Abstract
Modern software systems rely on communication, for example mobile applications communicating with a central server, distributed systems coordinating a telecommunications network, or concurrent systems handling events and processes in a desktop application. However, reasoning about concurrent programs is hard, since we must reason about each process and the order in which communication might happen between processes. In this paper, I describe a type-driven approach to implementing communicating concurrent programs, using the dependently typed programming language Idris. I show how the type system can be used to describe resource access protocols (such as controlling access to a file handle) and verify that programs correctly follow those protocols. Finally, I show how to use the type system to reason about the order of communication between concurrent processes, ensuring that each end of a communication channel follows a defined protocol.

Keywords
Dependent Types, Domain Specific Languages, Verification, Concurrency
1. Introduction

Implementing communicating concurrent systems is hard, and reasoning about them even more so. Nevertheless, the majority of modern software systems rely to some extent on communication, whether over a network to a server such as a web or mail server, between peers in a networked distributed system, or between processes in a concurrent desktop system. Reasoning about concurrent systems is particularly difficult; the order of processing is not known in advance, since processes are running independently.

In this paper, I describe a type-driven approach to reasoning about message passing concurrent programs, using a technique similar to Honda’s Session Types [14]. A Session Type describes the state of a communication channel, in particular the expected sequence of communication over that channel. I will use the dependently typed programming language Idris [4] to implement typed communication channels, parameterised by their state, exploiting the Idris type checker to verify that systems communicating over those channels correctly implement a communication protocol and correctly coordinate with other processes.

By type-driven, I mean that the approach involves writing an explicit type describing the pattern of communication, and verifying that processes follow that pattern by type-checking. Communication channels are explicitly parameterised by their state; operations on a channel require a channel in the correct state, and return a channel with an updated state. Hence, a well-typed program working with a communication channel is guaranteed to follow the correct protocol for that channel.

1.1. The Idris Programming Language

Idris is a purely functional language with dependent types. The syntax is heavily inspired by Haskell [19]. Like Haskell, Idris supports algebraic data types with pattern matching, type classes, and syntactic conveniences such as list comprehensions and do notation. Unlike Haskell, however, Idris is evaluated eagerly by default, and supports full spectrum dependent types. This means that types may be predicated on any value; hence, properties of a program can be expressed in a type and verified by type checking. Furthermore, Idris supports uniqueness types. A value with a unique type has the property that at run-time there is guaranteed to be at most one reference to that value. As such, a value with a unique type can be used to represent a reference to a resource in a specific state. Throughout the paper I will introduce features of Idris as necessary; a full tutorial is available elsewhere [22].

1.2. Contributions

I will show how combining dependent types and uniqueness types, as implemented in Idris, can support type-safe message passing concurrent programs. I make the following specific contributions:
I present a method for describing resource access protocols in the type system, using dependent types to capture resource state, and uniqueness types to guarantee non-aliasing of resources.

I present a language for describing multi-party communication protocols, and show how protocols are translated into resource protocols for each participant.

I outline a library of type safe concurrent programming primitives, built on the protocol description language.

The overall approach involves implementing domain specific languages (DSLs), embedded in Idris, using the type system to capture properties of programs in those DSLs which are then verified by the Idris type checker. The type system itself is expressive enough to describe and formally verify properties of concurrent programs, without any plugins or extensions. The code and examples in this paper are available in full online, at http://github.com/edwinb/ConcIO.

1.3. Introductory Example: an Echo Server

Consider an “echo server” protocol, in which a client sends a server a string, and the server responds by echoing the string back, following by the length of the string as a natural number. In our communication protocol DSL, this protocol can be described as in Listing 1.

Listing 1: Echo Protocol

```
|echo : Protocol ['C, 'S] ()
|echo = do msg <- 'C == > 'S | String
|                         'S == > 'C | Literal msg
|                         'S == > 'C | Nat
```

A Protocol description is parameterised on a list of participants, here labelled 'S for the server and 'C for the client. In the echo protocol, we have bound msg to the string sent from the client to the server, meaning that we can refer to the string later in the protocol. The type Literal s represents a literal string which is required to have the value s, meaning here that the server must echo the string back to the client. There are no restrictions on the natural number, however.

A sample implementation of the server end of this protocol is shown in Listing 2. I will briefly summarise this implementation; full details are deferred until Section 3.

The server begins by listening on a channel chan. If there is a value waiting on the channel, the protocol can proceed, otherwise the server will wait. Throughout this implementation, we use chan to refer to the communication channel. Each operation which uses the channel returns a new channel, for example:

```haskell
(msg @@ chan) <- recv chan
```
Listing 2: Echo Server Implementation

```haskell
echo_server : Server 'S 'C echo
echo_server chan
  = do (True @@ chan) <- listen chan  
     | (False @@ chan) => echo_server chan
     (msg @@ chan) <- recv chan
     chan <- send (MkLit msg) chan
     chan <- send (length msg) chan
     let chan = reset chan
     echo_server chan
```

The result of `recv chan` is a message, `msg`, paired with a new channel `chan`. The type system ensures that there is at most one use of each instance of `chan`; once a value has been sent or received over a channel, it cannot be used again.

A sample client is given in Listing 3. The client connects to a server channel, reads a message from the keyboard, then follows the client end (`'C`) of the `echo` protocol and outputs the results received from the server. Finally, it must `close` the connection to the server.

Listing 3: Echo Client Implementation

```haskell
echo_client : Client 'C 'S echo
echo_client s
  = do chan <- connect s
      print "Message: "
      msg <- getLine
      chan <- send msg chan
      (MkLit msg @@ chan) <- recv chan
      (len @@ chan) <- recv chan
      print (msg ++ "(" ++ show len ++ ")\n")
      close chan
```

In both cases, client and server, any violation of the protocol or repeated use of a communication channel would result in a type error. For example, if we attempt to `close` the channel before the final `recv`, Idris reports an error such as:

```
echo.idr:25:16:
When elaborating right hand side of case block in echo_client:
  When elaborating argument c to function Channels.close:
    End of channel used when Recv from 'S required
```

2015/02/24; 00:02 str. 4/21
Finally, we can write a concurrent main program, running from the point of view of the client ('C) which starts a server running according to the echo protocol, then repeatedly sends requests to the server via the echo_client:

```haskell
conc_main : Conc 'C ()
conc_main = do h <- start_server echo echo_server
            client_loop h
            where client_loop h = do echo_client h
                           client_loop h
```

In the rest of this paper, I will describe how to build a domain specific language for concurrent programs. Before we deal with concurrency itself, however, we will need to establish how to manage resources with associated state and resource access protocols.

2. Resource Management in Idris

![State machine representing a door](image)

Figure 1. State machine representing a door

Consider a state machine representing the states and actions on a door, as shown in Figure 1. A door can be in one of two states, OPEN or CLOSED, with the following actions available:

- **Knock**, which is valid when the door is closed, and does not change the door’s state.
- **OpenDoor**, which is valid when the door is CLOSED, and changes the door’s state to OPEN.
- **CloseDoor**, which is valid when the door is OPEN, and changes the door’s state to CLOSED.
In this section, we explore how to represent and implement this state machine in Idris. Our goal is to ensure via the type system that no program can violate the protocol.

2.1. First Attempt: Parameterised Handle

We will use a handle\(^1\) (much like a file handle) to access a door’s current state. As a first attempt, let us try explicitly capturing the state in the type of the handle (we keep the definition of DoorH abstract):

\[
\text{data DoorState} = \text{Opened} \mid \text{Closed}
\]

\[
\text{data DoorH} : \text{DoorState} \to \text{Type}
\]

We describe the possible actions on a door using the DoorCmd data type below.

\[
\text{data DoorCmd} : \text{Type} \to \text{Type where}
\]

\[
\begin{align*}
\text{OpenDoor} & : \text{DoorH Closed} \to \text{DoorCmd (DoorH Opened)} \\
\text{Knock} & : \text{DoorH Closed} \to \text{DoorCmd ()} \\
\text{CloseDoor} & : \text{DoorH Opened} \to \text{DoorCmd (DoorH Closed)}
\end{align*}
\]

Each action is restricted to running on handles in the valid state: we can only open a closed door, or close an open door. We use a data type here, rather than directly implementing these as functions for one principal reason: it allows us to implement the actions in different ways for different execution contexts. For example, in this case, one implementation may send an electronic signal to an automatic door; another implementation may be a computer simulation. This follows the algebraic effects and handlers [20, 15, 5, 6] approach to implementing side-effecting programs in a pure language.

These actions are then included in a DoorLang DSL, which essentially allows us to sequence operations on a door handle:

\[
\text{data DoorLang} : \text{Type} \to \text{Type where}
\]

\[
\begin{align*}
\text{Return} & : a \to \text{DoorLang a} \\
\text{Action} & : \text{DoorCmd a} \to \text{DoorLang a} \\
(\gg=) & : \text{DoorLang a} \to (a \to \text{DoorLang b}) \to \text{DoorLang b}
\end{align*}
\]

Aside: Like Haskell, Idris supports do notation which desugars to \(\gg=\). Unlike Haskell, Idris does not require this to be part of a Monad instance. While type classes (including Monad) are available, we often require more flexibility in the type of the \(\gg=\) operator.

The following program is a valid use of the door protocol; the program knocks, opens, then closes the handle, before finally returning:

---

\(^1\)Pun intended
doorOK : DoorH Closed -> DoorLang ()
doorOK h = do Action (Knock h)

h <- Action (OpenDoor h)

h <- Action (CloseDoor h)

Return ()

Unfortunately, there is a problem. The following program is also a valid program in DoorLang despite not being a valid use of the door protocol:

doorBad : DoorH Closed -> DoorLang ()
doorBad h = do Action (Knock h)

hbad <- Action (OpenDoor h)

h <- Action (CloseDoor hbad)

h <- Action (CloseDoor hbad)

Return ()

The handle hbad has been closed twice! If the handle is implemented as a direct link to a real world resource, this is an invalid operation and may lead to an invalid state. The problem is that once the handle has been used, it should be considered no longer valid.

2.2. Second Attempt: Unique Handles

For our second attempt, we still use a handle but declare it as UniqueType rather than Type:

data DoorH : DoorState -> UniqueType

The full typing rules for UniqueType follow those in the Clean language [23] and are outside the scope of this paper. Intuitively, however, if there is a variable x : t, and t : UniqueType, then x can be referred to at most once after it is bound. As a result, we can be certain that at run-time there is only at most one live reference to x. The commands and language are declared as before, with two minor variations:

data DoorCmd : Type* -> Type* where

OpenDoor : DoorH Closed -> DoorCmd (DoorH Opened)
Knock : DoorH Closed -> DoorCmd (DoorH Closed)
CloseDoor : DoorH Opened -> DoorCmd (DoorH Closed)


data DoorLang : Type* -> Type*

... Firstly, the types are parameterised by Type* rather than Type, and secondly the Knock operation returns a handle rather than a value of the unit type ()

The type Type* can be read as “either UniqueType or Type.” Conservatively, the type checker assumes that any polymorphic variable with type Type* will be instantiated by a UniqueType, and therefore must also be referenced at most once. We now write doorOK as follows, returning the handler after Knock:
doorOK : DoorH Closed → DoorLang ()
doorOK h = do h ← Action (Knock h)
    h ← Action (OpenDoor h)
    h ← Action (CloseDoor h)
    Return ()

This program is valid: h appears several times but is reassigned after each action to the result of the action, so every use is unique. A program such as doorBad above, however, no longer type checks, giving the error:

Door.idr:22:10:Unique name hbad is used more than once

This is a big improvement: by preventing aliasing using a uniqueness type we are required to follow the door protocol by always using the handle output from one operation as the input to the next. However, there are still some unanswered questions which arise when using protocols in practice:

1. What happens if an operation fails? For example, what if the door jams when we try to open it?
2. We have ensured that operations are valid, but how do we ensure that a protocol is run to completion?

2.3. Third attempt: Managing Failure

The resulting state of a handle may be different depending on the success or failure of an operation. For example, if a door might be jammed, then OpenDoor may result in one of two states, depending on the result, according to the following rules:

1. If the door opens successfully, the state is Opened
2. If the door jams, the state is Closed

Therefore, OpenDoor should not only return whether the operation was successful and the new handle, but also guarantee that the result and the new handle are consistent with these rules. We use the following dependent pair type to capture pairs of values where the type of the second element is computed from the value of the first:

```haskell
data Res : (a : Type*) → (a → Type*) → Type* where
    (@@) : (val : a) → k val → Res a k
```

For example, we can have a Bool paired with either an Int or a String, depending on the value of the Bool:

```haskell
intOrString : Type*
intOrString = Res Bool (\ok => if ok then Int
                           else String)
```

If the first element is True, the second element must be an Int, otherwise it must be a String. Therefore, (True @@ 42) and (False @@ "No") are valid elements of this type, but (True @@ "Yes") and (False @@ 94) are not. We can use this type
to calculate the resulting state of a door handle based on run-time information about whether \texttt{OpenDoor} has succeeded or not:

\begin{verbatim}
data DoorCmd : Type* -> Type* where
  OpenDoor : DoorH Closed ->
    DoorCmd (Res Bool
      (\ok => if ok
        then DoorH Opened
        else DoorH Closed))
  Knock : DoorH Closed -> DoorCmd (DoorH Closed)
  CloseDoor : DoorH Opened -> DoorCmd (DoorH Closed)

In order to use the result from \texttt{OpenDoor}, we must inspect the first (\texttt{Bool}) element of the pair, which will determine the state of the handle. If opening the door fails, we return, otherwise we continue with the protocol as previously:

\begin{verbatim}
doorOK : DoorH Closed -> DoorLang ()
doorOK h = do h <- Action (Knock h)
  hres <- Action (Open h)
  case hres of
    False @@ h => Return ()
    True @@ h => do
      Action (Close h)
      Return ()
\end{verbatim}

Idris provides a notation for pattern matching in do notation which can be more convenient when dealing with failure, as follows:

\begin{verbatim}
doorOK : DoorH Closed -> DoorLang ()
doorOK h = do h <- Action (Knock h)
  (True @@ h) <- Action (Open h)
  | (False @@ h) => Return ()
    Action (Close h)
  Return ()
\end{verbatim}

This is exactly equivalent to the definition with the explicit case. In general, in a do-block, the syntax...

\begin{verbatim}
do pat <- val | <alternatives>
  p

...is desugared to...

do x <- val
  case x of
    pat => p
    <alternatives>
\end{verbatim}
Finally, we can ensure the protocol is run to completion by changing the type of `doorOK` to return the door handle. There are only two ways for `doorOK` to return a closed door handle: either returning immediately, or running the protocol to completion:

```haskell
doorOK : DoorH Closed -> DoorLang (DoorH Closed)
doorOK h = do h <- Action (Knock h)
    (True @@ h) <- Action (Open h)
    | (False @@ h) => Return h
    Action (Close h)
    Return h
```

3. Communication Protocols

In this section, I will show how to use the approach described in Section 2 to track the state of a channel in a communicating system. The goal is to ensure that all participants in the system are working to the same overall script. When spawning a new process, we ensure that communication between it and its parent is synchronised.

3.1. Channels

A `Channel` is parameterised by its end points (the local and remote processes) and the remaining messages to be transmitted on the channel (the `Actions`):

```haskell
data Channel : (src : proc) -> (dest : proc) -> Actions -> UniqueType
```

This is a `UniqueType`, meaning that a channel in a particular state can be used at most once. The local (`src`) and remote (`dest`) processes have type `proc`, which is a type variable, and essentially serves as a way of labelling processes. When creating a server we may wish to have several client processes connecting to a single server. For this case, we allow replicable channels, declared as follows:

```haskell
data RChannel : (dest : proc) -> Actions -> Type
```

No message can be sent directly on an `RChannel`; however, an unlimited number of `Channels` can be created from an `RChannel`. Each of `Channel` and `RChannel` is parameterised by the remaining actions in a process; `Actions` is declared as follows:

```haskell
data Actions : Type where
    DoListen : (client : proc) -> Actions -> Actions
    DoSend : (dest : proc) ->
        (a : Type) -> (a -> Actions) -> Actions
    DoRecv : (src : proc) ->
        (a : Type) -> (a -> Actions) -> Actions
    DoRec : Inf Actions -> Actions
    End : Actions
```
This type declares that, at each stage, a `Channel` can accept one of the following operations:

- **DoListen** `c k`: check whether a message is waiting from a client `c`, then continue with actions `k`.
- **DoSend** `d a k`: send a message of type `a` to destination `d`, then continue with actions `k x`, where `x` (of type `a`) is the message sent. Note that this means a protocol can change according to the value transmitted!
- **DoRecv** `s a k`: receive a message of type `a` from source `s`, then continue with actions `k x`, where `x` (of type `a`) is the message received.
- **DoRec** `k`: recursively continue with actions `k`. Inf indicates that this is a **coinductive** argument.
- **End**: no more messages may be sent on this channel.

Aside: Idris supports **mixed inductive and coinductive** [10] definitions, such as `Actions`. Only total functions will be evaluated during type checking; to be considered total, a recursive call on a coinductive argument such as the argument to **DoRec** must be **guarded** by another constructor, with no non-constructor definitions in between. For our purposes, it suffices to know that any recursive protocol definition must be guarded by the **DoRec** constructor.

### 3.2. Operations

Listing 4 shows the operations available on channels. These include sending and receiving values, listening for an incoming message, connecting to a server process and closing a connection to a server process. Like `DoorCmd`, each action is restricted to running on channels in the appropriate state: we can only send on a channel in the **DoSend** state, receive from a channel in the **DoRecv** state, and so on.

The **Cmd** structure also supports some basic I/O (Print and GetLine) though in a complete implementation, we would parameterise over a flexible set of effects [5].

As well as the value it returns (the `Type*` argument), **Cmd** is parameterised over:

- A process of type `proc` (typically referred to as `me`). This ensures that all `Channel`s used have the correct local process.
- An input and output list of remote processes, of type `List proc`. In particular, note that `Connect`ing to a server channel extends this list of processes:

  ```
  Connect : (c : RChannel t p) ->
  Cmd me xs (t :: xs) (Channel me t p)
  ```

  Then, **Close** will remove a process from the list:

  ```
  Close : (c : Channel me t End) ->
  {auto prf : Elem t xs} ->
  Cmd me xs (dropElem xs prf) ()
  ```

  The proof argument, `Elem t xs`, ensures that the `Channel` being closed has previously been connected, with the **auto** keyword meaning that Idris will attempt
Listing 4: Operations on Channels

data Cmd : proc -> List proc -> List proc ->
    Type* -> Type* where
  Send : (val : a) -> Channel me t (DoSend t a k) ->
    Cmd me xs xs (Channel me t (k val))
 Recv : (c : Channel me t (DoRecv t a k)) ->
    Cmd me xs xs (Res a (∖v => Channel me t (k v)))
  Listen : (c : Channel me t (DoListen t k)) ->
    { auto prf : Elem t xs} ->
    Cmd me xs xs (Res Bool (∖ok =>
        if ok then Channel me t k
        else Channel me t (DoListen t k)))
  Connect : (c : RChannel t p) ->
    Cmd me xs (t :: xs) (Channel me t p)
  Close : (c : Channel me t End) ->
    {auto prf : Elem t xs} ->
    Cmd me xs (dropElem xs prf) ()
  Print : String -> Cmd me xs xs ()
  GetLine : Cmd me xs xs String

to construct the proof automatically at compile-time. It can be helpful to think of
arguments marked auto as side conditions on an operation, stating some information
which must be statically known.

Also note the types of Recv and Listen. Recv returns a Channel where the
continuation of the protocol is computed from the value received:

\[
\text{Res } a (\\backslash v \Rightarrow \text{Channel me } t \ (k \ v))
\]

Listen will return whether a message is waiting. If so, the Channel state can move
on to the next step, otherwise it remains in the DoListen state:

\[
\text{Res } \text{Bool} (\\backslash ok \Rightarrow
\begin{array}{l}
\text{if } ok \text{ then } \text{Channel me } t \ k \\
\text{else } \text{Channel me } t \ (\text{DoListen } t \ k)
\end{array})
\]

3.3. Protocol Descriptions

We now have channels parameterised by a valid list of actions, and commands which
act on those channels. However, we do not yet have any way of guaranteeing that
concurrent processes are running according to a corresponding communication pro-
tocol. To achieve this, we introduce a Protocol DSL, which describes an overall
communicating system between several processes each labelled by a proc. The DSL is shown in Listing 5.

Listing 5: Protocol DSL

```idris
data Protocol : List proc -> Type -> Type where
  Initiate : (c : proc) -> (s : proc) ->
    \auto prfc : Elem client xs ->
    \auto prfs : Elem server xs ->
    Protocol [c, s] () -> Protocol xs ()

  Send : (from : proc) -> (to : proc) -> (a : Type) ->
    \auto prf : SendOK a from to xs b ->
    Protocol xs b

  Rec : Inf (Protocol xs a) -> Protocol xs a
  Return : a -> Protocol xs a
  (>>=) : Protocol xs a -> (a -> Protocol xs b) ->
    Protocol xs b
```

This DSL supports two commands, `Initiate` and `Send`, as well as three control structures, `Rec`, `Return` and a `>>=` operator. The control structures support recursion and do notation; the commands are:

- **Initiate c s**: A client process `c` initiates a communication with a server process `s` by sending it a message. There are two side conditions: each of `c` and `s` must be in the list of processes in the overall protocol.

- **Send from to ty**: A process `from` sends a message of type `ty` to process `to`. Again, there is a side condition to ensure that it is valid to send the message. We omit details of this condition for brevity\(^2\).

Idris allows us to define syntactic sugar (to a limited extent). Here, we define protocol syntax which resembles security protocol notation [21]:

```
syntax [ from ] "== >" [ to ] "|" [ t ] = Send from to t
```

Recall the echo protocol example briefly described in Section 1.3

```
echo : Protocol ['C, 'S] ()
echo = do msg <- 'C == > 'S | String
       'S == > 'C | String
       'C == > 'S | Nat
```

A Protocol description can be translated into the corresponding Actions on a Channel for each participant in the protocol, using protoAs:

---
\(^2\)Essentially, a message is valid if both participants are in the protocol, but if there are more than two participants then future messages cannot depend on this message, since only two participants can know the message which was sent.
protoAs : (x : proc) -> Protocol xs () ->
{auto prf : Elem x xs} -> Actions

In the case of echo, we can calculate the Actions for the client \( 'C \) at the Idris REPL as follows:

\[
*echo > protoAs 'C echo
DoSend 'S String (\cmd =>
  DoRecv 'S (Literal cmd) (\cmd =>
    DoRecv 'S Nat (\x => End))) : Actions
\]

That is, a client for the echo protocol must send a String, then receive a String which is guaranteed to be identical to the String which was sent, then receive a Nat.

For a given protocol, we can calculate the protocol actions a server for that protocol will take, as follows:

\[
serverLoop : (c : proc) -> Protocol [c, s] () ->
  Protocol [c, s] ()
\]

\[
serverLoop c \{s\} proto
  = Initiate c s (do
    proto
    Rec (serverLoop c proto))
\]

A serverLoop, then, waits for a connection to be initiated by a client, runs the protocol, then recursively calls the serverLoop. The protocol for the echo server then becomes:

\[
*echo > protoAs 'S (serverLoop 'C echo)
DoListen 'C
  (DoRecv 'C String (\cmd =>
    DoSend 'C (Literal cmd) (\cmd =>
      DoSend 'C Nat (\x =>
        DoRec (serverLoop 'C echo))))))
\]

That is, it must wait for a connection from a client, then receive a String, send back an identical String, send back a Nat, then start again.

We are now in a position to define a language of Concurrent IO, supporting commands on channels, forking processes and starting server processes. Top level functions for this language are given in Listing 6. Note that we are presenting top level functions here, rather than the full definition of the language; the full language is available online\(^3\). In practice, many of the commands are direct calls to commands in the Cmd structure.

Note in particular, the types of fork and start_server. Each ensures that the process being spawned has a communication structure which corresponds to the Channel it uses, and the Channel being returned. To start a worker process:

\(^3\)http://github.com/edwinb/ConcIO
Listing 6: Concurrent IO language

```haskell
data CIO : proc -> List proc -> List proc -> Type* -> Type*

fork : (proto : Protocol [c,s] ()) ->
  (Channel s c (protoAs s proto) ->
    CIO s (c :: xs) xs () ->
    CIO c xs (s :: xs) (Channel c s (protoAs c proto))

start_server
  : (proto : Protocol [c,s] ()) ->
  (Channel s c (protoAs s (serverLoop c proto)) ->
    CIO s (c :: xs) (c :: xs) Void) ->
  CIO c xs xs (RChannel s (protoAs c proto))

send : (val : a) -> Channel me t (DoSend t a k) ->
  CIO me xs xs (Channel me t (k val))
recv : (c : Channel me t (DoRecv t a k)) ->
  CIO me xs xs (Res a (\v => Channel me t (k v)))
listen : (c : Channel me t (DoListen t k)) ->
  {auto prf : Elem t xs} ->
  CIO me xs xs (Res Bool (listenRes me t k))
connect : (c : RChannel t p) ->
  CIO me xs (t :: xs) (Channel me t p)
close : (c : Channel me t End) ->
  {auto prf : Elem t xs} ->
  CIO me xs (dropElem xs prf) ()

reset : Channel s t (DoRec act) -> Channel s t act
```

This works according to a protocol, `proto`. The spawned process takes a `Channel` working from the point of view of `s`, whereas the returned channel works from the point of view of `c`. To start a server process:

```haskell
start_server : (proto : Protocol [c,s] ()) ->
  (Channel s c (protoAs s (serverLoop c proto)) ->
    CIO s (c :: xs) (c :: xs) Void) ->
  CIO c xs xs (RChannel s (protoAs c proto))
```

This also works according to a protocol, but differs in that the server process loops. Note that by returning `Void` (the empty type), it is clear from the type that
the server process can never stop because it can never create an element of the empty type. The server returns an \texttt{RChannel}; we cannot transmit over an \texttt{RChannel} but we can use it to create a valid communication channel using \texttt{connect}:

\begin{verbatim}
connect : (c : RChannel t p) ->
        CIO me xs (t :: xs) (Channel me t p)
\end{verbatim}

To provide more convenient notation for types of concurrent programs with client and server processes, there are type synonyms \texttt{Conc}, \texttt{Server} and \texttt{Client}, for top level processes, server processes and client processes respectively. These are all specialised cases of \texttt{CIO}. A top level program is a \texttt{CIO} program which preserves the list of channels:

\begin{verbatim}
Conc : Type -> Type -> Type
Conc p r = {xs : _} -> CIO p xs xs r
\end{verbatim}

A server process receives a client channel as input, and returns \texttt{Void} meaning that it is expected to loop forever:

\begin{verbatim}
Server : (s, c : proc) -> Protocol [c, s] () -> Type*
Server s c p = {xs : _} ->
    Channel s c (protoAs s (serverLoop c p)) ->
    CIO s (c :: xs) (c :: xs) Void
\end{verbatim}

A client process receives a replicable server channel as input:

\begin{verbatim}
Client : (c, s : proc) -> Protocol [c, s] () -> Type*
Client c s p = {xs : _} ->
    RChannel s (protoAs c p) ->
    CIO c xs xs ()
\end{verbatim}

4. Client/Server Example

Listing 7 gives the protocol for a simple server which accepts three commands: \texttt{Mul}, which expects to receive two integers and sends back the product of those integers; \texttt{StrLen}, which expects to receive a \texttt{String} and sends back the length of the \texttt{String}, and \texttt{Uptime} which sends back the number of cycles the server has been running.

Note how the protocol can vary depending on the first message sent to the server. An implementation of the server, therefore, will need to check which command was sent and use the appropriate protocol for sending and receiving to and from the client. A sample implementation is given in Listing 8. One interesting feature of this server is that it maintains a \texttt{state}, the number of cycles it has been running, by recursively calling itself with an updated state. As such, we can use this pattern to maintain safe concurrent access to some program state.

One possible way to use such a concurrent server would be to treat it as a module, providing services concurrently for a number of worker threads in an application. We
Listing 7: Simple Utility Protocol

```haskell
data UtilCmd = Mul | StrLen | Uptime

utils = do cmd <- 'C == > 'S | UtilCmd
    case cmd of
        Mul => do 'C == > 'S | (Int, Int)
        'S == > 'C | Int
        StrLen => do 'C == > 'S | String
        'S == > 'C | Nat
        Uptime => 'S == > 'C | Int
```

Listing 8: Sample Server Implementation

```haskell
utils_server : Int -> Server 'S 'C utils
utils_server uptime chan
    = do (True @@ chan) <- listen chan
        | (False @@ chan) => utils_server uptime chan
        (cmd @@ chan) <- recv chan
        case cmd of
            Mul => do ((x, y) @@ chan) <- recv chan
                chan <- send (x * y) chan
                utils_server (uptime + 1) (reset chan)
            StrLen => do (str @@ chan) <- recv chan
                chan <- send (length str) chan
                utils_server (uptime + 1) (reset chan)
            Uptime => do chan <- send uptime chan
                        utils_server (uptime + 1) (reset chan)
```

might write functions such as `multiply`, given in Listing 9, to access features of the server via a replicable channel.

By implementing the client and server in an embedded DSL for managing channel state and protocol actions, we can be sure that each channel is accessed only by a process which is allowed to send and receive on that channel, and that protocol actions are synchronised between communicating threads. However, what if there is a compile-time error? For example, what if in the server implementation in Listing 8, we swap the `Uptime` and `StrLen` cases? In our library implementation, Idris reports:

```
When elaborating argument c to function Channels.recv:
Channel Error: Recv from 'C used when
Send to 'C required
```

Idris has been designed with the intention of implementing embedded DSLs, therefore it supports several features to help with building and using DSLs. Here, we
have used *error message reflection* [9], which allows us to rewrite error messages before reporting them to the user. In practice, error messages arising from DSL programs take a very similar form, so intercepting and rewriting messages can present very helpful diagnostics to the DSL programmer.

5. Related Work

The Idris virtual machine has message passing concurrency as a primitive, following Erlang style concurrency [17] and Scala style actors [13]. The underlying implementation in Idris is untyped; library support such as that described in this paper is required to give types to communicating programs. Our approach of describing communication protocols in the type system is closely related to Session types [14, 18]. Like our *Channel* type, Session Types provide static guarantees that communicating systems respect a given protocol. In a dependently typed language supporting uniqueness types, such as Idris, we have the flexibility to implement a Session Typed approach to communication by embedding in the type system. No type system extensions are required, and in particular we can exploit features of the type system to express dependencies in protocols, such as a protocol depending on the transmitted command in the server example in Section 4.

Our approach relies on uniqueness types [23] to enforce non-aliasing of resources. Uniqueness types are similar to linear types [8], except that variables must be referenced at most once, rather than exactly once. The interaction of dependent types and linear types, while possible [16], is difficult since we need to establish whether a use is in a type or a value, whereas a uniqueness type avoids this difficulty. Dependent uniqueness types are related to Typestate [1], in that they allow us to express a state in a type.

Previous work in Idris has used handlers of algebraic effects [20, 3, 15, 5] to describe side-effecting computations, and furthermore to describe resource access protocols [6]. However, when there are increasing numbers of communication channels, tracking each in a separate effect becomes unwieldy.
The protocol description language is inspired by the security protocol notation introduced for Kerberos [21]. Ultimately, we hope to be able to apply our approach to communicating programs over a network, and particularly to reason about secure communication protocols in the type system [11, 12].

6. Conclusion

This paper documents the first experiments in implementing typed communicating programs in Idris. The paper has shown how to build resource access protocols using dependent types and uniqueness types, and how to use this technique to build a library of communication primitives. Overall, we have followed a type-driven approach to implementing concurrent systems. We write the type of the system as a whole (as a Protocol), calculate resource protocols for each participant, and ensure that the final program corresponds to the protocol by type checking. We have seen that this technique allows us to implement some common concurrent patterns, specifically spawning worker and server processes.

There is much still to explore. Perhaps most importantly, we have assumed throughout that all communication succeeds. Within the Idris run-time system, this should be a reasonable assumption (as long as we do not run out of memory!) since we have expressed in the types that each process completes its part of the protocol. However, across a network, in a distributed concurrent system, we cannot make this assumption. By giving more expressive types to operations such as send and recv, we should be able to establish whether there are communication errors in transmission (including whether a connection remains open, and whether a value of the correct type has been trasmitted.) We can learn a lot from Erlang [2] about how to manage failure in distributed systems, and we hope to apply these techniques within our system.

Communication occurs in many contexts, not only in distributed concurrent systems but also in network application protocols (web servers, DNS, etc) and in secure communication protocols. Our approach of implementing communication primitives in a DSL means that we can apply a variety of interpretation functions to programs in the DSL. For example, one may communicate between internal Idris processes, another may communicate over a network. In this way, we hope to be able to apply our Protocol DSL to reason about secure network protocols as well as concurrent programs.

Finally, we have implemented some useful models of concurrency, in that we can spawn a worker process which communicates with a parent thread, and we can spawn a server process which accepts connections from multiple clients, but there are many models of concurrency and programming patterns. In future work, we intend to explore which programming patterns (e.g. Skeletons [7]) fit within our system.
References

Types. In: *Principles of Programming Languages (POPL)*. 2015.


**Affiliations**

**Edwin Brady**  
School of Computer Science, University of St Andrews

Received: ???  
Revised: ???  
Accepted: ???