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
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?

Could covering or total be default rather than partial?
Where can we learn more about theorem proving in Idris?
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?
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?
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
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);
/* ... */
```
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)
/* ... */
```
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

- Closed
- Ready
- Bound
- Open
- Listening

Transitions:
- Bind
- Connect
- Listen
- Accept
- Accept (create)
- Close
Smaller example: A Data Store

- LoggedOut
  - login (failure)
  - logout
- LoggedIn
  - readSecret
  - login (success)
Towards better APIs

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

Wishlist:
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)
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)
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*
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*
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*
The type of stateful programs

\[ \text{ST} : (m : \text{Type} \rightarrow \text{Type}) \rightarrow (\text{ty} : \text{Type}) \rightarrow \text{List} (\text{Action ty}) \rightarrow \text{Type} \]

- Underlying context
- Result type
- State transitions
- may depend on the result
State in Idris: ST Example

A DataStore Program

```idris
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
```

ST is a type level function. \texttt{List (Action ty)} translates into:

- \textit{input resources}
- \textit{output resources}, calculated from the result of the operation
STrans : (m : Type -> Type) -- Underlying context
  -> (ty : Type) -- Result type
  -> Resources -- Input states
  -> (ty -> Resources) -- Output states
  -- may depend on the result
  -> Type

The type of stateful programs

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

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

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

Some example resource lists

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

```plaintext
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

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

Logging out

```idris
logout : (store : Var) ->
    ST m () [store ::: Store LoggedIn :->
        Store LoggedOut]
```
State in Idris: ST Example

Logging in (almost...)

\[
\text{login} : (\text{store} : \text{Var}) \rightarrow \\
\quad \text{ST m (()) [store ::: \text{Store LoggedOut :->}} \\
\quad \quad \text{Store LoggedIn]}
\]
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))]]
Connecting to a store

connect : ST m Var [add (Store LoggedOut)]
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 ... -}
To implement an interface, we need a *concrete* representation for Store:

### The DataStore Interface

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

{- ... -}
```

We also need the ability to *create*, *destroy*, *read* and *write* concrete resources.
We can create **new** resources, and **delete** resources, of type **State ty**: 

**Creating and destroying**

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

<table>
<thead>
<tr>
<th>Reading and writing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong> : (lbl : Var) -&gt;</td>
</tr>
<tr>
<td>{auto prf : InState lbl (State ty) res} -&gt;</td>
</tr>
<tr>
<td>STrans m ty res (const res)</td>
</tr>
<tr>
<td><strong>write</strong> : (lbl : Var) -&gt;</td>
</tr>
<tr>
<td>{auto prf : InState lbl ty res} -&gt;</td>
</tr>
<tr>
<td>(val : ty') -&gt;</td>
</tr>
<tr>
<td>STrans m () res</td>
</tr>
<tr>
<td>(const (updateRes res prf (State ty')))</td>
</tr>
</tbody>
</table>
Demonstrations: States in Action
Socket States, Revisited

Diagram:
- **Closed**
  - Connect to **Ready**
- **Ready**
  - Bind to **Bound**
  - Connect to **Open**
- **Open**
  - Close to **Closed**
  - Accept to **Listening**
- **Bound**
  - Listen to **Open**
- **Listening**
  - Accept (create) to **Bound**
  - Close to **Closed**

Arrows indicate transitions between states.
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

...
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**

\[
\text{send} : (\text{chan} : \text{Var}) \rightarrow (\text{val} : \text{ty}) \rightarrow \\
\text{ST m () [chan ::: Channel (Send ty f) :-> Channel (f val)]}
\]

\[
\text{recv} : (\text{chan} : \text{Var}) \rightarrow \\
\text{ST m ty [chan ::: Channel (Recv ty f) :-> (\res => Channel (f res))]}\]