State Machines All The Way Down
An Architecture for Dependently Typed Applications

Edwin Brady

University of St Andrews, KY16 9SX, Scotland/UK,
ecb10@st-andrews.ac.uk

Abstract A useful pattern in dependently typed programming is to define a state transition system, for example the states and operations in a network protocol, as a parameterised monad. We index each operation by its input and output states, thus guaranteeing that operations satisfy pre- and post-conditions, by typechecking. However, what if we want to write a program using several systems at once? What if we want to define a high level state transition system, such as a network application protocol, in terms of lower level states, such as network sockets and mutable variables? In this paper, I present an architecture for dependently typed applications based on a hierarchy of state transition systems, implemented as a library called `states`. Using `states`, I show: how to implement a state transition system as a dependent type, with type level guarantees on its operations; how to account for operations which could fail; how to *combine* state transition systems into a larger system; and, how to implement larger systems as a hierarchy of state transition systems. As an example, I implement a simple high level network application protocol.

1 Introduction

Software relies on state, and many components rely on state machines. For example, they describe network transport protocols like TCP [27], and implement event-driven systems and regular expression matching. Furthermore, many fundamental resources like network sockets and files are, implicitly, managed by state machines, in that certain operations are only valid on resources in certain states, and those operations can change the states of the underlying resource. For example, it only makes sense to send a message on a connected network socket, and closing a socket changes its state from "open" to "closed". State machines can also encode important security properties. For example, in the software which implements an ATM, it's important that the ATM dispenses cash only when the machine is in a state where a card has been inserted and the PIN verified.

Despite this, state transitions are generally not checked by compilers. We routinely use type checkers to ensure that variables and arguments are used consistently, but statically checking that operations are performed only on resources in an appropriate state is not well supported by mainstream type systems.

In this paper, I show how we can represent state machines precisely in the dependently typed programming language Idris [5], inspired by the work of

Hancock and Setzer on describing interactive programming in dependent type theory with command and response trees [13], and by ongoing work on algebraic effects and handlers [26,4]. All of the code in this paper is available online[1].

1.1 Contributions

I build on previous work on algebraic effects in Idris [6,7]. In this earlier work, an *effect* is described by an algebraic data type which gives the operations supported by that effect, and which is parameterised by a *resource* type. Operations can change the types of those resources, meaning that we can implement and verify state transition systems using effects. However, there are shortcomings: the concrete resource type is defined by the *effect signature* rather than by its *implementation*; it is not possible to create *new* resources in a controlled way; and, it is not possible to implement *handlers* for effects in terms of other effects. In this paper, I address these shortcomings, making the following specific contributions:

- I present a library `states` which supports describing state machines and transitions in an algebraic data type, combining multiple state machines in a larger program, and creating and deleting resources as required (Section 3)
- I show how to describe existing stateful APIs in terms of state transition systems in `states` (Section 4)
- I show how `states` supports the implementation of high level state transition systems in terms of lower level systems, using a network application protocl as an example (Section 5)

Using `states`, we can encode the assumptions we make about state transitions in a type, and ensure by type checking that these assumptions always hold at run time. Moreover, by allowing state machines to be composed both horizontally (using multiple state machines at once within a function) and vertically (implementing a state machine in terms of other, lower level, state machines), `states` provides an *architecture* for larger scale dependently typed programs.

1.2 Motivating Example: Client-Server Communication

A motivation for this work is to be able to define communicating systems, where the type system verifies that each participant follows a clearly defined protocol. This is inspired by Session Types [16,17], in which the state of a session at any point represents the allowed interactions on a communication channel. Our overall goal is to represent session types using dependent types.

However, in order to use session types effectively in practice, we need to be able to use them in larger systems, interacting with other components. The `states` library provides a framework for implementing a system as a hierarchy of state machines, and as an example I will use it to implement a client-server system in which a client requests a random number within a specific bound from

[1] `https://github.com/edwinb/States`

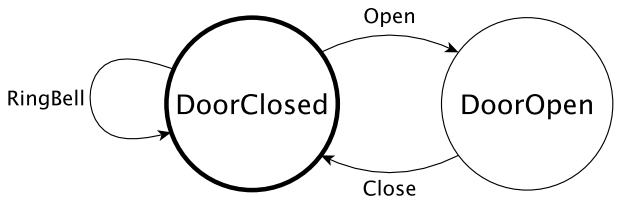
a server. As well as dealing with the session, therefore, the server will need a source of randomness. Furthermore, the high level session description will be implemented in terms of a lower level state machine to machine network socket state. To achieve this, we need to be able to:

- Use multiple states at once (for the protocol and the source of randomness), creating new state machines where necessary
- Implement a high level state machine, for the protocol description, in terms of a low level state machine, for the network sockets

For each state machine, I will use the type system to guarantee that operations satisfy their necessary *preconditions*, for example that we can only send a reply to a message after receiving a request. To begin, therefore, I'll describe how to represent a state machine in a type, include error handling (in Section 2) then go on to describe a general framework for multiple state machines (in Section 3).

2 Type-level State Machines

To demonstrate how to represent the state of a system in a type, let's consider how to represent the state of a door, with a door bell. The state of a door is either `DoorOpen` or `DoorClosed`. There are three operations: `Open` which opens a closed door; `Close` which closes an open door, and `RingBell` which rings the door bell when the door is closed. We can illustrate this system as follows:



By defining this state machine in a type, we can ensure that any sequence of operations which type checks is a valid sequence of operations on a door. For example, attempting to close a door which is already closed should be a type error. In this section, I'll describe how to define such a type, how to extend it to handle possible errors (such as the door failing to open), and discuss the general problem of writing larger applications with state transition systems.

2.1 The `DoorCmd` type: First attempt

Listing 1 shows how we can represent this system as a type. This is a parameterised monad [2], parameterised by the state of the door *before* and *after* each operation. The arguments to `DoorCmd` are: a `Type`, which represents the return type of the operation; two `DoorStates`, which represent the *input* state (the precondition) and the *output* state (the postcondition) of the operation.

Listing 1. The Door state transitions represented as a type

```
data DoorState = DoorOpen | DoorClosed

data DoorCmd : Type -> DoorState -> DoorState -> Type where
     Open     : DoorCmd () DoorClosed DoorOpen
     Close    : DoorCmd () DoorOpen   DoorClosed
     RingBell : DoorCmd () DoorClosed DoorClosed
     Pure : ty -> DoorCmd ty state state
     (>>=) : DoorCmd a state1 state2 ->
             (a -> DoorCmd b state2 state3) ->
             DoorCmd b state1 state3
```

Each type describes how the operation affects the state; an operation and its type give a Hoare Triple [15]. We can use `do` notation, since `(>>=)` is desugared as in Haskell, with its type explaining how sequencing changes the door's state. Using `Pure` we can return a pure value. If a function using `DoorCmd` to sequence operations on a door typechecks, we can be sure that it's a valid sequence of operations. For example:

```
doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = do RingBell; Open; Close
```

In this example, we control the state transitions: when we ask to `Open` the door, it does open. Realistically, though, operations might fail. If the door jams, for example, it's still in the `DoorClosed` state even after we run `Open`. So not only do we need to allow operations to describe how they change state, we need to explain how *external* factors can change state.

2.2 Handling errors: Refining the `DoorCmd` type

Instead of using a specific output state for an operation, it can depend on the *result* of an operation. We can refine the type of `DoorCmd` to the following:

```
DoorCmd : (ty : Type) -> DoorState -> (ty -> DoorState) -> Type
```

As before, there are three arguments to `DoorCmd`:

- A `Type`, named `ty`, which represents the return type of the operation
- An *input* `DoorState`, which represents the precondition on the operation
- A function to *compute* the output state from the *result* of the operation

Then, we can express that `Open` might fail in its type:

```
data DoorResult = Jammed | OK

Open : DoorCmd DoorResult DoorClosed
          (\res => case res of Jammed => DoorClosed
                               OK => DoorOpen)
```

By refining the type in this way, we can express that the result of an operation can depend on some *external* effect, such as whether the door jams. The only way we can *know* whether a door opened successfully, and therefore the state it's in, is if we *check* the result of `Open`:

```
doorProg : DoorCmd Bool DoorClosed (const DoorClosed)
doorProg = do RingBell
              OK <- Open | Jammed => Pure False
              Close
              Pure True
```

The `| Jammed` notation is a notational convenience, introduced with an earlier version of the Idris effects library [7]. Idris allows *pattern matching* bindings in `do` blocks, like Haskell, but additionally allows *alternatives* to be given inline. This is equivalent to a `case` block which inspects the result of `Open`.

If we don't check the result of `Open`, as in the following (incorrect) implementation, an attempt to `Close` the door will not type check:

```
doorProg : DoorCmd () DoorClosed (const DoorClosed)
doorProg = do RingBell; Open; Close
```

So far, we've only *described* operations on the door in a type. To *execute* these operations, need we define a function to *interpret* the operations in some context. For example, we could define a function `run` to simulate the operations by printing the results of the actions in the `IO` context. To simulate the possibility of the door jamming, we have a counter `jam_time`. If the counter is at zero, opening the door will fail:

```
run : (jam_time : Nat) -> DoorCmd t in_state out_fn -> IO t
```

This is only a simulation, but separating the *description* and the *implementation* in this way means that we can implement the operations in different ways for different contexts. For example, as well as this simple simulation, we could have an implementation which controls a motor for an automatic door, or which displays a graphical representation on screen.

2.3 Limitations: Compositionality and Scalability

The advantage of representing `DoorCmd` operations in a parameterised monad is that we can be sure by type checking that operations always satisfy preconditions, and that protocols are run to completion. The type of `doorProg`, for example, states that the door must be closed on exit. While `DoorCmd` is mostly an abstract example, several real world systems work similarly: files, connections to a database, graphics contexts, network sockets, and many others. In a complete application, it's likely that we'll want to use several such state transition systems at once.

For example, perhaps we'd like to include some mutable state in `doorProg`, and use it to count the number of times the door opened successfully. We could add `Get` and `Put` operations to `DoorCmd`, and extend `run` accordingly, but we don't want to have to do this every time we add a new collection of operations, for

every program. In the rest of this paper, therefore, I'll present a library, `states`, which allows us to:

- Define state transition systems like `DoorCmd` *independently* of each other
- Define different *implementations* for a state transition system, in the same way that we can define different implementations of `run`
- Associate implementations with *resources* which store the underlying state of a system
- Combine multiple state transition systems in a larger system
- Implement *hierarchies* of state transition systems, where high level protocols are implemented in terms of lower level systems

Following Landin [22], `states` provides a way of "expressing things in terms of other things", allowing a programmer to extend the "basic set of given things". It provides a general framework for creating, using and combining state machines.

3 `states`: A Library for Composing State Machines

The `states` library aims to capture the common structure of computation with state machines, as exemplified by `DoorCmd` and `run` in Section 2. In this section, I introduce `states`, and show how to use it to reproduce and extend the `doorProg` example from Section 2.2. I also show how we can define mutable variables using `states`, and combine multiple states in a larger program.

3.1 Defining State Machines

Listing 2 defines a record type `SM` for representing state machines. A state machine record is parameterised over the state it represents, and consists of:

- An *initial* state
- A predicate which defines a set of *final* states
- A set of operations which define the state transitions
- a set of operations which define how to create new state machines from existing states (I'll defer discussion of this until Section 4)

Note in particular how `SM_sig`, the type which defines the operations a state machine can perform, corresponds to the structure of the `DoorCmd` type:

```
DoorCmd : (ty : Type) -> DoorState -> (ty -> DoorState) -> Type
```

And, indeed, we can define a data type `DoorOp` which describes the operations we can perform on a door as an instance of `SM_sig`:

```
data DoorOp : SM_sig DoorState where
     Open     : DoorOp DoorResult DoorClosed
                       (\res => case res of
                                     OK => DoorOpen
                                     Jammed => DoorClosed)
     Close    : DoorOp () DoorOpen (const DoorClosed)
     RingBell : DoorOp () DoorClosed (const DoorClosed)
```

Listing 2. Defining a state machine

```
SM_sig : Type -> Type
SM_sig stateType
    = (ty : Type) -> stateType -> (ty -> stateType) -> Type

record SM stateType where
  constructor MkSM
  init       : stateType
  final      : stateType -> Type
  operations : SM_sig stateType
  creators   : SM_sig stateType
```

We instantiate `stateType` with `DoorState`, to represent the current state of the door (`DoorOpen` or `DoorClosed`) as the precondition and postcondition of each operation. Other than the absence of `Pure` and `(>>=)`, this is the same as our earlier definition of `DoorCmd`.

A program using `states` can create and delete state machines. When created, a state machine is in its initial state (here, we'll use `DoorClosed`), and it can only be deleted when it is in a final (or *accepting*) state. We define final states using a predicate. Here, only `DoorClosed` is a valid final state:

```
data DoorFinal : DoorState -> Type where
     ClosedFinal : DoorFinal DoorClosed
```

Using these, we can define `Door` as a state machine record `SM`:

```
Door : SM DoorState
Door = MkSM DoorClosed DoorFinal DoorOp None
```

That is, its initial state is `DoorClosed`, its final state must satisfy `DoorFinal`, and its operations are defined by `DoorOp`. The final argument, `None`, defines an empty set of operations:

```
None : SM_sig stateType
```

This final argument is used for state transition systems where an operation can create a new instance of the system. We'll see an example of this in Section 4 when we create an open network connection from a socket which is listening for connections. Until then (and in most realistic use cases) we'll always use `None` for the `creators` field of `SM`. We can implement `doorProg` as follows:

```
doorProg : (door : State Door) ->
       SMs m () [] [Trans door DoorClosed (const DoorClosed)]
doorProg door = do on door RingBell
                   OK <- on door Open | Jammed => pure ()
                   on door Close
```

I will give the precise type for `SMs` shortly, in Section 3.4. For now, it suffices to know that, in order, the arguments are:

– The underlying *context* (of type `Type -> Type`) in which the program will be executed. Typically, this will be a monad. Here, `m` indicates that we can run the program in any context.
– The return type of the operations, here `()`.
– The state machines which the function can *create*. Here, this is an empty list `[]` so the funtion can't create any new machines.
– A list of state transitions. Here, there's a machine `door` which starts and ends in the `DoorClosed` state.

When a machine stays in the same state, like `door`, we can use `Stable` instead of `Trans` in the list of state transitions:

```
doorProg : (door : State Door) ->
           SMs m () [] [Stable door DoorClosed]
```

The `on` function runs an operation, either `RingBell`, `Open` or `Close` on a given state machine, provided that the machine is initialised and is in an appropriate state. Both `pure` and `(>>=)` are defined for `SMs`, so we can sequence operations using `do`-notation. We can also create new state machines using `new` and remove them using `delete`:

```
doorProg : SMs m () [Door] []
doorProg = do door <- new Door
              on door RingBell
              OK <- on door Open | Jammed => delete door
              on door Close
              delete door
```

The type expresses that `doorProg` is allowed to create new `Door` state machines, and must begin and end with *no* state machines. In other words, `doorProg` must delete any machine it creates, and a machine must be in a final state before it can be deleted. Note in particular that we must delete `door` in both the cases where the door is `Jammed` and when the door opens successfully.

3.2 Defining Mutable State

At the end of Section 2, I noted that adding a mutable variable to `DoorCmd` would involve modifying the data type, adding operations for `Get` and `Put`. In `states`, we can define mutable state using `SM`:

```
data VarOp : SM_sig Type where
     Get : VarOp a a (const a)
     Put : b -> VarOp () a (const b)

Var : SM Type
Var = MkSM () (\x => ()) VarOp None
```

In the `Var` state machine, the state is the *type* of the value currently stored in the mutable variable. We can change the type using `Put`. The initial state is the empty tuple, and *any* state is a valid final state. Listing 3 gives an example

of how we can use mutable state to add labels to a binary tree, labelling each node in order of depth-first, left to right traversal. In this listing, we use `doTag` to initialise a mutable variable for `tag`.

Listing 3. Using a mutable state to tag nodes in a tree

```
data Tree a = Empty | Node (Tree a) a (Tree a)

tag : (num : State Var) ->
      Tree a -> SMs m (Tree (Nat, a)) [] [Stable num Nat]
tag num Empty = pure Empty
tag num (Node left val right)
      = do left' <- tag num left
           t <- on num Get
           on num (Put (t + 1))
           right' <- tag num right
           pure (Node left' (t, val) right')

doTag : Tree a -> SMs m (Tree (Nat, a)) [Var] []
doTag t = do num <- new Var
             on num (Put 0)
             t' <- call (tag num t)
             delete num
             pure t'
```

The function `call`, used by `doTag`, invokes a function with a *smaller* set of states that it can create, or a smaller set of state machines in use. As with the rest of the `states` API, we'll see its precise type in Section 3.4. Here, we need use `call` because `tag` has a smaller set of states that it can create than `doTag`.

We can also combine multiple state machines in one function. For example, we can have a state machine for the door, and a mutable variable to count the number of times the door has been opened successfully:

```
doorProgCount : (door : State Door) -> (count : State Var) ->
       SMs m () [] [Stable door DoorClosed, Stable count Nat]
doorProgCount door count
    = do on door RingBell
         OK <- on door Open | Jammed => pure ()
         n <- on count Get
         on count (Put (n + 1))
         on door Close
```

3.3 Executing State Machines

So far, we've used `SMs` to write programs which *describe* sequences of state transitions, but we haven't seen how to *execute* those programs. The first argument

to `SMs` is the context in which the state transitions can be executed. For example, the type of `doorProg` says that we can run it in *any* context:

```
doorProg : SMs m () [Door] []
```

However, in order to run it, we also need to explain *how* each operation is executed. We can achieve this by defining an implementation for the `Execute` interface, shown in Listing 4. An interface in Idris is like a type class an Haskell, but can be parameterised over *anything*, rather than merely types, and there can be multiple implementations, with names.

Listing 4. The `Execute` interface, which explains how to execute operations on a state machine

```
interface Execute (sm : SM state) (m : Type -> Type) where
    resource : state -> Type
    initialise : resource (init sm)

    exec : (res : resource in_state) ->
           (ops : operations sm ty in_state out_fn) ->
           (k : (x : ty) -> resource (out_fn x) -> m a) -> m a

    create : (res : resource in_state) ->
             (ops : creators sm ty in_state out_fn) ->
             (k : (x : ty) -> resource (out_fn x) -> m a) -> m a
```

An implementation of `Execute sm m` explains how to implement the state machine `sm` in a specific computation context `m`. The implementation is associated with a *resource*, which is a concrete, run-time representation of the state. The methods of `Execute` are:

- `resource`, which calculates the type of the underlying resource
- `initialise`, which gives the initial value of a resource in the initial state
- `exec`, which takes the current resource and an operation, and a continuation which takes a return value and an updated resource
- `create`, which acts like `exec` but for `creators`

We can implement `Execute` for `Var`, in *any* context `m`, as follows:

```
Execute Var m where
    resource x = x
    initialise = ()

    exec res Get     k = k res res
    exec res (Put x) k = k () x

    create res op k = pass op
```

In this case, the type of the resource is given directly by the state of the `Var`, initialised to `()`. To define `exec`, in the case of `Get`, we pass the resource value to the continuation both as the return value and the updated resource. In the case of `Put`, we pass the argument as the updated resource. There are no `creators`; `pass` is a function which eliminates the empty type.

Once we've defined `Execute` for all the state machines in a context, we can use either `run` or `runPure` to run the state transitions:

```
run : (Applicative m, ExecList m ops) =>
      SMProg m a ops [] (const []) -> m a
runPure : ExecList id ops =>
          SMProg id a ops [] (const []) -> a
```

Here, `ExecList m ops` is an interface which extends `Execute` to a `PList` of state machines in the context `m`. We can try this with the `doTag` function we defined earlier. Given a tree...

```
testTree : Tree String
testTree = Node (Node Empty "One" (Node Empty "Two" Empty))
                "Three" (Node Empty "Four" Empty)
```

...the result of `runPure (doTag testTree)` is:

```
Node (Node Empty (0, "One") (Node Empty (1, "Two") Empty))
     (2, "Three")
     (Node Empty (3, "Four") Empty) : Tree (Nat, String)
```

Sometimes, we might want to implement the operations in a state machine in terms of a different state machine, rather than by directly implementing it for some context `m`. We can achieve this using the `Transform` interface:

```
interface Transform (sm : SM state) (sms : PList SM)
                    (ops : PList SM)
                    (m : Type -> Type) | sm, m where
```

The notation `| sm, m` indicates that implementations of `Transform` will be determined by the arguments `sm` and `m` along; this is a simplified version of functional dependencies. An implementation of `Transform sm sms ops m` explains how to implement `sm` in a context `m` in terms of other state machines, `sms`, possibly creating new states `ops` in the process. I will show an example of this in Section 5, when I implement a higher level network protocol in terms of network sockets, and using mutable state.

3.4 The `states` API

Having seen an example of `states` in action, we can now look more closely at the precise types of `SMs`, `new`, `delete` and `on`. The data type underlying all programs in `SMs` is `SMProg`:

```
data SMProg : (m : Type -> Type) -> (ty : Type) ->
              (ops : PList SM) -> (in_states : Context ts) ->
              (out_fn : ty -> Context us) -> Type
```

As with SMs, the first three arguments to SMProg are the underlying context m, the type to be returned ty, and a list of new state machines which a program can create ops. The other arguments are in_states, which gives a list of states on input (the preconditions) and out_fn, which gives a list of states on output computed from the return value. This follows the pattern with DoorCmd from Section 2, but with a list.

Rather than an ordinary list type, the type we use to represent lists is a list of *parameterised* types, PList:

```
data PList : (Type -> Type) -> Type where
     Nil : PList p
     (::) : p state -> PList p -> PList p
```

Each entry in a PList is a specific instance of the parameterised type. A PList SM, therefore, is a list of state machine structures, but without any concrete instantiations of that machine. A Context, on the other hand, stores details of machines which *have* been instantiated, include their current resource type:

```
data Resource : SM state -> Type

data Context : PList SM -> Type where
     Nil : Context []
     (::) : Resource t -> Context ts -> Context (t :: ts)
```

A Resource is a pair of a *label* and a state:

```
data State : SM state -> Type

(:::) : {sm : SM state} ->
        (label : State sm) -> (p : state) -> Resource sm
```

For example, door ::: DoorClosed is a resource with the label door, currently in the state DoorClosed. The label identifies a state machine.

Rather than writing the type of a program with complete contexts of input states and a list of output states, it's convenient to define types as a list of actions on state machines. These actions can be:

- Stable label state, where a machine label begins and ends in state
- Trans label in_state out_fn, where a machine label begins in in_state and ends in a state computed from the result by out_fn
- Add label state, where a program *creates* a new machine label which ends in state
- Remove label state, where a program *deletes* a machine label which begins in state

These are captured in the following data type:

```
data Action : Type -> Type where
     Stable : State sm -> st -> Action ty
     Trans : State sm -> st -> (ty -> st) -> Action ty
     Add : State sm -> st -> Action ty
     Remove : State sm -> st -> Action ty
```

Using a list of `Action`, we can define `SMs`, which we used earlier for our examples with the `Door` and `Var` state machines. This translates a list of actions into an instance of `SMProg` with the corresponding input and output states:

```
SMs : (m : Type -> Type) -> (ty : Type) ->
      (ops : PList SM) -> List (Action ty) -> Type
```

We can sequence operations using `do` notation by defining `(>>=)`, and return pure values using `pure`. It's also convenient to be able to `lift` operations in the underlying context, provided that it implements the `Monad` interface:

```
pure : (x : val) -> SMProg m val ops (out_fn x) out_fn
(>>=) : SMProg m a ops st1 st2_fn ->
        ((x : a) -> SMProg m b ops (st2_fn x) st3_fn) ->
        SMProg m b ops st1 st3_fn

lift : Monad m => m t -> SMProg m t ops ctxt (const ctxt)
```

Using `lift`, we can invoke operations in the underlying context, for example `IO` operations like `putStrLn`. We can therefore consider `SMProg` to be a monad transformer. By defining an appropriate interface, we can then restrict the operations which a particular program can run. For example, `states` defines an interface `ConsoleIO`, and we can write a type of this form:

```
consoleStates : ConsoleIO io =>
        (count : State Var) -> SMs io [] [Stable count Nat]
```

We can create a new state machine, provided that it is in the list `ops`. To check this, the predicate `PElem` shows that an instance of a parameterised type appears in a `PList`.

```
data PElem : p state -> PList p -> Type where
     Here  : PElem a (a :: as)
     There : PElem a as -> PElem a (b :: as)
```

Therefore, `PElem sm ops` means that a machine `sm` appears in `ops`. Using this, we can give a type for `new`:

```
new : (sm : SM state) ->
      {auto prf : PElem sm ops} ->
      SMProg m (State sm) ops ctxt
               (\lbl => (lbl ::: init sm) :: ctxt)
```

This returns a new label, of type `State sm`, and add a resource in its initial state to the context. The `auto` flag on the implicit argument `prf` means that Idris will construct a proof that the machine is available automatically.

We can run an operation on a state machine, provided that the machine is in a state which satisfies the precondition on the operation:

```
data InState : (sm : SM state) -> State r ->
               state -> Context ts -> Type where
  Here : InState sm lbl st (MkRes lbl sm st :: rs)
  There : InState sm lbl st rs -> InState sm lbl st (r :: rs)
```

A value of type `InState sm lbl st ctxt` is a proof that a machine `sm`, with label `lbl`, has state `st` in a context `ctxt`. Using this, we can state precisely when it is valid to run an operation on a state machine:

```
on : (lbl : State sm) ->
     {auto prf : InState in_state sm lbl ctxt} ->
     (op : operations sm t in_state out_fn) ->
     SMProg m t ops ctxt
          (\res => updateCtxt ctxt prf (out_fn res))
```

The return type of `on` is given by the operation, `op`, and the effect on the list of states is to update the state in the context using the output function given by the operation. The function `updateCtxt` updates a state in a context, given a proof that the relevant input state exists in the context.

We can `delete` a state machine if its current state, checked with `InState`, satisfies the predicate which defines the final state of that machine:

```
delete : (lbl : State iface) ->
         {auto prf : InState st sm lbl ctxt} ->
         {auto finalok : final sm st} ->
         SMs m () ops ctxt (const (drop ctxt prf))
```

The function `drop` removes a state from a context, given a proof that the state exists in the context.

Finally, if we want to call a function with a smaller set of existing states, or a smaller set of states which we can create, we need to provide a proof that the function we call uses a subset of those states. We can do this using `call`:

```
data SubCtxt : Context ts -> Context us -> Type
data SubList : PList a -> PList a -> Type

call : {auto op_prf : SubList ops' ops} ->
       {auto ctxt_prf : SubCtxt ys xs} ->
       SMs m t ops' ys ys' ->
       SMs m t ops xs
         (\result => updateWith (ys' result) xs ctxt_prf)
```

The predicates `SubCtxt` and `SubList` state that the first list is a subset (or equal to) the second, allowing for reordering, and `updateWith` rebuilds the context given the state updates from the function being called. Again, Idris constructs the proofs automatically, if possible, by proof search.

3.5 Implementation note

The implementation consists of a data type, `SMProg`, along with an interpreter `run`. Each of the functions `new`, `on`, `delete` and `call`, and the operations to deal with sequencing and monadic lifting, are, internally, constructors of `SMProg`. Although the details are intricate in places, the implementation follows a standard method of implementing a well-typed interpreter [3], and following the previous implementations of the Idris `effects` library [7] to deal with managing an environment of concrete values representing each state.

4 Network Socket Programming with `states`

The Berkeley Sockets API[2] allows communication between two processes, where a socket is the endpoint of a communication. In Idris, `Socket` is an abstract data type which represents a socket, over which we can send and receive messages. Briefly, to implement a network client, we need to: create a socket, connect it to a server, then send and receive messages over that socket. To implement a server, we need to: create a socket; bind it to a port; listen for incoming connections; accept a connection, then send and receive messages over the socket.

The operations for connecting a socket, binding it, listening for conenctions, and so on, are only valid in certain states, and in order to implement a client or server we need to execute the operations in the correct order. Furthermore, most of these operations could fail, resulting in an invalid socket. Using `states`, we can give types to the sockets API to represent the behaviour of each operation precisely. We define a state machine which can be in one of the following states:

- `Closed`, meaning that there is no `Socket`
- `Ready`, meaning that the `Socket` has been created
- `Bound`, meaning that the `Socket` has been bound to an address
- `Connected`, meaning that the `Socket` is connected to a server
- `Listening`, meaning that the `Socket` is listening for incoming connections
- `Open` either as a `Client` or `Server`, meaning that we can send and receive messages on the `Socket`

It's only in the final state, `Open`, when we can send messages. In this section, I'll show how to describe the sockets API using `states` and implement a minimal client/server system, where the client sends a *bound* to a server, which sends back a random number within that bound.

4.1 Operations on Sockets

Figure 1 shows how operations on a socket state machine affect the state.

We distinguish connections which are open as clients or servers, and define two predicates: `CloseOK` defines in which states it is valid to `Close` a socket, and `NetFinal` defines the final state of a socket, when it can be deleted. For simplicity, we don't distinguish between TCP and UDP sockets; we'll assume TCP throughout. We can represent these states using the following data types:

```
data Role = Client | Server
data SocketState = Closed | Ready | Bound | Connected
                 | Listening | Open Role
data NetFinal : SocketState -> Type where
     ClosedFinal : NetFinal Closed
```

[2] See for example `http://man7.org/linux/man-pages/man2/socket.2.html`

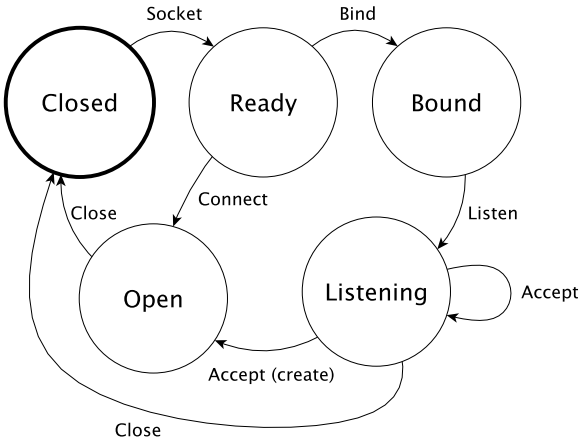


Figure 1. A state transition diagram which shows the states and operations on sockets

The machine is initialised in the `Closed` state, and can only be deleted in the `Closed` state. Note that there are *two* result states on the `Accept` operation, which accepts a connection from a client. This is because `Accept` creates a *new* socket, on which we can communicate with the client, while still allowing the current socket to receive connections from new clients.

Having defined socket states, we can define the operations on a socket state as a data type. Listing 5 gives the definition of `NetOp`, which represents the state transitions from Figure 1. By convention, we use `Either` to indicate the possibility of error, even if there is no further detail about the error or any further return result. We use `CloseOK` as a predicate to state when it is valid to `Close` a socket, since `Close` is a valid state transition from two states:

```
data CloseOK : SocketState -> Type where
     CloseOpen : CloseOK (Open role)
     CloseListening : CloseOK Listening
```

Also note that `Accept` is missing from the definition of `NetOp`, so at the moment there is no way to reach the `Open Server` state. This is a case where we need the `creators` field of `SM`. Not only is `Accept` only valid in a specific state, it also *creates* a new socket. Therefore, we define it separately, as in Listing 6.

By defining an implementation of `Execute` for `Net` using the low level Idris bindings for `Socket` operations in `IO`, we can execute programs which use `Net`:

```
Execute Net IO where
    resource Closed = ()
    resource Ready = Socket
    resource Bound = Socket
    resource Listening = Socket
    resource (Open x) = Socket
```

Listing 5. Defining operations on a state machine for socket communication

```
data NetOp : SM_sig SocketState where
  Socket : SocketType ->
           NetOp (Either SocketError ()) Closed
                 (either (const Closed) (const Ready))
  Bind : (addr : Maybe SocketAddress) -> (port : Port) ->
         NetOp (Either () ()) Ready
               (either (const Closed) (const Bound))
  Listen : NetOp (Either () ()) Bound
                 (either (const Closed) (const Listening))
  Connect : SocketAddress -> Port ->
            NetOp (Either () ()) Ready
                  (either (const Closed) (const (Open Client)))
  Close : {auto prf : CloseOK st} -> NetOp () st (const Closed)
  Send : String -> NetOp (Either () ()) (Open x)
                         (either (const Closed) (const (Open x)))
  Recv : NetOp (Either () String) (Open x)
               (either (const Closed) (const (Open x)))
```

Listing 6. Adding an `Accept` operation to create a new state machine

```
data NetCreate : SM_sig SocketState where
  Accept : NetCreate (Either SocketError SocketAddress)
                 Listening
                 (either (const Closed) (const (Open Server)))

Net : SM SocketState
Net = MkSM Closed NetFinal NetOp NetCreate
```

```
initialise = ()

exec = ...
```

I omit the details of `exec`; it is a direct translation to the low level socket API calls. As long as we've created a socket, the resource is represented as a `Socket`, otherwise it's the empty tuple `()`.

4.2 Implementing a Client

Using `Net`, we can implement the client part of the random number client/server system as in Listing 7. In this client, note that on each network operation we need to check whether the result was successful. If not, the socket reverts to the `Closed` state, we display an error message, then exit.

Listing 7. Defining a client of a random number server using `Net`

```
client_main : ConsoleIO io => (socket : State Net) ->
              SMs io () [] [Stable socket Closed]
client_main socket = do
   putStr "Bound: "
   x <- getStr
   Right ok <- on socket (Socket Stream)
      | Left err => putStrLn "Error on socket creation"
   Right ok <- on socket (Connect (Hostname "localhost") 9442)
      | Left err => putStrLn "Error on connect"
   Right ok <- on socket (Send x)
      | Left err => putStrLn "Send failed"
   Right reply <- on socket Recv
      | Left err => putStrLn "Error on recv"
   putStrLn reply
   on socket Close
```

By using dependent types in our description of the socket API we can also use *holes* to help us to develop the client. For example, after opening the socket, we can leave a hole `?client_rest` to stand for the rest of the program:

```
client_main socket
  = do putStr "Bound: "
       x <- getStr
       Right ok <- on socket (Socket Stream)
          | Left err => putStrLn "Error on socket creation"
       ?client_rest
```

Then, if we check the type of `client_rest`, we see that the current state of the socket (`Ready`) and the necessary ending state of the socket (`Closed`):

```
client_rest : SMProg io () [] [socket ::: Ready]
                   (\result => [socket ::: Closed])
```

4.3 Implementing a Server

Listing 8 gives an outline of a server, focussing on the creation of the connection, and using a mutable variable as a seed for a random number generator. Again, any error results in the socket being closed.

The `newFrom` function is similar to `on`, except that it returns the label for a *new* resource in addition to the result of the operation. Here, we use `newFrom socket Accept` to create a new socket in the `Open Server` state from a `Listening` socket.

```
newFrom : {auto prf : InState sm lbl in_state ctxt} ->
   (lbl : State sm) -> (op : creators sm t in_state out_fn) ->
   SMProg m (t, State sm) ops ctxt
       (\ (res, lbl) => (lbl ::: out_fn res) :: ctxt)
```

Listing 8. Defining a random number server using `Net`

```
rndServer : ConsoleIO io =>
            (socket : State Net) -> (seed : State Var) ->
            SMs io () [] [Trans socket Listening (const Closed),
                          Stable seed Integer]
rndServer socket seed = do
    (Right addr, conn) <- newFrom socket Accept
          | (Left err, conn) => do delete conn; on socket Close

    Right msg <- on conn Recv
          | Left err => do delete conn; on socket Close
    {- ... -}
    on conn Close
    delete conn
    rndServer socket seed
```

5 High Level Protocols: A Random Number Server

In Section 1.2, I described the general form of a client-server system which receives a message, then replies. The random number server is an instance of this. Instead of defining it using the sockets API directly, it is cleaner to develop a high level representation of the protocol, again as a state machine, then implement it in terms of the low level API. In this section, I show how to describe the server as a state machine itself, then implement it in terms of `Net` and `Var`.

I will define a state machine `RandServer` which includes the high level operations necessary to define the server, including the message passing and random number generation. Then, I will *implement* the server by translating the state machine `RandServer` into the lower level state machines `Net` and `Var`, using the `Transform` interface I briefly introduced in Section 3.3.

5.1 Defining `RandServer`

Figure 2 shows the general form of the states in a server which waits for connections from clients, and on receiving a connection initiates a *session* in which it waits for an incoming message, then replies. There are two parts to this system:

1. **Session initiation:** either the server is not running, or it's ready for an incoming connection. When it receives an incoming connection, it sets up a new *session interaction* state machine for that connection and continues waiting for connections
2. **Session interaction:** the session waits for an incoming message, processes that message and sends a response, then the session is complete

We can represent this system using the following types, where `ServerState` represents the state and `ServerFinal` is a predicate which says that `Idle` and `Done` are final states:

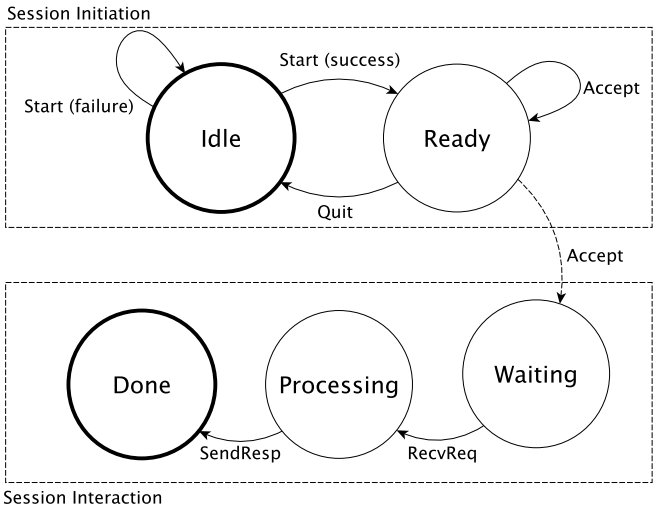


Figure 2. A state transition diagram which shows the states and operations of a server which waits for connections from a client. On accepting a connection, it starts a new `Waiting` machine. The initial state is `Idle` and the final states are `Idle` and `Done`.

```
data ServerState = Idle | Ready | Waiting | Processing | Done

data ServerFinal : ServerState -> Type where
     IdleFinal : ServerFinal Idle
     DoneFinal : ServerFinal Done
```

Listing 9 defines the operations in a random number server, following the transitions given by Figure 2. Like `Net` in the previous section, when it `Accepts` a connection, this starts a new session, with a new resource, so we define this in a separate signature.
Listing 10 shows a main program `serverLoop` for the random number server. It repeatedly accepts connections from clients and responds to requests. An implementation of `serverLoop` which type checks is guaranteed to correctly implement the random number protocol[3].

5.2 Executing `RandServer` using `Transform`

The implementation of `serverLoop` merely *describes* a random number server, however. We still need to provide an `Execute` implementation for `RandServer`:

```
Execute RandServer IO where
```

[3] This does, however, ignore the issue of totality checking. I'll return to this point in Section 7.

Listing 9. Defining operations on a state machine for a random number server

```
data RandOp : SM_sig ServerState where
     Start : RandOp Bool Idle
                    (\res => if res then Ready else Done)
     Quit : RandOp () Ready (const Idle)
     RecvReq : RandOp (Maybe Integer) Waiting
                      (\res => case res of
                                    Nothing => Done
                                    Just _ => Processing)
     SendResp : Integer -> RandOp () Processing (const Done)
     GetSeed : RandOp Integer Ready (const Ready)

data RandCreate : SM_sig ServerState where
     Accept : RandCreate Bool Ready
                         (\res => if res then Waiting else Done)

RandServer : SM ServerState
RandServer = MkSM Idle RandFinal RandOp RandCreate
```

Since we've already implemented a state machine for describing network communication using sockets, it would be natural to implement `RandServer` with it, then we would also have a compile-time guarantee that we are using sockets correctly. Unfortunately, we can't do this using `Execute`, since here it requires us to implement operations directly in terms of the computation context `IO`.

Instead, we can use the `Transform` interface, which allows us to explain how to implement a state machine in terms of lower level machines:

```
interface Transform (sm : SM state) (sms : PList SM)
                    (ops : PList SM) (m : Type -> Type) | sm, m
```

The arguments are: `sm`, which is the higher level state machine being transformed; `sms`, which are the lower level state machines used to implement `sm`; `ops`, which are the new state machines we can create in the process; and `m` which is the context in which the transformed machines will run. Interface resolution uses `sm` and `m` alone to determine the implementation. To implement `Transform`, we must implement the following methods:

- `toState`, which explains how the state of the higher level machine maps to the states of the lower level machines
- `initOK`, which shows that the initial states of the high level and low level machines correspond
- `finalOK`, which shows that the final states of the high level and lowe level machines correspond
- `execAs` and `createAs`, which explain how to implement the `operations` and `creators` respectively

Listing 10. Defining the main loop of a random number server

```
serverLoop : ConsoleIO io => (s : State RandServer) ->
             SMTrans io () [Stable s Ready]
serverLoop s = do
    num <- on s GetSeed
    (True, session) <- newFrom s Accept
         | (False, session) => do delete session
                                  serverLoop s
    Just bound <- on session RecvReq
       | Nothing => do delete session
                       serverLoop s
    on session (SendResp (num `mod` (bound + 1)))
    delete session
    serverLoop s
```

The types of the methods in `Transform` are generic, and calculate the types for state machine programs given the translation from high level states to low level states. Rather than showing the interface declaration in full, therefore, it's informative to see an example of how the method types are constructed in specific cases. We can begin a `Transform` implementation for `RandServer` as follows, implementing the random number server using `Net` and `Var`:

```
ConsoleIO io => Transform RandServer [Net, Var] [Var] io where
    toState Idle = (Closed, ())
    toState Ready = (Listening, Integer)
    toState Waiting = (Open Server, ())
    toState Processing = (Open Server, ())
    toState Done = (Closed, ())

    initOK = Refl
    finalOK Idle IdleFinal = (ClosedFinal, ())
    finalOK Done DoneFinal = (ClosedFinal, ())

    execAs (server, seed) op = ?exec_transform
    createAs (server, seed) op = ?create_transform
```

The `toState` method gives a mapping from higher to lower level states:

- If `Idle`, there is no socket and no seed.
- If `Ready`, the server is `Listening` and there is an `Integer` seed.
- Once there is a session and the server is `Waiting` or `Processing`, the server is `Open`. The seed is associated with the main server loop, so it is `()`.
- When processing is `Done`, the socket is `Closed` and there is no seed.

The listing leaves holes for the definitions of `execAs` and `createAs`. By pattern matching on the `op`, we can see how we need to write the transformation to a program in `SMProg`. For example, in the following case...

```
execAs (server, seed) Quit = ?trans_quit
```

...checking `trans_quit` tells us we have a `server` in the `Listening` state, and an `Integer` seed, and we need to close `server` and remove the `seed`.

```
exec_transform_quit : SMProg io () [Var]
         [server ::: Listening, seed ::: Integer]
         (\result => [server ::: Closed, seed ::: ()])
```

Given an implementation of `Transform sm sms ops m` and implementations of `Execute` for all the necessary machines, we get an implementation of `Execute sm m` for free:

```
(Transform sm sms ops m, ExecList m ops, ExecList m sms)
    => Execute sm m where
```

As a result, our implementation of `Transform` for `RandServer` in `IO` via `Net` and `Var` is enough to give us an implemenation of `Execute RandServer IO`.

All that matters to a user of the `RandServer` state machine is that there is an implementation in the context they need. There is no need to know that it is implemented in terms of `Net` and `Var`, and indeed if necessary we could provide multiple implementations using different underlying transport protocols.

6 Related Work

This paper builds on previous work on algebraic effects in Idris [6,7], and the approach to describing interactive programs in particular is inspired by Hancock and Setzer's approach to IO in dependent type theory [13], defining an `IO` type parameterised by possible commands and responses. An important goal of this research is to be able to implement and verify communicating systems, inspired by Session Types [16,17]. Session Types describe the state of a communication channel, and recent work has shown the correspondence between propositions and sessions [33] and how to implement this in a programming language [23]. The present paper is also related to linear types [31], in that the executing an operation on a state machine *consumes* the current state.

The representation of state transition systems using dependent types owes much to Atkey's study of indexed monads [2], and McBride's implementation of dynamic interaction in Haskell [24]. The state transitions given by operations in `SM` are reminiscent of Hoare Triples [15]. This has previously been implemented as an axiomatic extension to Coq [25]. Unlike the current work, however, these systems do not attempt to compose several independent state transition systems.

Earlier work has recognised the importance of state transition systems in describing applications [14]. In this paper, we have used `states` to describe systems in terms of state transitions on resources, both at the level of external resources like network sockets and at the application level. The problem of reasoning about protocols has previously been tackled using special purpose type systems [34], by creating DSLs for resource management [8], or with Typestate [1,29]. In these approaches, however, it is difficult to compose systems from multiple resources

or to implement a resource in terms of other resources. In `states`, we can combine resources by extending the list of available operations and using `new`, and `Transform` a from a high level state to a lower level.

The description of state machines as algebraic data types is closely related to work on algebraic effects [26,4]. The `Execute` and `Transform` interfaces, in a sense, are *handlers* of algebraic effects. They are, however, more limited, in that the implementations don't forward requests to other handlers. Algebraic effects are increasingly being proposed as an alternative to monad transformers for structuring Haskell applications [21,20]. In previous work, we have explored algebraic effects in Idris [7,11], and this paper addresses several of the weaknesses of earlier implementations. In particular, it is now possible to implement state machines in terms of other, lower level, state machines.

The `SMProg` data type allows us to construct complex data types, with different operations, from individual components. This follows earlier work in combining free monads in Haskell [30], and to some extent begins to addresses the Expression Problem [32] in that we can add functionality to a data type without needed to recompile existing code.

A strong motivation for the work in the present paper is to be able to incorporate a form of Session Types [16,17] into dependently typed applications, while also supporting other stateful components. Type systems in modern programming languages, such as Rust [18] and Haskell [28], are strong enough to support Session Types, and by describing communication protocols in types, we expect to be able to use the type system to verify correctness of implementations of security protocols [12,19].

7 Discussion

I have shown how to describe systems in terms of type-dependent state machines and given some examples which show state machines can be used in practice. In particular, I have shown that we can design larger scale systems by composing state machines both horizontally (that is, using multiple state machines in the same function definition) and vertically (that is, implementing the behaviour of a state machine using other lower level state machines). As well as the examples in this paper, the accompanying code[4] includes examples of a game with the rules encoded in its type, and concurrent programming. In the former example, we have a high level state machine to initiate a game, and a lower level state machine to implement the rules, following the pattern in Section 5.

A significant strength of this approach to structuring dependently typed programs is that state machines are ubiquitous, if implicit, in realistic APIs, especially when dealing with external resources. The `states` library makes these state machines explicit in the types of operations, meaning that we can be sure by type checking that a program correctly follows the state machine. As the `Net` example shows, we can give precise types to an existing API, and use the

[4] `https://github.com/edwinb/States`

operations in more or less the same way as before, with additional confidence in their correctness. Furthermore, as I briefly discussed in Section 4.2, we can use *interactive*, type-driven, program development and see the internal state of a program at any point during development.

Since `SMProg` is parameterised by an underlying computation context, which is most commonly a monad, it is a monad transformer. Also, an `SMProg` which preserves states is itself a monad, so `states` programs can be combined with monad transformers, and therefore are compatible with a common existing method of organising large scale functional programs.

7.1 Limitations and Future work

There are limitations in the system as it stands, which we hope to address in future work. In particular, `states` works will with *resources* but does not fit so well with *control* structures such as exceptions or threads. To some extent, we should not expect exceptions to fit well, because we need to understand program control to know the system state at any point. Threads are a problem because, on forking, we need to consider which thread gets access to which resource (two different threads should not be able to modify the same external resource, for example). We can address this in future work, by defining an appropriate type for `fork` which passes different resources to different threads.

Another limitation is that, although the types help us construct programs interactively, when we make a mistake the error messages can be hard to decipher. I hope to address this in future work using Idris' error reflection mechanism [10].

I have not discussed totality checking in this paper, especially when writing servers which loop indefinitely. In practice, we can only be confident that a program follows a protocol correctly if it is total. Although it possible to implement total servers using a coinductive type, I will defer discussion to future work. We have also not discussed the efficiency of this approach. There is a small overhead due to the interpreter for `SMProg` and the algebraic types for operations, but we expect to be able to eliminate this using a combination of partial evaluation [9] and a finally tagless approach to interpretation.

Most importantly, however, I believe there are several applications to this approach in defining security protocols, and verified implementation of distributed systems. For the former, security protocols follow a clearly defined series of steps and any violation can be disastrous, causing sensitive data to leak. For the latter, we are currently developing an implementation of Session Types [16,17] embedded in Idris, generalising the random number server presented in Section 5.

The `states` library uses a number of generic functions and interfaces, such as `on`, `new`, `Execute` and `Transform`, to implement a useful pattern in dependently typed program in such a way that it is *reusable* by application developers. State is everywhere, and introduces complexity throughout applications. Dependently typed purely functional programming gives us the tools we need to keep this complexity under control.

References

1. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th conference on Object Oriented Programming Systems Languages and Applications*, pages 1015–1012, 2009.
2. R. Atkey. Parameterised notions of computation. In *Proceedings of Workshop on Mathematically Structured Functional Programming (MSFP 2006)*, BCS Electronic Workshops in Computing, 2006.
3. L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter, 1999.
4. A. Bauer and M. Pretnar. Programming with Algebraic Effects and Handlers, 2012. Available from `http://arxiv.org/abs/1203.1539`.
5. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013.
6. E. Brady. 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.
7. E. Brady. Resource-dependent algebraic effects. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming (TFP '14)*, volume 8843 of *LNCS*. Springer, 2014.
8. E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102:145–176, 2010.
9. E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 297–308, New York, NY, USA, 2010. ACM.
10. D. Christiansen. Reflect on your mistakes! lightweight domain-specific error messages, 2014. Draft.
11. S. Fowler and E. Brady. Dependent types for safe and secure web programming. In *Implementation and Application of Functional Languages (IFL)*, 2013.
12. A. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. *Journal of computer security*, 11(4):451–520, 2003.
13. P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proc. of 14th Ann. Conf. of EACSL, CSL'00, Fischbau, Germany, 21–26 Aug 2000*, volume 1862, pages 317–331. Springer-Verlag, Berlin, 2000.
14. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
16. K. Honda. Types for dyadic interaction. In *International Conference on Concurrency Theory*, pages 509–523, 1993.
17. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
18. T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, 2015. ACM.

19. D. Kaloper-Mersinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. In *USENIX Security 2015*, 2015.
20. O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th International Conference on Functional Programming (ICFP '13)*. ACM, 2013.
21. O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM.
22. P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966.
23. S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *ESOP '15)*, 2015.
24. C. McBride. Kleisli arrows of outrageous fortune, 2011.
25. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: reasoning with the awkward squad. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2008. ACM.
26. G. Plotkin and M. Pretnar. Handlers of Algebraic Effects. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 80–94, 2009.
27. J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.
28. R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 25–36, New York, NY, USA, 2008. ACM.
29. R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.
30. W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.
31. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.
32. P. Wadler. The expression problem, 1998. Email, available from `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.
33. P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM.
34. D. Walker. A Type System for Expressive Security Policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 254–267. ACM, 2000.