Dependent Types in the Idris Programming Language

Part 4: Type-level State Machines

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

OPLSS, July 4th 2017

Questions from Lecture 3 (1)

- Is putting states in types related to Session Types?

Questions from Lecture 3 (1)

- Is putting states in types related to Session Types?
- Is it possible to parameterise a type by whether it is infinite or lazy?

Questions from Lecture 3 (1)

- Is putting states in types related to Session Types?
- Is it possible to parameterise a type by whether it is infinite or lazy?
- Could `covering` or `total` be default rather than `partial`?

Questions from Lecture 3 (1)

- Is putting states in types related to Session Types?
- Is it possible to parameterise a type by whether it is infinite or lazy?
- Could `covering` or `total` be default rather than `partial`?
- Where can we learn more about theorem proving in Idris?

Questions from Lecture 3 (2)

- Is it possible to extract Idris to Haskell?

Questions from Lecture 3 (2)

- Is it possible to extract Idris to Haskell?
- What is the hardest/most annoying thing about implementing Idris?

Questions from Lecture 3 (2)

- Is it possible to extract Idris to Haskell?
- What is the hardest/most annoying thing about implementing Idris?
- What's your favourite undocumented feature of Idris?

Writing a Server

To create a server:

- Create a *socket*
- *Bind* the socket to a port
- *Listen* for connections to the socket
- *Accept* a connection:
    - This gives us a *new* socket for the connection
    - Continue *listening* on the original socket

Socket Programming (in C)

Creating a server

```
int socket(int domain, int type, int protocol);

int bind(int socket, const struct sockaddr *address,
        socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket,
           struct sockaddr *restrict address,
           socklen_t *restrict address_len);

int connect(int socket, struct sockaddr *address,
            socklen_t address_len)
/* ... */
```

Socket Programming (in C)

Which int is which? (Sockets)

```c
int socket(int domain, int type, int protocol);

int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket,
           struct sockaddr *restrict address,
           socklen_t *restrict address_len);

int connect(int socket, struct sockaddr *address,
            socklen_t address_len)
/* ... */
```

Socket Programming (in C)

Which int is which? (Errors)

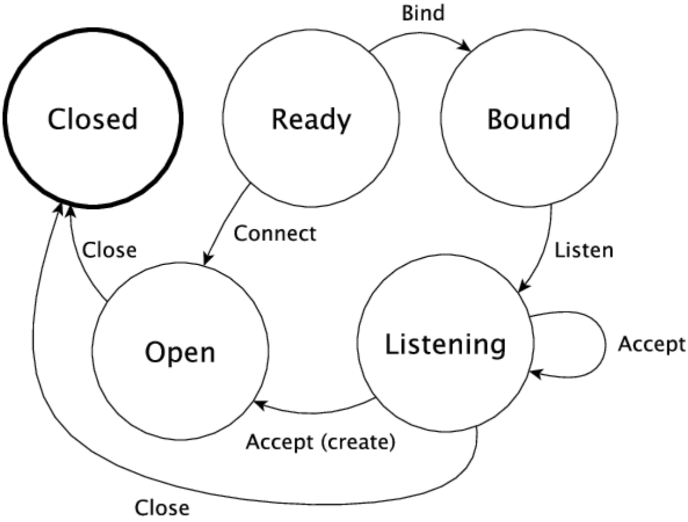
```c
int socket(int domain, int type, int protocol);

int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket,
           struct sockaddr *restrict address,
           socklen_t *restrict address_len);

int connect(int socket, struct sockaddr *address,
            socklen_t address_len)
/* ... */
```

Socket Programming (in Haskell)

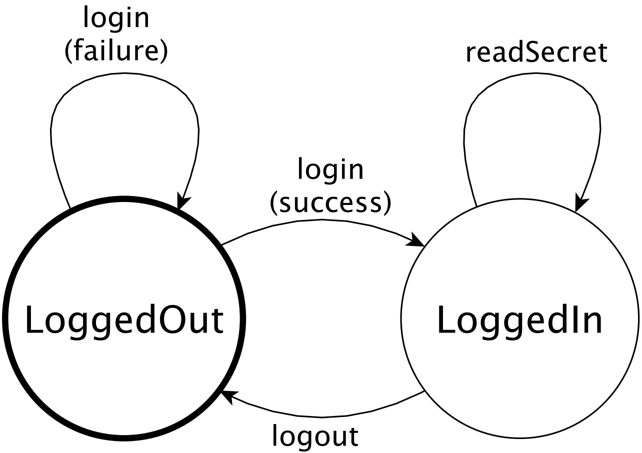
Creating a server

```
socket :: Family -> SocketType ->
          ProtocolNumber -> IO Socket

bind :: Socket -> SockAddr -> IO ()
listen :: Socket -> Int -> IO ()
accept :: Socket -> IO (Socket, SockAddr)

connect ::  Socket -> SockAddr -> IO ()
```

Socket States



Smaller example: A Data Store



Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)
- Describe how operations affect the *environment* (i.e. postconditions)

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)
- Describe how operations affect the *environment* (i.e. postconditions)
- Interoperate with *other stateful APIs*

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)
- Describe how operations affect the *environment* (i.e. postconditions)
- Interoperate with *other stateful APIs*
- Remain *Readable*

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)
- Describe how operations affect the *environment* (i.e. postconditions)
- Interoperate with *other stateful APIs*
- Remain *Readable*
- Lead to *helpful error messages*

Towards better APIs

Challenge: *How can we give descriptive APIs to stateful systems?*

Wishlist:

- Capture the *states* of the store (socket, file, channel . . .)
- Describe *when* operations are valid (i.e. preconditions)
- Describe how operations affect the *environment* (i.e. postconditions)
- Interoperate with *other stateful APIs*
- Remain *Readable*
- Lead to *helpful error messages*

Inspiration: *Separation logic*, *Ynot*, *Indexed Monads*, *Linear Types* . . .

State in Idris: ST

The type of stateful programs

```
ST : (m : Type -> Type) -- Underlying context
  -> (ty : Type)        -- Result type
  -> List (Action ty)   -- State transitions
                        -- may depend on the result
  -> Type
```

State in Idris: ST Example

A DataStore Program

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             OK <- login st
               | BadPassword => do putStrLn "Failure"
                                   disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

State in Idris: `STrans`

`ST` is a type level function. `List (Action ty)` translates into:

- *input resources*
- *output resources*, calculated from the result of the operation

State in Idris: STrans

ST is a type level function. List (Action ty) translates into:

- *input resources*
- *output resources*, calculated from the result of the operation

The type of stateful programs

```
STrans : (m : Type -> Type) -- Underlying context
      -> (ty : Type)        -- Result type
      -> Resources          -- Input states
      -> (ty -> Resources)  -- Output states
                         -- may depend on the result
      -> Type
```

Resources in ST

A resource

```
data Resource : Type where
     (:::) : Var -> Type -> Resource
```

Some example resource lists

```
[]
[door ::: Door Closed, count ::: State Int]
[d1 ::: Door Closed, d2 ::: Door Open,
 st ::: Store LoggedIn]
```

Actions in ST

```
data Action : Type -> Type where
     Stable : Var -> Type -> Action ty
     Trans : Var -> Type -> (ty -> Type) -> Action ty
     Remove : Var -> Type -> Action ty
     Add : (ty -> Resources) -> Action ty
```

Actions in ST

```
data Action : Type -> Type where
     Stable : Var -> Type -> Action ty
     Trans : Var -> Type -> (ty -> Type) -> Action ty
     Remove : Var -> Type -> Action ty
     Add : (ty -> Resources) -> Action ty
```

Example actions

```
Stable st (Store LoggedIn)
Trans st (Store LoggedIn) (const (Store LoggedOut))
Remove st (Store LoggedOut)
Add (\var => [var ::: Store LoggedOut])
```

Actions in ST

```
data Action : Type -> Type where
     Stable : Var -> Type -> Action ty
     Trans : Var -> Type -> (ty -> Type) -> Action ty
     Remove : Var -> Type -> Action ty
     Add : (ty -> Resources) -> Action ty
```

Example actions, alternative notation

```
st ::: Store LoggedIn
st ::: Store LoggedIn :-> Store LoggedOut
remove st (Store LoggedOut)
add (Store LoggedOut)
```

Interfaces and Implementations

General idea: Define *interfaces* for operations in ST:

- Abstract away underlying *resource types* (`Store` here)
- Implement interface for different *contexts*
- Use only needed interfaces

Interfaces and Implementations

General idea: Define *interfaces* for operations in ST:

- Abstract away underlying *resource types* (`Store` here)
- Implement interface for different *contexts*
- Use only needed interfaces

The DataStore Interface

```
interface DataStore (m : Type -> Type) where
    ...

implementation DataStore IO where
    ....
```

State in Idris: ST Example

Reading the secret

```
readSecret : (store : Var) ->
             ST m String [store ::: Store LoggedIn]
```

State in Idris: ST Example

Reading the secret

```
readSecret : (store : Var) ->
             ST m String [store ::: Store LoggedIn]
```

Logging out

```
logout : (store : Var) ->
         ST m () [store ::: Store LoggedIn :->
                            Store LoggedOut]
```

State in Idris: ST Example

Logging in (almost. . .)

```
login : (store : Var) ->
   ST m () [store ::: Store LoggedOut :->
                      Store LoggedIn]
```

State in Idris: ST Example

Logging in (with possible failure)

```
data LoginResult = OK | BadPassword

login : (store : Var) ->
      ST m Result
        [store ::: Store LoggedOut :->
          (\res => Store (case res of
                               OK => LoggedIn
                               BadPassword => LoggedOut))]
```

State in Idris: ST Example

Connecting to a store

```
connect : ST m Var [add (Store LoggedOut)]
```

Interfaces and Implementations

The DataStore Interface

```
interface DataStore (m : Type -> Type) where
    Store : Access -> Type
    connect : ST m Var [add (Store LoggedOut)]
    disconnect : (store : Var) ->
          ST m () [remove store (Store LoggedOut)]

    {- ... login, logout, readSecret elided ... -}
```

Implementing DataStore

To implement an interface, we need a *concrete* representation for Store:

The DataStore Interface

```
implementation DataStore IO where
    Store _ = State String

    {- ... -}
```

We also need the ability to *create*, *destroy*, *read* and *write* concrete resources.

Internals: Creating and destroying resources

We can create `new` resources, and `delete` resources, of type `State ty`:

Creating and destroying

```
new : (val : state) ->
        STrans m Var res
              (\lbl => (lbl ::: State state) :: res)

delete : (lbl : Var) ->
         {auto prf : InState lbl (State st) res} ->
         STrans m () res (const (drop res prf))
```

Internals: Reading and writing resources

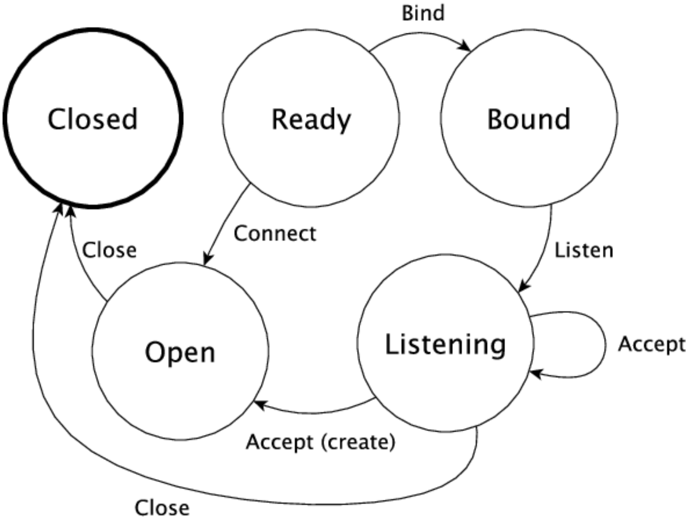
We can `read` and `write` resources as long as we know they are of type `State ty`:

Reading and writing

```
read : (lbl : Var) ->
       {auto prf : InState lbl (State ty) res} ->
       STrans m ty res (const res)
write : (lbl : Var) ->
       {auto prf : InState lbl ty res} ->
       (val : ty') ->
       STrans m () res
               (const (updateRes res prf (State ty')))
```

Demonstrations: States in Action

Socket States, Revisited

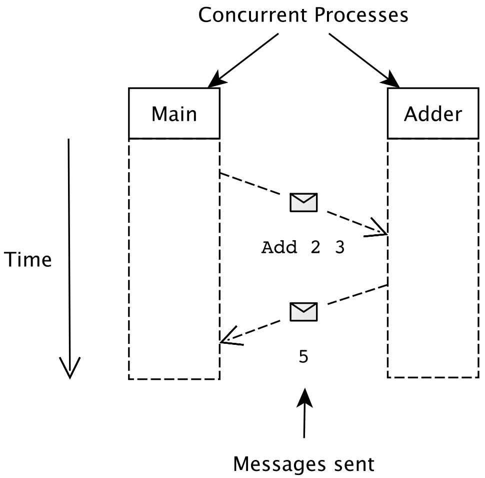


Concurrency in Idris

The Idris run-time system supports *message passing* concurrency

- A process can `spawn` another process
- A process can create a `Channel`, using:
  - `connect`, which initiates a connection to another process
  - `listen`, which waits for incoming connections
- Processes can `send` and `receive` messages on a `Channel`

Message Passing Concurrency in Idris



To write correct concurrent programs in this style, we'd like to ensure, at least:

- *Requests* (like `Add 2 3`) and *Responses* (like `5`) are well-typed w.r.t. each other
- *Server* processes (like `Adder`) run indefinitely
- Server processes always complete responses to requests

...

Message Passing Concurrency in Idris

To write correct concurrent programs in this style, we'd like to ensure, at least:

- *Requests* (like `Add 2 3`) and *Responses* (like `5`) are well-typed w.r.t. each other
- *Server* processes (like `Adder`) run indefinitely
- Server processes always complete responses to requests

. . .
A job for *Session Types*?

Session types:

- Describe the *state* of a communication channel
- i.e. Track the *expected sequence* of communications on a channel

Session Types and ST

Session types:

- Describe the *state* of a communication channel
- i.e. Track the *expected sequence* of communications on a channel

`ST` can capture this! We can:

- Create threads
- Create channels for communicating between threads
- Give types to channels for tracking communication state

Session Types and ST

Channel States

```
data Actions : Type where
     Send : (a : Type) -> (a -> Actions) -> Actions
     Recv : (a : Type) -> (a -> Actions) -> Actions
     Done : Actions
```

Session Types and ST

Sending and Receiving

```
send : (chan : Var) -> (val : ty) ->
        ST m () [chan ::: Channel (Send ty f) :->
                          Channel (f val)]
recv : (chan : Var) ->
        ST m ty [chan ::: Channel (Recv ty f) :->
                    (\res => Channel (f res))]
```

Demonstration: Concurrency with `ST`