Type Driven Development with Idris

Lecture 3: Interaction and Concurrency

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

MGS, April 11th 2018

Interactive Programs in Idris

Idris, like Haskell, uses `IO` for writing interactive programs

- A value of type `IO ty` is a *description* of an interactive action which results in a value of type `ty`

Example: Sequencing IO Actions

```
hello : IO ()
hello = do putStr "What is your name? "
           name <- getLine
           putStr ("Hello " ++ name)
```

Interactive Programs in Idris

- Problem: we often want interactive programs to run *indefinitely*

Example: Looping IO Actions

```
loopy : IO ()
loopy = do putStr "What is your name? "
           name <- getLine
           putStr ("Hello " ++ name)
           loopy   -- Not total!
```

- Composing actions in a recursive function may not be *total*
  - No structurally decreasing argument, in general

Solution: *Describe* looping programs as a *tree* of `IO` actions:

```
data InfIO : Type where
     Do : IO a -> (a -> Inf InfIO) -> InfIO

(>>=) : IO a -> (a -> Inf InfIO) -> InfIO
(>>=) = Do
```

Then define a run function to *execute* those descriptions:

```
run : InfIO -> IO ()
```

Interactive Total Functional Programs

Then define a `run` function to *execute* those descriptions:

```
run : InfIO -> IO ()
```

Compare with `IO`:

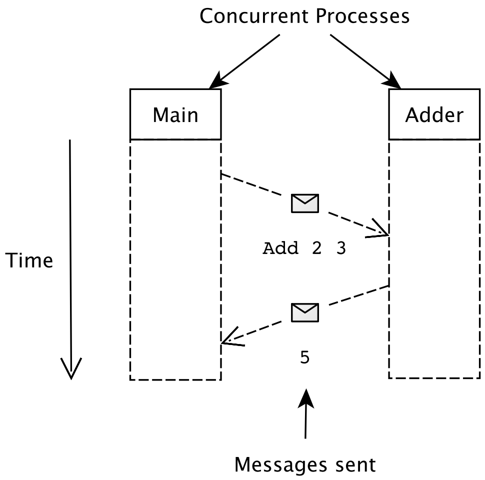
- `IO ty` is a description of actions which result in a `ty`
- The run-time system *executes* those actions
- `run` on `InfIO` does a similar job, at a different level

Example: Concurrency

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`

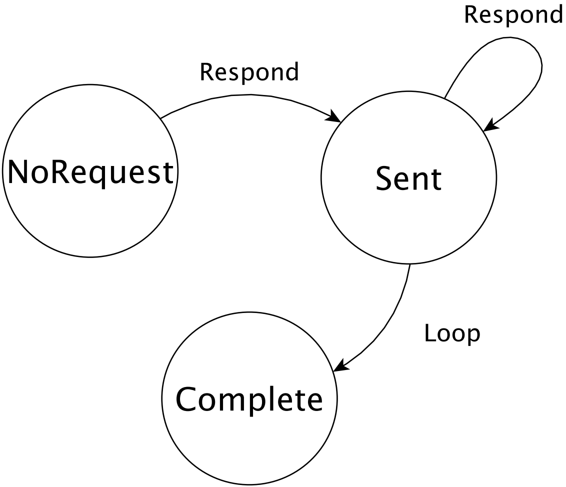
Message Passing Concurrency in Idris



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
  - That is, they are *productive*
- Server processes always complete responses to requests
  - That is, processing a response *terminates*

Server Processes



Types for Message Passing

We can achieve this with types:

- Define a type for *Requests*
- Define a function to calculate *Response* types from requests
  - This describes valid message types for interactions between processes

Types for Message Passing

We can achieve this with types:

- Define a type for *Requests*
- Define a function to calculate *Response* types from requests
  - This describes valid message types for interactions between processes
- Define a type for servers, parameterised by the *Request* and *Response* types it services
  - This defines the type of messages we can send to a process
  - Like `InfIO`, a process is an infinite sequence of commands
  - Like `InfIO`, it guarantees *productivity*
  - Processes run indefinitely, and always complete requests

Types for Message Passing

Adder Requests/Responses

```
data Request = Add Nat Nat

Response : Request -> Type
Response (Add x y) = Nat
```

Adder Implementation

```
adder : ServerLoop Response ()
adder = do Accept (\msg =>
                    case msg of
                         Add x y => Pure (x + y))
           Loop adder
```

Demonstration

Concurrent Processes in Action

Further Reading

On *total* functional programming:

- David Turner, *Elementary Strong Functional Programming*, 2005

On *interactive* programming with dependent types

- Peter Hancock and Anton Setzer, *Interactive Programs in Dependent Type Theory*, 2000

On types for *communicating systems*:

- Kohei Honda, *Types for Dyadic Interaction*, 1993
- Kohei Honda, Nobuko Yoshida, Marco Carbone, *Multiparty Asynchronous Session Types*, 2008
- Philip Wadler, *Propositions as Sessions*, 2012

Summary

- *Total* programs are either *terminating* or *productive*
  - Together, this allows us to write long running processes, where every *request* is processed in finite time
- A useful pattern for concurrent programming is to:
  - Define *server* processes which respond to *requests*
  - Write programs as a collection of client processes, making remote procedure calls to servers
- We can define long running, well typed, concurrent processes as potentially infinite streams of commands
- Using dependent types (in particular, *first class functions*), we've described simple message passing protocols