Edwin Brady

TYPE-DRIVEN DEVELOPMENT OF CONCURRENT COMMUNICATING SYSTEMS

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 the 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

Citation Computer Science 18(3) 2017: 219–240

1. Introduction

Implementing communicating concurrent systems is hard, and reasoning about them is 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 the 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 [13]. 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, parameterized 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 the other processes.

By *type-driven*, I mean that the approach involves writing an explicit type describing the pattern of communication and verifying that the processes follow this pattern by *type-checking*. Communication channels are explicitly parameterized 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, the properties of a program can be expressed in a type and *verified* by type checking. Furthermore, in an experimental extension, Idris supports *uniqueness types*.

A value with a unique type has the property that is guaranteed to be *at most one* reference to that value at run-time. As such, a value with a unique type can be used to represent a reference to a resource in a specific state. Throughout this 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 those 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 the 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 the *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 a form of an "echo" protocol in which a client sends a server a string, and the server responds by echoing the string back, followed 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 parameterized on a list of participants, here labeled `'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. Type `Literal s` represents a literal string that is required to have value `s`, meaning here that server *must* echo the string back to the client. There are no restrictions on the natural number.

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

The server begins by `listen`ing on 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 that uses the channel returns a *new* channel, for example:

```
(msg @@ chan) <- recv chan
```

Listing 2. Echo Server Implementation.

```
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.

```
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 the 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
```

Finally, we can write a concurrent main program running from the point of view of a client (`'C`), which starts a server running according to the `echo` protocol, then repeatedly sends requests to the server via the `echo_client`:

```
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 a domain-specific language for concurrent programs. Before we deal with concurrency itself, however, we will need to establish how to manage the *resources* with the associated *state* and *resource access protocols*.

2. Resource management in Idris

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 state.
- OpenDoor, which is valid when CLOSED, and changes the state to OPEN.
- CloseDoor, which is valid when OPEN, and changes the state to CLOSED.

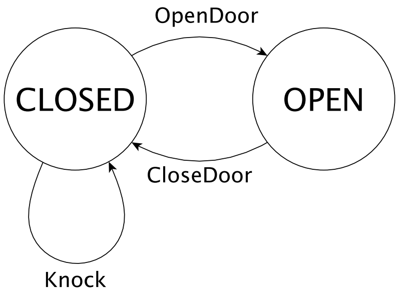

Figure 1. State machine representing a door.

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: parameterized handle

We will use a handle (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):

```
data DoorState = Opened | Closed
data DoorH : DoorState -> Type
```

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

```
data DoorCmd : Type -> Type where
     OpenDoor  : DoorH Closed -> DoorCmd (DoorH Opened)
     Knock     : DoorH Closed -> DoorCmd ()
     CloseDoor : DoorH Opened -> DoorCmd (DoorH Closed)
```

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, 14, 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:

```
data DoorLang : Type -> Type where
  Return : a -> DoorLang a
  Action : DoorCmd a -> DoorLang a
  (>>=) : DoorLang a -> (a -> DoorLang b) -> DoorLang b
```

Aside: Like Haskell, Idris supports do notation that desugars to >>=. 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 >>= operator.

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

```
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. 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 a `UniqueType`:

```
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 there is at most one live reference to `x` at run-time. 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*
     ...
```

First, the types are parameterized by `Type*` rather than `Type`; second, the `Knock` operation returns a handle rather than a value of the unit type `()`.

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 one time at most. 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 following 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 that 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 as well as the new handle, but it should also guarantee that the result and 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:

```
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`:

```
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 `OpenDoor` has succeeded or not:

```
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 `OpenDoor`, we must inspect the first (`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:

```
doorOK : DoorH Closed -> DoorLang ()
doorOK h = do h <- Action (Knock h)
              hres <- Action (OpenDoor h)
              case hres of
                   False @@ h => Return ()
```

```
                          True @@ h => do
                             Action (CloseDoor h)
                             Return ()
```

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

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

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

```
do pat <- val | <alternatives>
   p
```

...is desugared to...

```
do x <- val
   case x of
        pat => p
        <alternatives>
```

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: returning immediately, or running the protocol to completion:

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

3. Concurrent communication protocols

Idris supports concurrent programming by allowing processes to send messages to each other on a *channel*. 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 concurrent 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 synchronized.

3.1. Channels

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

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

This is a `UniqueType`, meaning that a channel in a particular state can be used one time at most. The local (`src`) and remote (`dest`) processes have type `proc`, which is a type variable, and essentially serves as a way of labeling the processes. The `Actions`, defined below, describe the current *state* of a `Channel`, which explains the sequence of messages that are still to be sent over that channel. When creating a server, we may also wish to have several client processes connecting to a single server. For this case, we allow *replicable* channels, declared as follows:

```
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 `Channel` and `RChannel` is parameterized by the remaining actions in a process; `Actions` is declared as follows:

```
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`. This is to allow a server to accept a new connection from a client.
- `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 *potentially infinite* argument.
- `End`: no more messages may be sent on this channel.

Aside: The `Inf` type, used in `DoRec`, means that the list of actions is potentially infinite; for example, a server is allowed to run indefinitely. In practice, this means that the Idris type checker will evaluate potentially infinite `Actions` lists at compile time. The full details are beyond the scope of this paper (and not necessary to understand how to use `Cmd` and `Actions`), but `Inf` arises from Idris's support for *mixed inductive and coinductive* [9] definitions. 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 parameterize over a flexible set of effects [5].

Listing 4. Operations on Channels.

```
data Cmd : proc -> List proc -> List proc ->
           Type* -> Type* where

  Connect : RChannel srv p ->
            Cmd me xs (srv :: xs) (Channel me srv p)
  Close : Channel me srv End ->
          {auto prf : Elem srv xs} ->
          Cmd me xs (dropElem xs prf) ()

  Listen : 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)))
  Send : (val : a) -> Channel me t (DoSend t a k) ->
         Cmd me xs xs (Channel me t (k val))
  Recv : Channel me t (DoRecv t a k) ->
         Cmd me xs xs (Res a (\v => Channel me t (k v)))

  Print : String -> Cmd me xs xs ()
  GetLine : Cmd me xs xs String
```

As well as the value it returns (the `Type*` argument), `Cmd` is parameterized 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`.

By looking at these parameters for each operation, we can see how that operation affects the state of a communication channel and the overall state of the communicating system:

- `Connect` connects to a server, and extends the list of remote processes:

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

  That is, before `Connect`ing to a process `srv`, the remote processes are `xs`; afterwards, the remote processes are extended with `srv` and become `srv :: xs`.
- `Close` disconnects from a process, and removes that process from the list:

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

  Proof argument (`Elem t xs`) ensures that the `Channel` being closed has previously been connected, with the `auto` keyword meaning that Idris will attempt 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 that must be *statically* known.
- `Listen` returns whether a message is waiting from a client process on a server's channel. If so, the `Channel` state can move on to the next step; otherwise, it remains in the `DoListen` state, as described in its return type:

  ```
  Res Bool (\ok => if ok then Channel me t k
                          else Channel me t (DoListen t k))
  ```

- `Send` sends a message on a channel, as long as that channel is in a state where the next action is to send a message:

  ```
  Send : (val : a) -> Channel me t (DoSend t a k) ->
         Cmd me xs xs (Channel me t (k val))
  ```

  It returns a new `Channel` where the continuation of the protocol is computed from the value sent.
- `Recv` receives a message on a channel:

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

  Like `Send`, it returns a new `Channel`, but where the continuation of the protocol is computed from the value received.

We now have channels parameterized by a valid list of actions as well as the commands that act on those channels. However, we do not yet have any way of guaranteeing that concurrent processes are running according to a corresponding communication protocol. We will achieve this by defining a language of protocol descriptions.

3.3. Protocol descriptions

Listing 5 introduces a `Protocol` DSL, which describes an overall communicating system between several processes (each labeled by a `proc`). A `Protocol` is parameterized by a list of processes involved in the protocol and the type of the value being transmitted by a protocol action. Parameterizing over this value means that later parts of a protocol can *depend* on earlier communications. For example, we could write a protocol in which we begin by sending a number that gives the number of messages still to come.

Listing 5. Protocol DSL.

```
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) -> (ty : Type) ->
         {auto prf : SendOK ty 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: `c` and `s` must each 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 the details of this condition for brevity[1].

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

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

[1]Essentially, a message is valid if both participants are in the protocol; however, if there are more than two participants, then future messages cannot depend on this message since only two participants can know the message that was sent.

Recall the echo protocol example briefly described in Section 1.3

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

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

```
protoAs : (x : proc) -> Protocol xs () ->
          {auto prf : Elem x xs} -> Actions
```

In the case of echo, we can calculate the Actions for 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 that is guaranteed to be identical to the String that was sent, then receive a Nat. For a given protocol, we can calculate the protocol actions that 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 waits for a connection to be initiated by a client, runs the protocol, then recursively calls the serverLoop. The s parameter to serverLoop is left *implicit* in the type so that, when we *use* serverLoop, we only need to give the client's identity. We can see this in the echo example, where the protocol server 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.

3.4. A DSL for communicating concurrent programs

We are now in a position to define a domain specific language for concurrent interactive programs, supporting commands on channels, forking processes, and starting server processes. This DSL is defined by a data type, `CIO`, that allows for the sequencing of commands as defined by `Cmd`.

The 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[2].

Listing 6. Concurrent IO language.

```
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
(>>=) : CIO me xs xs' a ->
        (a -> CIO me xs' xs'' b) -> CIO me xs xs'' b
```

In practice, many of the commands are direct calls to commands in the `Cmd` structure. By defining `(>>=)` for `CIO`, we can sequence operations with `do` notation.

[2]`http://github.com/edwinb/ConcIO`

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

```
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))
```

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:

```
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 `RChannel`; we cannot transmit over an `RChannel`, but we can use it to create a valid communication channel using `connect`:

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

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

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

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

```
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
```

A client process receives a replicable server channel as input:

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

The `reset` function resets a server to its starting state before making a recursive call to the server:

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

Consider the last step in the `echo` protocol, for example: `DoRec (serverLoop C echo)`. By calling `reset`, we reset the state of the channel to `serverLoop 'C echo`, which allows the server to continue serving requests. Listing 7 shows how we can use `CIO` to implement a server for the `echo` protocol.

Listing 7. An `echo` server in the Concurrent IO language.

```
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
         let chan = reset chan
         echo_server chan
```

Note that, at the end, we need to use `reset` before calling `echo_server` recursively.

4. Client/server example

Listing 8 gives the protocol for a simple server that accepts three commands: `Mul`, which expects to receive two integers and sends back the product of those integers; `StrLen`, which expects to receive a `String` and sends back the length of the `String`; and `Uptime`, which sends back the number of cycles the server has been running.

Listing 8. Simple Utility Protocol.

```
data UtilCmd = Mul | StrLen | Uptime
utils : Protocol ['C, 'S] ()
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
```

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 9. One interesting feature of this server is that it maintains a *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.

Listing 9. Sample Server Implementation.

```
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)
```

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 might write functions such as `mul`, given in Listing 10, to access features of the server via a replicable channel.

Listing 10. Sample Client Function.

```
mul : RChannel 'S (protoAs 'C utils) -> Int -> Int -> Conc 'C Int
mul s x y = do chan <- connect s
               chan <- send Mul chan
               chan <- send (x, y) chan
               (res @@ chan) <- recv chan
               close chan
               pure res
```

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 that is allowed to send and receive on that channel and that the protocol actions are synchronized between the communicating threads. However, what if there is a compile-time error? For example, what if in the server implementation in Listing 9, 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* [8], 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 [16] and Scala actors [12]. The underlying implementation in Idris is untyped; library support is required to give types to communicating programs. Our approach is closely related to Session types [13, 17]. Like our `Channel` type, Session Types provide static guarantees that communicating systems respect a given protocol. In a dependently typed language supporting uniqueness types, we have the flexibility to implement a Session Typed approach to communication by embedding in the type system. No 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 in Section 4.

Our approach relies on uniqueness types [23] to enforce the non-aliasing of resources. Uniqueness types are similar to linear types [24], except variables must be referenced *at most* once, rather than exactly once. The interaction of dependent and linear types, while possible [15], is difficult, since we need to establish whether a use is in a type or a value. Nevertheless, recent work by McBride [18] has successfully combined linear and dependent types in an elegant fashion, enabling reasoning about the usage of variables. Indeed, there is a recent experimental extension to Idris that implements McBride's linear type system, and it is likely that, in the longer term, linear dependent types will form a theoretically sound basis for the kind of concurrent system described in the present paper. Combining linear and dependent types leads to a system related to Typestate [1] in that they allow us to express a state in a type but with the expressivity of a full purely functional language at the type level.

Previous work in Idris has used handlers of algebraic effects [3, 5, 14, 20] 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 over a network, and particularly to reason about secure communication in the type system [10, 11].

6. Conclusion

This paper documents the first experiments in implementing typed communicating programs in Idris. We have seen how to build resource access protocols using dependent 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 program corresponds to each protocol by *type checking.* This technique allows us to implement some common concurrent patterns, specifically spawning worker and server processes.

There is still much to explore. Perhaps most importantly, we have assumed throughout that communication succeeds. This is fine within the Idris run-time system (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 `send` and `recv`, we should be able to establish whether there are errors in transmission (e.g., whether a channel remains open or a value of the correct type has been transmitted.) We can learn a lot from Erlang [2] about managing 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. For example, one may communicate between internal Idris processes, and 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 that communicates with a parent thread, and we can spawn a server process that accepts connections from multiple clients; however, 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.

Acknowledgements

Thank you to the anonymous reviewers who provided many helpful suggestions, and to the organizers and participants of Lambda Days for encouraging the writing of this paper. This work was kindly supported by SICSA (the Scottish Informatics and Computer Science Alliance) and EPSRC grant EP/N024222/1 (Type-driven Verification of Communicating Systems).

References

[1] Aldrich J., Sunshine J., Saini D., Sparks Z.: Typestate-oriented programming. In: *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pp. 1015–1012, 2009.

[2] Armstrong J.: *Making Reliable Distributed Systems in the Presence of Software Errors.* Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[3] Bauer A., Pretnar M.: Programming with Algebraic Effects and Handlers, *Journal of Logical and Algebraic Methods in Programming*, vol. 84(1), pp. 108–123, 2015.

[4] Brady E.: Idris, a general-purpose dependently typed programming language: Design and implementation, *Journal of Functional Programming*, vol. 23, pp. 552–593, 2013.

[5] Brady E.: Programming and Reasoning with Algebraic Effects and Dependent Types. In: *ICFP '13: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ACM, 2013.

[6] Brady E.: Resource-dependent Algebraic Effects. In: Hage J., McCarthy J. (eds.), *Trends in Functional Programming (TFP '14)*, *LNCS*, vol. 8843, Springer, 2014.

[7] Brown C., Hammond K., Danelutto M., Kilpatrick P.: A Language-independent Parallel Refactoring Framework. In: *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pp. 54–58. ACM, New York, 2012.

[8] Christiansen D.: *Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages*, Draft, 2014. `http://www.itu.dk/people/drc/drafts/error-reflection-submission.pdf`.

[9] Danielsson N.A.: Total Parser Combinators. In: *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pp. 285–296. ACM, New York, 2010.

[10] Fournet C., Bhargavan K., Gordon A.D.: Cryptographic verification by typing for a sample protocol implementation. In: Aldini A., Gorrieri R., *Foundations of Security Analysis and Design VI*, 2010.

[11] Gordon A., Jeffrey A.: Authenticity by Typing for Security Protocols, *Journal of Computer Security*, vol. 11(4), pp. 451–520, 2003.

[12] Haller P., Odersky M.: Scala Actors: Unifying thread-based and event-based programming, *Theoretical Computer Science*, special issue: Distributed Computing Techniques, vol. 410(2–3), pp. 202–220, 2009.

[13] Honda K., Yoshida N., Carbone M.: Multiparty asynchronous session types. In: *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pp. 273–284, 2008.

[14] Kammar O., Lindley S., Oury N.: Handlers in action. In: *Proceedings of the 18th International Conference on Functional Programming (ICFP '13)*, ACM, 2013.

[15] Krishnaswami N.R., Pradic P., Benton N.: Integrating Linear and Dependent Types. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 17–30, 2015.

[16] Larson J.: Erlang for Concurrent Programming, *Queue*, vol. 6(5), pp. 18–23, 2008.
[17] Lindley S., Morris J.G.: A semantics for propositions as sessions. In: *European Symposium on Programming (ESOP '15)*, 2015.
[18] McBride C.: I Got Plenty o Nuttin'. In: *A List of Successes that can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, LNCS, Springer, 2016.
[19] Peyton J.S.: Haskell 98 Language and Libraries – The Revised Report, 2002. `http://www.haskell.org/`.
[20] Plotkin G., Pretnar M.: Handlers of Algebraic Effects. In: *ESOP 09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pp. 80–94, 2009.
[21] Steiner J.G., Neuman C., Schiller J.I.: Kerberos: An Authentication Service for Open Network Systems. In: *USENIX*, pp. 191–202, 1988.
[22] The Idris Community: *The Idris Tutorial*, 2017. `http://docs.idris-lang.org/en/latest/tutorial/index.html`.
[23] de Vries E., Plasmeijer R., Abrahamson D.M.: Uniqueness Typing Simplified. In: Chitil O., Horváth Z., Zsók V. (eds.), *Implementation and Application of Functional Languages (IFL '07)*, *Lecture Notes in Computer Science*, vol. 5083, pp. 201–218, Springer, 2007.
[24] Wadler P.: Linear types can change the world! In: Broy M., Jones C. (eds.), *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pp. 347–359, North Holland, 1990.

Affiliations

Edwin Brady
School of Computer Science, University of St Andrews, ecb10@st-andrews.ac.uk

Received: 24.02.2015
Revised: 10.05.2017
Accepted: 16.05.2017