<=) : a -> a -> Bool
  (>=) : a -> a -> Bool
  max : a -> a -> a
  min : a -> a -> a
```

- Extending `Eq`
- Minimal complete definition: `compare`
- `data Ordering = LT | EQ | GT`

Basic Usage of Interfaces

```
sort : Ord a => List a -> List a

sortAndShow : (Ord a, Show a) => List a -> String
sortAndShow xs = show (sort xs)
```

Named Implementations

Using default Show implementation for Nat

```
Idris> show (S (S (S Z)))
"3" : String
```

Named Implementations

Using default Show implementation for Nat

```
Idris> show (S (S (S Z)))
"3" : String
```

Named Implementation

```
[myShowNat] Show Nat where
  show Z = "z"
  show (S k) = strCons 's' (show k)
```

```
Idris> show @{myShowNat} (S (S (S Z)))
"sssz" : String
```

Named Implementations

Using default Show implementation for Nat

```
Idris> show (S (S (S Z)))
"3" : String
```

Named Implementation

```
[myShowNat] Show Nat where
  show Z = "z"
  show (S k) = strCons 's' (show k)
```

```
Idris> show @{myShowNat} (S (S (S Z)))
"sssz" : String
```

```
f : Show a => a -> String
f a = "Result: " ++ show a
```

```
Idris> f @{myShowNat} (S Z)
"Result: sz" : String
```

Some Other Interfaces from Prelude

Numeric Interfaces

```
interface Num a where
 (+) : a -> a -> a
 (*) : a -> a -> a
 fromInteger : Integer -> a

interface Num a => Neg a
                      where
 negate : a -> a
 (-) : a -> a -> a
 abs : a -> a

interface Integral a where
 div : a -> a -> a
 mod : a -> a -> a
```

Bounded Interfaces

```
interface Ord b => MinBound b
                         where
 minBound : b

interface Ord b => MaxBound b
                         where
 maxBound : b
```

Enumeration

```
interface Enum a where
 pred : a -> a
 succ : a -> a
 succ e = fromNat (S (toNat e))
 toNat : a -> Nat
 fromNat : Nat -> a
```

Content

Semigroup (Prelude.Algebra)

```
interface Semigroup a where
  (<+>) : a -> a -> a
```

Semigroup (Prelude.Algebra)

```
interface Semigroup a where
  (<+>) : a -> a -> a
```

```
Idris> "123" <+> "456"
"123456" : String
Idris> [1,2,3] <+> [4,5]
[1, 2, 3, 4, 5] : List Integer
Idris> Just 5 <+> Just 10
Just 5 : Maybe Integer
Idris> Nothing <+> Just 10
Just 10 : Maybe Integer
```

Semigroups for Maybe a

```
Semigroup (Maybe a) where
  Nothing   <+> m = m
  (Just x)  <+> _ = Just x

[collectJust] Semigroup a => Semigroup (Maybe a) where
  Nothing   <+> m         = m
  m         <+> Nothing = m
  (Just m1) <+> (Just m2) = Just (m1 <+> m2)
```

```
Idris> (<+>) (Just "123") (Just "456")
Just "123" : Maybe String
Idris> (<+>) @{collectJust} (Just "123") (Just "456")
Just "123456" : Maybe String
```

Monoid (Prelude.Algebra)

```
interface Semigroup a => Monoid a where
  neutral : a
```

```
Idris> the String neutral
"" : String
Idris> the (Maybe Nat) neutral
Nothing : Maybe Nat
```

Numeric Monoids

Wrappers for Nats

```
record Additive where
  constructor GetAdditive
  _ : Nat

record Multiplicative where
  constructor GetMultiplicative
  _ : Nat
```

```
Idris> the Additive neutral
GetAdditive 0 : Additive
Idris> the Multiplicative neutral
GetMultiplicative 1 : Multiplicative
Idris> GetAdditive 5 <+> GetAdditive 10
GetAdditive 15 : Additive
Idris> GetMultiplicative 5 <+> GetMultiplicative 10
GetMultiplicative 50 : Multiplicative
```

Numeric Monoids: Implementation

```
Semigroup Additive where
  left <+> right = GetAdditive $ left' + right'
    where
      left'  : Nat
      left'  = case left of
                 GetAdditive m => m

      right' : Nat
      right' = case right of
                 GetAdditive m => m

Monoid Additive where
  neutral = GetAdditive Z
```

What can be a parameter to an interface?

```
interface InterfaceName a where
  ...
```

What can be a parameter to an interface?

```
interface InterfaceName a where
  ...
```

- a Type
- a Type-valued function (arbitrary type constructor) — we need an explicit type declaration in this case

Content

Functor

Definition of a Functor

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

Functor

Definition of a Functor

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

What can be a value in the context?

- A value of some type with the possibility of failure (`Maybe a`).
- A value of some type or explanation of failure (`Either a b`).
- A result of nondeterministic computation (`List a`).

Functor

Definition of a Functor

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

What can be a value in the context?

- A value of some type with the possibility of failure (`Maybe a`).
- A value of some type or explanation of failure (`Either a b`).
- A result of nondeterministic computation (`List a`).

Idea of a Functor

Functor enables modification of a value without altering its context so that it abstracts context out.

Using Functor Implementations

```
Idris> map (+1) (Just 1)
Just 2 : Maybe Integer
Idris> map (+1) Nothing
Nothing : Maybe Integer
Idris> the (Either String Integer) (map (+1) (Right 5))
Right 6 : Either String Integer
Idris> map (+1) (Left "some mistake")
Left "some mistake" : Either String Integer
Idris> map (+1) [1,2,3,4,5]
[2, 3, 4, 5, 6] : List Integer
```

Using Functor Implementations

```
Idris> map (+1) (Just 1)
Just 2 : Maybe Integer
Idris> map (+1) Nothing
Nothing : Maybe Integer
Idris> the (Either String Integer) (map (+1) (Right 5))
Right 6 : Either String Integer
Idris> map (+1) (Left "some mistake")
Left "some mistake" : Either String Integer
Idris> map (+1) [1,2,3,4,5]
[2, 3, 4, 5, 6] : List Integer
```

Synonym for map

```
Idris> :t (<$>)
(<$>) : Functor f => (a -> b) -> f a -> f b
Idris> negate <$> (Just 1)
Just -1 : Maybe Integer
```

Applicative: Abstraction over Function Application

Definition of a Functor

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

Definition of an Applicative

```
interface Functor f => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b
```

Applicative: Abstraction over Function Application

Definition of a Functor

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

Definition of an Applicative

```
interface Functor f => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b
```

```
Idris> pure max <*> (Just 2) <*> (Just 3)
Just 3 : Maybe Integer
Idris> max <$> (Just 2) <*> (Just 3)
Just 3 : Maybe Integer
```

Adding Maybes: Version 1

```
m_add : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add (Just n) (Just m) = Just (n + m)
m_add _ _ = Nothing
```

```
Idris> m_add (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add (Just 5) Nothing
Nothing : Maybe Nat
```

- We have to deal with Nothing by ourselves

Adding Maybes: Version 2 (Using Applicative for Maybe)

```
m_add' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add' a b = pure plus <*> a <*> b
```

```
Idris> m_add' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add' (Just 5) Nothing
Nothing : Maybe Nat
```

Adding Maybes: Version 2 (Using Applicative for Maybe)

```
m_add' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add' a b = pure plus <*> a <*> b
```

```
Idris> m_add' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add' (Just 5) Nothing
Nothing : Maybe Nat
```

```
Applicative Maybe where
    pure = Just

    (Just f) <*> (Just a) = Just (f a)
    _        <*> _        = Nothing
```

Adding Maybes: Version 3 (Idiom Brackets)

```
m_add'' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add'' a b = [| a + b |]
```

```
Idris> m_add'' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add'' (Just 5) Nothing
Nothing : Maybe Nat
```

- Idiom brackets — syntactic sugar for applicatives
- `[| f a1 ...an |]`
  is translated into
  `pure f <*> a1 <*> ... <*> an`

Adding Maybes: Version 4 (!-notation)

```
m_add''' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add''' a b = pure (!a + !b)
```

```
Idris> m_add''' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add''' (Just 5) Nothing
Nothing : Maybe Nat
```

Let's add even more abstraction!

```
a_add : (Semigroup a, Applicative f) => f a -> f a -> f a
a_add a b = [| a <+> b |]
```

Let's add even more abstraction!

```
a_add : (Semigroup a, Applicative f) => f a -> f a -> f a
a_add a b = [| a <+> b |]
```

```
Idris> a_add (Just "123") (Just "456")
Just "123456" : Maybe String
Idris> a_add (Just (getAdditive 5)) (Just (getAdditive 10))
Just (getAdditive 15) : Maybe Additive
```

What if we take List for the context?

```
Idris> :let xs = map getMultiplicative [1,2,3]
defined
Idris> :let ys = map getMultiplicative [5,6]
defined
```

What if we take List for the context?

```
Idris> :let xs = map getMultiplicative [1,2,3]
defined
Idris> :let ys = map getMultiplicative [5,6]
defined
Idris> a_add xs ys
[getMultiplicative 5, getMultiplicative 6,
 getMultiplicative 10, getMultiplicative 12,
 getMultiplicative 15, getMultiplicative 18]
       : List Multiplicative
```

What is going on here?

```
Applicative List where
    pure x = [x]

    fs <*> vs = concatMap (\f => map f vs) fs
```

What is going on here?

```
Applicative List where
    pure x = [x]

    fs <*> vs = concatMap (\f => map f vs) fs
```

```
Idris> :doc concatMap
Prelude.Foldable.concatMap : Foldable t => Monoid m =>
    (a -> m) -> t a -> m

    Combine into a monoid the collective results of applying
    a function to each element of a structure
```

- List is a Functor
- Monoid operation for lists is concatenation
- List is Foldable

Interface Foldable

```
interface Foldable (t : Type -> Type) where
  foldr : (elt -> acc -> acc) -> acc -> t elt -> acc
  foldl : (acc -> elt -> acc) -> acc -> t elt -> acc
```

```
foldr op acc [x1,x2,...,xn]
    === x1 `op` (x2 `op` ... (xn `op` acc)...)

foldl op acc [x1,x2,...,xn]
    === (...((acc `op` x1) `op` x2)...) `op` xn
```

From Functor to Monad

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b

interface Functor f => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b

interface Applicative m => Monad (m : Type -> Type) where
    (>>=)  : m a -> (a -> m b) -> m b
```

From Functor to Monad

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b

interface Functor f => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b

interface Applicative m => Monad (m : Type -> Type) where
    (>>=)  : m a -> (a -> m b) -> m b
```

```
Idris> Just 10 >>= (\x => Just (x+1))
Just 11 : Maybe Integer
Idris> Nothing >>= (\x => Just (x+1))
Nothing : Maybe Integer
```

Example: Signal Processing (1)

Processing Stages

```
data SignalPreference s = Default s
                        | Received
                        | Corrected (s->s)

preprocess : Monad m => m s -> SignalPreference s -> m s
preprocess ms sp = case sp of
                     Default s => pure s
                     Received => ms
                     Corrected f => map f ms

process : Monad m => s -> m s
process = pure . id

postprocess : Monad m => (s -> s) -> s -> m s
postprocess f = pure . f
```

Example: Signal Processing (2)

```
signal : Monad m => m s                         -- original signal
                 -> m (SignalPreference s)      -- preprocessing
                 -> (s -> s)                    -- postprocessing
                 -> m s                         -- result
signal ms sp f =
       sp >>= preprocess ms >>= process >>= postprocess f
```

Example: Signal Processing (2)

```
signal : Monad m => m s                         -- original signal
                 -> m (SignalPreference s) -- preprocessing
                 -> (s -> s)                 -- postprocessing
                 -> m s                      -- result
signal ms sp f =
       sp >>= preprocess ms >>= process >>= postprocess f
```

```
Idris> signal Nothing (Just $ Default 0) (+1)
Just 1 : Maybe Integer
Idris> signal (Just 1) (Just $ Corrected (+1)) (+1)
Just 3 : Maybe Integer
Idris> signal Nothing (Just $ Corrected (+1)) (+1)
Nothing : Maybe Integer
Idris> signal (Just 1) Nothing (+1)
Nothing : Maybe Integer
```

Example: Signal Processing (3)

List as a monad

```
Idris> signal [1,2] [Corrected (+1),Received,Default 0] (+1)
[3, 4, 2, 3, 1] : List Integer
Idris> signal [] [Default 0, Default 1]  (+1)
[1, 2] : List Integer
```

Example: Signal Processing (3)

List as a monad

```
Idris> signal [1,2] [Corrected (+1),Received,Default 0] (+1)
[3, 4, 2, 3, 1] : List Integer
Idris> signal [] [Default 0, Default 1]  (+1)
[1, 2] : List Integer
```

Either as a monad

```
Idris> signal (Right 10) (Left $ "Unknown error") (+1)
Left "Unknown error" : Either String Integer
Idris> signal (Left "No sig") (Right $ Default 0) (+1)
Right 1 : Either String Integer
Idris> signal (Left "No sig") (Right $ Corrected (+1)) (+1)
Left "No signal" : Either String Integer
```

Example: Signal Processing (4)

Function signal using do-notation

```
signal : Monad m => m s
                 -> m (SignalPreference s)
                 -> (s -> s)
                 -> m s
signal ms sp f = do
  sp' <- sp
  s <- preprocess ms sp'
  s' <- process s
  postprocess f s'
```

Desugaring do-notation

```
do
  x <- v
  e                becomes          v >>= (\x => e)

do
  v
  e                becomes          v >>= (\_ => e)

do
  let x = v
  e                becomes          let x = v in e
```

Monad Implementations

```
Monad Maybe where
    Nothing  >>= k = Nothing
    (Just x) >>= k = k x

Monad (Either e) where
    (Left n) >>= _ = Left n
    (Right r) >>= f = f r

Monad List where
    m >>= f = concatMap f m
```

Interface Alternative

```
interface Applicative f => Alternative (f : Type -> Type)
                                                          where
    empty : f a
    (<|>) : f a -> f a -> f a

guard : Alternative f => Bool -> f ()
guard a = if a then pure () else empty
```

Example: Signal Processing (5)

```
sig2 : (Alternative m, Monad m, Ord s)
  => m s -> m s -> m (SignalPreference s) -> (s -> s) -> m s
sig2 ms1 ms2 sp f = do
  s1 <- ms1
  s2 <- ms2
  s <- signal ms1 sp f <|> signal ms2 sp f
  guard (s1 < s && s < s2)
  pure s
```

Example: Signal Processing (5)

```
sig2 : (Alternative m, Monad m, Ord s)
  => m s -> m s -> m (SignalPreference s) -> (s -> s) -> m s
sig2 ms1 ms2 sp f = do
  s1 <- ms1
  s2 <- ms2
  s <- signal ms1 sp f <|> signal ms2 sp f
  guard (s1 < s && s < s2)
  pure s
```

```
Idris> sig2 (Just 5) (Just 15) (Just $ Corrected (*2)) (+1)
Just 11 : Maybe Integer
Idris> sig2 (Just 5) (Just 15) (Just $ Default 20) (+1)
Nothing : Maybe Integer
Idris> sig2 Nothing (Just 15) (Just $ Default 20) (+1)
Nothing : Maybe Integer
```

Monad comprehensions

```
sig2 : (Alternative m, Monad m, Ord s)
  => m s -> m s -> m (SignalPreference s) -> (s -> s) -> m s
sig2 ms1 ms2 sp f = do
  s1 <- ms1
  s2 <- ms2
  s <- signal ms1 sp f <|> signal ms2 sp f
  guard (s1 < s && s < s2)
  pure s
```

```
sig2 ms1 ms2 sp f
     = [ s | s1 <- ms1,
             s2 <- ms2,
             s  <- signal ms1 sp f <|> signal ms2 sp f,
             s1 < s && s < s2 ]
```

Adding Maybes: Versions 1–6

Version 1 (pattern matching)

```
m_add : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add (Just n) (Just m) = Just (n + m)
m_add _ _ = Nothing
```

Version 2 (applicative style)

```
m_add a b = (+) <$> a <*> b
```

Version 3 (idiom brackets)

```
m_add a b = [| a + b |]
```

Version 4 (!-notation)

```
m_add a b = pure (!a + !b)
```

Version 5 (do-notation)

```
m_add a b = do
  a' <- a
  b' <- b
  pure (a'+b')
```

Version 6 (monad comprehensions)

```
m_add a b = [ x+y | x<-a, y<-b ]
```

Conclusion

- Interfaces enable abstraction over types and type constructors.

Conclusion

- Interfaces enable abstraction over types and type constructors.
- Implementations provide link between types and interfaces.

Conclusion

- Interfaces enable abstraction over types and type constructors.
- Implementations provide link between types and interfaces.
- You should remember such interfaces as Eq, Ord, Num, Semigroup, Monoid, Functor, Applicative, Foldable, Monad.

Content

Program as Collection of Modules

File “ModA.idr”

```
module ModA

-- interfaces and impl-s
-- types and functions
```

File “ModB.idr”

```
module ModB

-- interfaces and impl-s
-- types and functions
```

File “program.idr”

```
module Main

import ModA
import ModB

-- types, interfaces, implementations, functions

main : IO ()
main = ...
```

Modules in Subdirectories

File “Utils/Mod.idr”

```
module Utils.Mod

-- ???
```

File “program.idr”

```
module Main

import Utils.Mod

-- types, interfaces, instances, functions

main : IO ()
main = ...
```

Fully Qualified Names

```
Idris> :t (::)
ForeignEnv.(::) : (ffi_types f t, t) ->
                  FEnv f xs -> FEnv f (t :: xs)
Prelude.List.(::) : elem -> List elem -> List elem
Prelude.Stream.(::) : a ->
                      Lazy' LazyCodata (Stream a) -> Stream a
```

Fully Qualified Names

```
Idris> :t (::)
ForeignEnv.(::) : (ffi_types f t, t) ->
                  FEnv f xs -> FEnv f (t :: xs)
Prelude.List.(::) : elem -> List elem -> List elem
Prelude.Stream.(::) : a ->
                      Lazy' LazyCodata (Stream a) -> Stream a
```

```
Idris> :t Prelude.List.(::)
(::) : elem -> List elem -> List elem
```

Export Modifiers

- We can export names, constructors, implementations
- We can use modifiers private/export/public export for tuning which module components are exported
- There are default rules and directive `%access`
- See Export Modifiers section in Idris tutorial

Explicit Namespaces

```
module Foo

namespace x
  test : Double -> Double
  test x = x * 2

namespace y
  test : String -> String
  test x = x ++ x
```

```
Idris> Foo.x.test 10
20.0 : Double
Idris> test 10.5
21 : Double
Idris> test "aaa"
"aaaaaa" : String
```

Bibliography

- Idris Tutorial: Interfaces
  http://docs.idris-lang.org/en/latest/tutorial/interfaces.html
- Idris Tutorial: Modules and Namespaces
  http://docs.idris-lang.org/en/latest/tutorial/modules.html
- Idris Libraries Source Code
  https://github.com/idris-lang/Idris-dev/tree/master/libs/