Type Driven Development in Idris

Edwin Brady (ecb10@st-andrews.ac.uk)
University of St Andrews, Scotland, UK
@edwinbrady

Strange Loop, St. Louis, 24th September 2015

sicsa*

Scotland

Idris is a *Pac-man Complete* functional programming language with *dependent types*

- `cabal update; cabal install idris`
- `http://idris-lang.org/download`
- `http://idris-lang.org/STL2015/`

IDRIS is a *Pac-man Complete* functional programming language with *dependent types*

- `cabal update; cabal install idris`
- `http://idris-lang.org/download`
- `http://idris-lang.org/STL2015/`

This workshop is in two parts:

- Fundamentals: A tour of Idris
- Managing *side-effects* and *resources*

sicsa*

Ask me things!

If . . .

- There's anything you don't understand
- Anything you want me to explain better
- Any exercises you'd like help with

. . . *please don't hesitate to ask*!

- Either during the workshop. . .
- . . . or come and find me throughout the conference
- . . . or email me afterwards at `edwin.brady@gmail.com`

Why types?

We can use *type systems* for:

- *Checking* a program has the intended properties
- *Guiding* a programmer towards a correct program
- Building *expressive* and *generic* libraries

Why types?

We can use *type systems* for:

- *Checking* a program has the intended properties
- *Guiding* a programmer towards a correct program
- Building *expressive* and *generic* libraries

Type Driven Development puts *types* first. Three steps:

- *Type*: Write a type for a function
- *Define*: Create a (possibly incomplete) implementation
- *Refine*: Improve/complete the implementation

- *First class* dependent types
  - Functions can compute types, types can contain values

- *First class* dependent types
  - Functions can compute types, types can contain values
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation

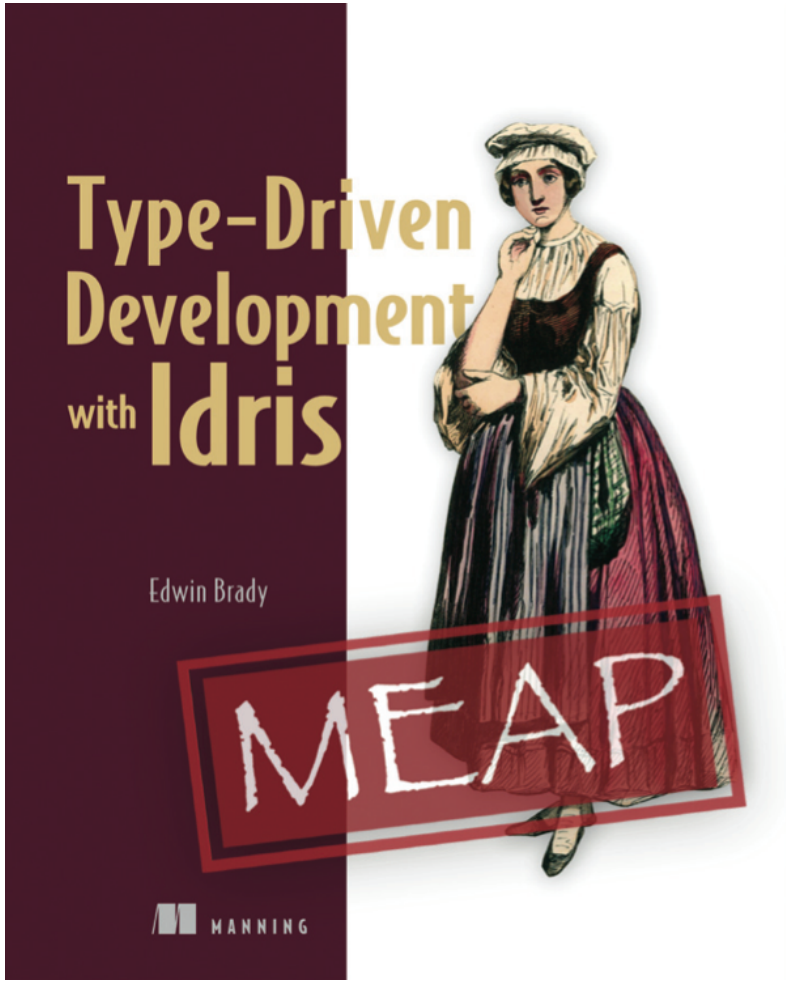
- *First class* dependent types
  - Functions can compute types, types can contain values
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation
- Type classes, like Haskell
  - No `deriving`, yet
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets

- *First class* dependent types
  - Functions can compute types, types can contain values
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation
- Type classes, like Haskell
  - No `deriving`, yet
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets
- *Strict* evaluation order, unlike Haskell
  - `Lazy` as a type

- *First class* dependent types
  - Functions can compute types, types can contain values
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation
- Type classes, like Haskell
  - No `deriving`, yet
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets
- *Strict* evaluation order, unlike Haskell
  - `Lazy` as a type
- Foreign functions, system interaction

Demonstration: Introductory Examples

https://www.manning.com/books/type-driven-development-with-idris



Manning Deal of the Day, September 24th
50% discount
Discount code: dotd092415tw

sicsa*

Demonstration: An Effectful Evaluator

Effectful programs

```
data EffM : (m : Type -> Type) -> (res : Type) ->
            (in_effects : List EFFECT) ->
            (out_effects : res -> List EFFECT) ->
            Type
```

Effects in Idris

Effectful programs

```
data EffM : (m : Type -> Type) -> (res : Type) ->
            (in_effects : List EFFECT) ->
            (out_effects : res -> List EFFECT) ->
            Type
```

Composing programs

```
(>>=) : EffM m res es es' ->
        ((x : res) -> EffM m b (es' x) es'') ->
        EffM m b es es''
```

Examples

```
get  :       EffM m t  [STATE t] (\x : t  => [STATE t])
put  : t -> EffM m ()  [STATE t] (\x : () => [STATE t])

putM : t' -> EffM m ()  [STATE t]
                                  (\x : () => [STATE t'])
```

Examples

```
get  :       Eff t [STATE t]
put  : t -> Eff () [STATE t]

putM : t' -> Eff () [STATE t] [STATE t']
```

Examples

Combining effects

```
inc : Eff () [STDIO, STATE Nat]
inc = do x <- get
         putStrLn ("Old value " ++ show x)
         put (x + 1)
```

Examples

Labelling effects

```
sumStates : Eff Nat ['xval ::: STATE Nat,
                     'yval ::: STATE Nat]
sumStates = do x <- 'xval :- get
               y <- 'yval :- get
               return (x + y)
```

Running Effectful Programs

```
run : Applicative m => {auto env : Env m xs} ->
      (prog : EffM m a xs xs') -> m a
runPure : {auto env : Env id xs} ->
          (prog : Eff id a xs xs') -> a
```

Effect Signatures

```
Effect : Type
Effect = (t : Type) -> (res : Type) ->
                       (res' : t -> Type) -> Type
```

```
data State : Effect where
     Get :       State a  a (const a)
     Put : b -> State () a (const b)
```

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

Effect Signatures

```
Effect : Type
Effect = (t : Type) -> (res : Type) ->
                       (res' : t -> Type) -> Type
```

```
data State : Effect where
     Get :       sig State a  a
     Put : b -> sig State () a b
```

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

Effect Signatures

```
data StdIO : Effect where
     PutStr : String -> sig StdIO ()
     GetStr :           sig StdIO String
     PutCh  : Char ->   sig StdIO ()
     GetCh  :           sig StdIO Char
```

```
STDIO : EFFECT
STDIO = MkEff () StdIO
```

Effect Handlers

Handlers

```
class Handler (e : Effect) (m : Type -> Type) where
  covering
  handle : (r : res) -> (eff : e t res resk) ->
           (k : ((x : t) -> resk x -> m a)) -> m a
```

Example Instances

```
instance Handler State m
instance Handler StdIO IO
instance Handler StdIO (List String ->
                                (a, List String))
```

Effect Handlers

State

```
instance Handler State m where
     handle st Get      k = k st st
     handle st (Put n) k = k () n
```

Effect Handlers

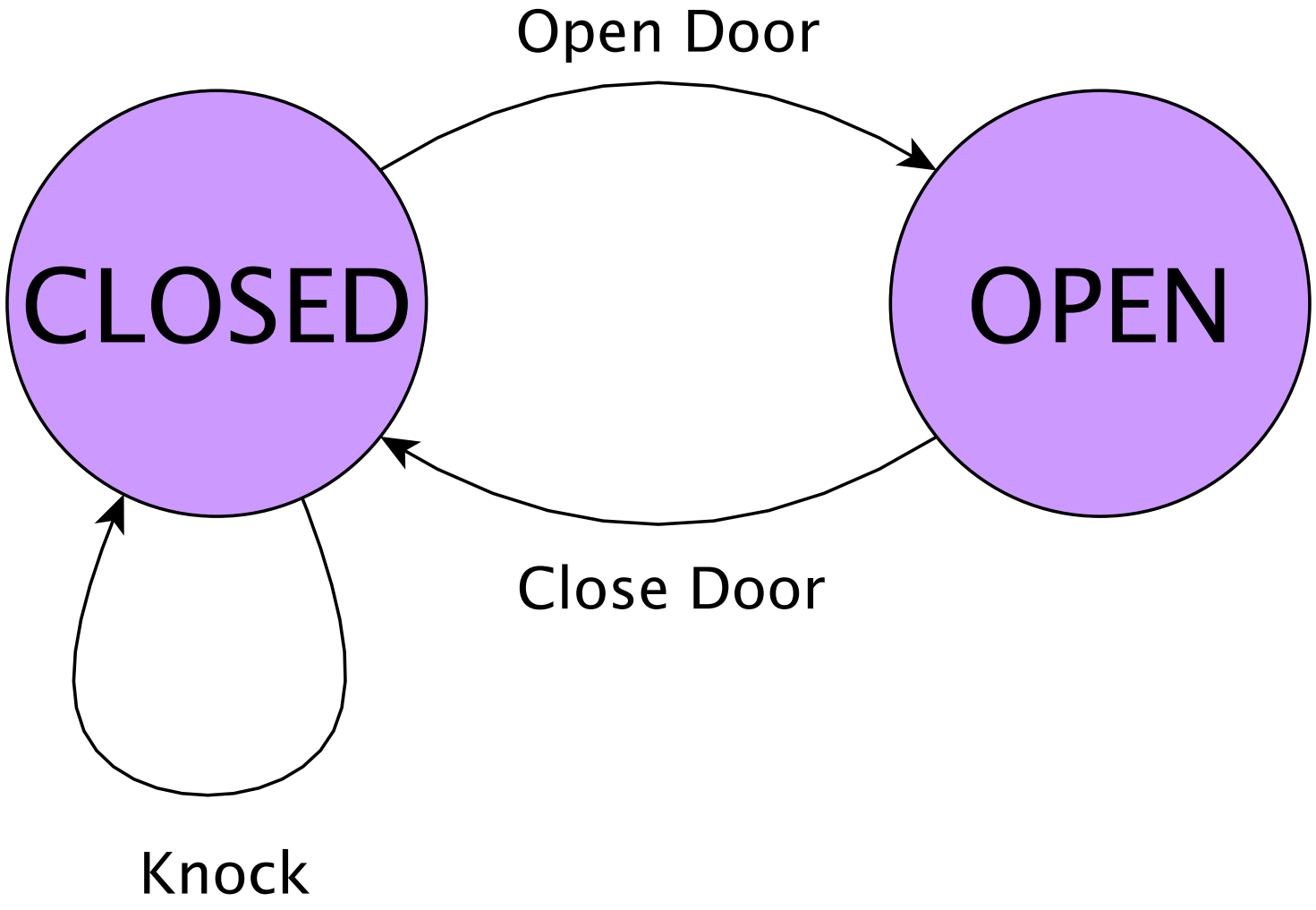
State

```
instance Handler State m where
     handle st Get      k = k st st
     handle st (Put n) k = k () n
```

StdIO

```
instance Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k x ()
    handle () (PutCh c)  k = do putChar c; k () ()
    handle () GetCh      k = do x <- getChar; k x ()
```

State machine example: Door opening



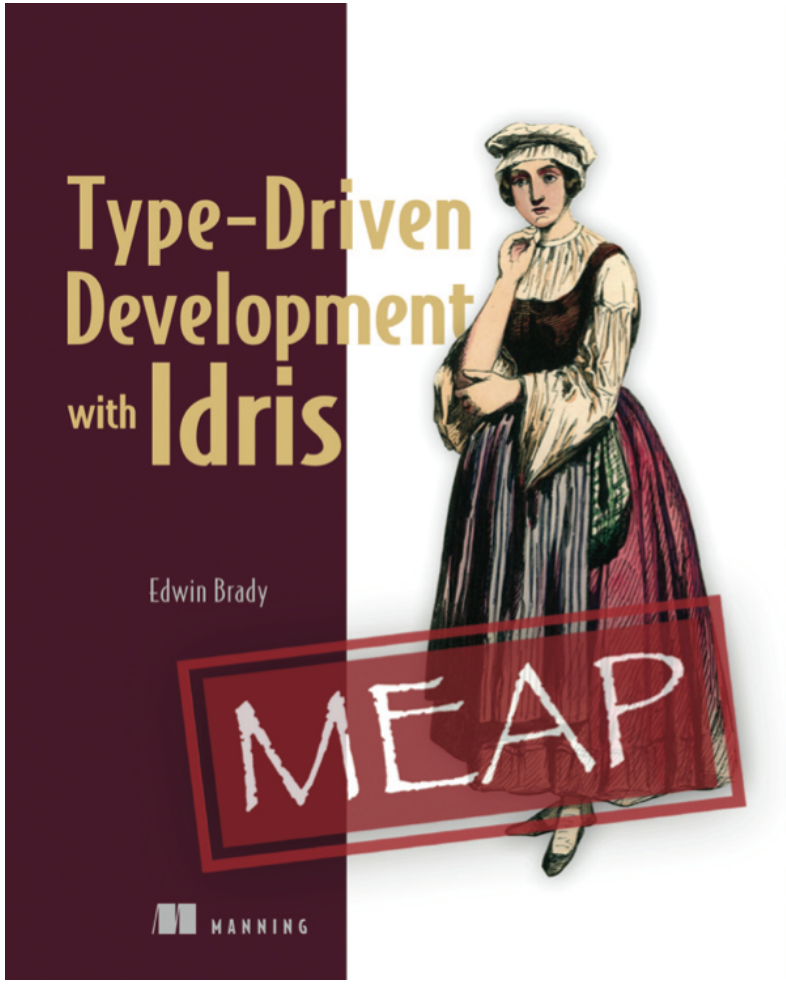
Demonstration: Door Protocol and Effects

Summary

Why are we interested in dependent types?

- *Safety*
  - Programs checked against precise specifications
- *Expressivity*
  - Better, more descriptive APIs
  - *Type directed* development
  - Type system should be *helping*, not telling you off!
- *Genericity*
  - e.g. program generation
- *Efficiency*
  - More precise type information should help the compiler
  - *Partial evaluation*, *erasure*

https://www.manning.com/books/type-driven-development-with-idris



Manning Deal of the Day, September 24th
50% discount
Discount code: dotd092415tw

sicsa*