Embedded Domain Specific Languages in Idris
Lecture 2: DSLs in Idris

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

SSGEP, Oxford, 7th July 2015

In Today's Lecture

Implementing *DSLs* in Idris

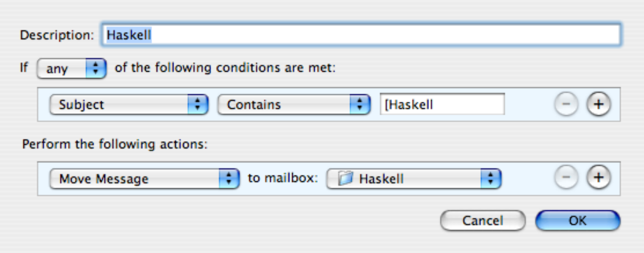
- Introductory examples:
    - A simple expression language
    - Simply typed λ calculus
- Partial Evaluation
- Resource management
    - State machines
    - A practical example: *Type Safe Concurrency*

Domain Specific Languages

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain
  - Very high level of abstraction
  - Typically *declarative*, i.e. say *what*, not *how*
  - Often *not Turing Complete*
- Examples:
  - Database and Internet applications — HTML, XML, SQL, . . .
  - Scientific programming — R, Mathematica
  - Computer games — UnrealScript
  - Hardware description — Verilog
  - Spreadsheet formulas

Domain Specific Languages

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain
  - Very high level of abstraction
  - Typically *declarative*, i.e. say *what*, not *how*
  - Often *not Turing Complete*
- Email filtering:



Description: Haskell

If any of the following conditions are met:

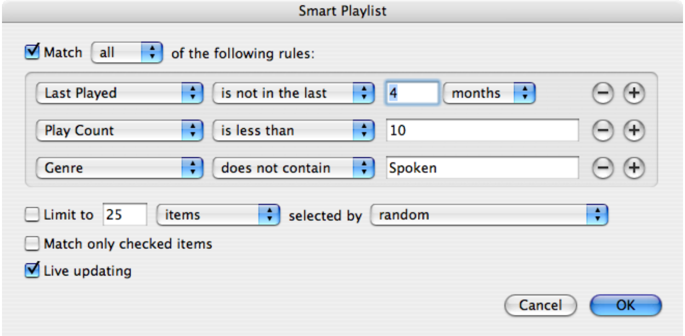
Subject Contains [Haskell

Perform the following actions:

Move Message to mailbox: Haskell

Domain Specific Languages

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain
  - Very high level of abstraction
  - Typically *declarative*, i.e. say *what*, not *how*
  - Often *not Turing Complete*
- Music playlists



Smart Playlist

☑ Match all of the following rules:

Last Played	is not in the last	4	months	⊖ ⊕
Play Count	is less than	10		⊖ ⊕
Genre	does not contain	Spoken		⊖ ⊕

□ Limit to 25 items selected by random

□ Match only checked items

☑ Live updating

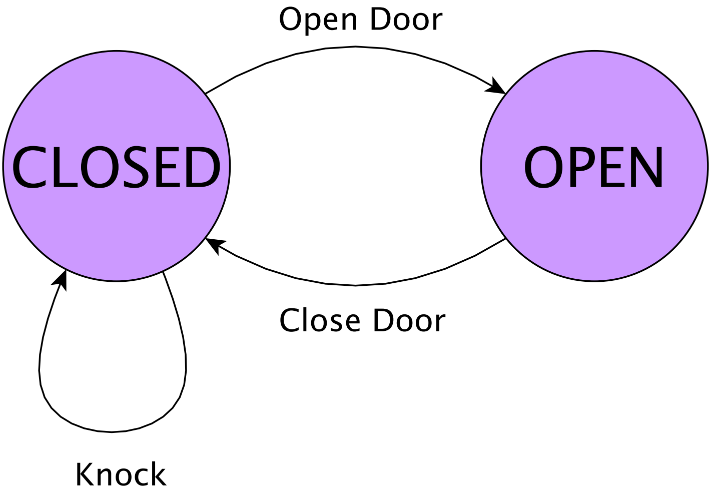
DSLs in IDRIS

IDRIS aims to support the implementation of *verified* domain specific languages. To illustrate this, we begin with an embedded interpreter for the *simply typed λ calculus*.

Idris

Demonstration: Introductory Examples

State machine example: Door opening



Idris

Demonstration: Door Protocol

sicsa*

Communication Protocol Actions

```
data Channel : p -> p -> Actions -> UniqueType

data Actions : Type where
     DoSend   : (dest : p) -> (pkt : Type) ->
                (cont : pkt -> Actions) -> Actions
     DoRecv   : (src : p) -> (pkt : Type) ->
                (cont : pkt -> Actions) -> Actions

     DoListen  : (client : p) -> Actions -> Actions
     DoRec    : Inf Actions -> Actions
     End      : Actions
```

Communication Protocols

Example

```
echo : Protocol [A, B] ()
echo = do Send A B String
          Send B A String
```

Communication Protocols

Example

```
echo : Protocol [A, B] ()
echo = do Send A B String
          Send B A String
```

Resulting actions

```
protoAs : (x : p) -> Protocol xs () -> Actions

*echo> protoAs A echo
DoSend B
       String
       (\c => DoRecv B String
       (\c1 => End)) : Actions
```

Communication Protocols

Example

```
echo : Protocol [A, B] ()
echo = do Send A B String
          Send B A String
```

Resulting actions

```
protoAs : (x : p) -> Protocol xs () -> Actions

*echo> protoAs B echo
DoRecv A
       String
       (\c => DoSend A String
       (\c1 => End)) : Actions
```

Communication Protocols

Example

```
echo : Protocol [A, B] ()
echo = do msg <- Send A B String
          Send B A (Literal msg)
```

Resulting actions

```
protoAs : (x : p) -> Protocol xs () -> Actions

*echo> protoAs A echo
DoSend B
       String
       (\msg => DoRecv B (Literal msg)
       (\c1 => End)) : Actions
```

Protocols

Client Implementation

```
echo_client : Client A B echo
echo_client s
    = do chan <- connect s
         print "Message: "
         msg <- getLine
         chan <- send msg chan
         (MkLit msg @@ chan) <- recv chan
         print ("ECHO: " ++ msg ++ "\n")
         close chan
```

Idris

Demonstration: Concurrent Echo Client/Server

sicsa*

Sample protocol

Protocol Description

```
utils : Protocol [A, B] ()
utils = do cmd <- Send A B UtilCmd
           case cmd of
                Mul => do Send A B (Int, Int)
                          Send B A Int
                StrLen => do Send A B String
                             Send B A Nat
                Uptime => Send B A Int
```

Idris

Demonstration: Utility Server

Summary

The combination of *dependent types* and *uniqueness type* allows:

- Precise descriptions of resource usage protocols. . .
- . . . with implementations verified by type checking

Concurrent DSL available from
`https://github.com/edwinb/ConcIO`

Knock Knock

```
joke : Protocol [A, B] ()
joke = do
  Send A B (Literal "Knock knock")
  Send B A (Literal "Who's there?")
  name <- Send A B String
  Send B A (Literal (name ++ " who?"))
  Send A B (punchline : String **
                     Literal (name ++ punchline))
```

Try it yourself

- `cabal update; cabal install idris`
  - OS X package available from `http://idris-lang.org/download`
- Concurrent DSL available from `https://github.com/edwinb/ConcIO`
- Demonstrations available at `https://github.com/edwinb/idris-demos`

Idris

Appendix: More Details

Channels

Unique Communication Channel

```
data Channel : (local : proc) ->
               (remote : proc) ->
               Actions -> UniqueType
```

Replicable Communication Channel

```
data RChannel : (remote : proc) ->
                Actions -> Type
```

Communication

As with the Door example, we define *commands* for manipulating unique channels, in a language CIO.

Commands

```
send : (val : a) -> Channel me t (DoSend t a k) ->
       CIO me xs xs (Channel me t (k val))

recv : Channel me t (DoRecv t a k) ->
       CIO me xs xs (Res a (\v => Channel me t (k v)))
```

Communication

Commands

```
listen : Channel me t (DoListen t k) ->
         {auto prf : Elem t xs} ->
         CIO me xs xs
           (Res Bool (\ok =>
             if ok then Channel me t k
                   else Channel me t (DoListen t k))

connect : RChannel t p ->
          CIO me xs (t :: xs) (Channel me t p)

close : Channel me t End ->
        {auto prf : Elem t xs} ->
        CIO me xs (dropElem xs prf) ()
```

Communication

```
Conc : Type -> Type -> Type
Conc p r = {xs : _} -> CIO p xs xs r

Server : (s, c : proc) -> Protocol [c, s] () -> AnyType
Server s c p = {xs : _} ->
    Channel s c (protoAs s (serverLoop c p)) ->
    CIO s (c :: xs) (c :: xs) Void

Client : (c, s : proc) -> Protocol [c, s] () -> AnyType
Client c s p = {xs : _} ->
    RChannel s (protoAs c p) ->
    CIO c xs xs ()
```