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

EDWIN BRADY, University of St Andrews

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 an indexed 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, we present an architecture for dependently typed applications based on a hierarchy of state transition systems, implemented in a generic data type `ST`. This is based on a monad indexed by *contexts* of resources, allowing us to reason about multiple state transition systems in the type of a function. Using `ST`, we 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. We illustrate the system by implementing a number of examples, including a graphics API, POSIX network sockets, asynchronous programming with threads, and a high level network application protocol.

**ACM Reference format:**
Edwin Brady. 2016. State Machines All The Way Down. 1, 1, Article 1 (January 2016), 26 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Software relies on state, and many components rely on state machines. For example, they describe network transport protocols like TCP (Postel 1981), 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, we show how to represent state machines precisely in the dependently typed programming language Idris (Brady 2013a), and present a library which allows composing larger programs from multiple state transition systems. The library supports composition in two ways: firstly, we can use several independently implemented state transition systems at once; secondly, we can implement one state transition system in terms of others.

A motivation for this work is to be able to define communicating systems, where the type system verifies that each participant follows a defined protocol. This is inspired by Session Types (Honda 1993; Honda et al. 2008), in which the state of a session at any point represents the allowed interactions on a communication channel.

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 `ST` type we present in this paper will allow us to implement a system as a hierarchy of state machines, and as an example we will implement a client-server system in which a client requests a random number within a specific bound from a server, and in which the server processes requests *asynchronously*. All of the examples are available online (for review: submitted as supplementary material).

## 1.1 Contributions

We build on previous work on algebraic effects in Idris (Brady 2013b, 2014). 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, we address these shortcomings, making the following specific contributions:

- We present a type, `ST`, which allows a programmer to describe a the pre- and post-conditions of a stateful function, creating and destroying resources as necessary (Section 3).
- We describe the implementation of `ST`, in particular using Idris' proof automation to construct proofs of correct state management without programmer intervention (Section 4).
- We show how to use `ST` to describe existing stateful APIs, notably working with network sockets, and to implement high level stateful systems, such as network application protocols, in terms of these lower level systems (Section 5).

An important goal of `ST` is *usability*, providing notation to help write *correct* programs without a significant *cost*. Concretely, this means that the types need to be readable and need to help direct a programmer to a working implementation, and error messages need to explain problems clearly. We therefore use holes (Section 2.3) to help build programs interactively, type level programming (Section 3.4) to provide a readable type level notation, and error reflection (Section 4.5) to rewrite error messages into the language of the problem domain.

Using `ST`, we can encode the assumptions we make about state transitions in a type, and ensure that these assumptions 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), `ST` provides an *architecture* for larger scale dependently typed programs.

## 2 EXAMPLE: A DATA STORE, REQUIRING A LOGIN

Our goal is to use the type system not only to describe the inputs and outputs of functions precisely, but also to explain the effect of functions on any external resources. One way to achieve this is to use an indexed monad (Atkey 2006), where for each operation the monad supports, the type carries a *precondition* which must hold before the operation is executed, and a *postcondition* which holds after the operation is executed, possibly depending on the result of the operation. In a sense, these are Hoare triples (Hoare 1969), embedded in the type. In this section, we introduce this idea with a small example: a data store which requires users to log in before giving access to secret data.

## 2.1 State Transitions in the Store

The store has two states: `LoggedIn`, which means that a user is logged in and able to access data; and `LoggedOut`, which means that the user is not logged in. Reading data is only allowed when the system is in the `LoggedIn` state. Figure 1 illustrates the data store, showing the *states* the system can be in (`LoggedOut` and `LoggedIn`) and the operations a user can perform on the system. The operations are:

- `login`, which is valid when the system is in the `LoggedOut` state, and either results in the system being in the `LoggedIn` state, if login was successful, or the `LoggedOut` state otherwise (for example, due to an incorrectly entered password).
- `logout`, which is valid when the system is in the `LoggedIn` state, and moves the system to the `LoggedOut` state.
- `readSecret`, which returns the data held in the store, and is only valid when the system is in the `LoggedIn` state.

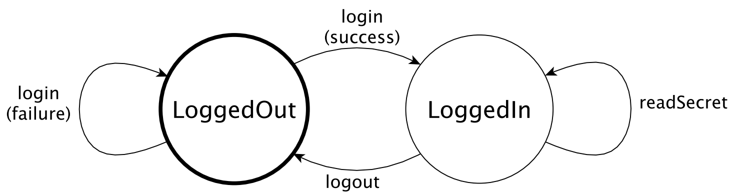


Fig. 1. A state machine describing the states and transitions in a system which allows a program to read some secret data only after successfully logging in to a data store.

By using the type system to capture the state of this system, we can be sure that a program will only succeed in reading data when it has successfully logged in to the system.

## 2.2 Representing State Transitions in a Type

Our goal is that attempting to read data without logging in should result in a static type error. Listing 1 shows how we can achieve this using an indexed monad.

The `Store` indexed by the type of the result of the operation, the required input state, and an output state calculated from the result of the operation. In the simplest cases, such as `Logout`, the result has no influence on the result, so we use `const` (which returns its first argument and discards its second) to compute the result state:

```
Logout : Store () LoggedIn (const LoggedOut)
```

When we `Login`, on the other hand, the result state depends on the result of the operation. If login is successful, it returns `OK`, and the resulting state of the system is `LoggedIn`; otherwise, it is `LoggedOut`:

```
Login : Store LoginResult LoggedOut (\res => case res of
                                                 OK =>          LoggedIn
                                                 BadPassword => LoggedOut)
```

## 2.3 Implementing a program to access the `Store`

Listing 2 shows how to combine operations on the store into a complete function which attempts to login, then reads and prints the data if successful, or prints an error if not.

Rather than writing `getData` in one go, we develop it interactively using *holes*. The following listing contains a hole `getData_rest`, which stands for the rest of the sequence after attempting to login:

Listing 1. Implementing the state store as an indexed monad

```
data Access = LoggedOut | LoggedIn
data LoginResult = OK | BadPassword

data Store : (ty : Type) -> Access -> (ty -> Access) -> Type where
     Login : Store LoginResult LoggedOut
                  (\res => case res of
                                OK =>          LoggedIn
                                BadPassword => LoggedOut)
     Logout :     Store ()     LoggedIn  (const LoggedOut)
     ReadSecret : Store String LoggedIn  (const LoggedIn)

     Pure : (x : ty) -> Store ty (st x) st
     Lift : IO ty -> Store ty st (const st)
     (>>=) : Store a st1 st2 -> ((x : a) -> Store b (st2 x) st3) -> Store b st1 st3
```

Listing 2. A function which logs in to the store and reads the secret data if login succeeds

```
getData : Store () LoggedOut (const LoggedOut)
getData = do result <- Login
             case result of
                  OK => do secret <- ReadSecret
                           Lift (putStr ("Secret: " ++ show secret ++ "\n"))
                           Logout
                  BadPassword => Lift (putStr "Failure\n")
```

```
getData : Store () LoggedOut (const LoggedOut)
getData = do result <- Login
             ?getData_rest
```

Then, if we check the type of the hole, we can see both the current state of the system explicitly in the type, and the required state at the end of the function. Here, we see we need to inspect `result`:

```
getData_rest : Store () (case result of
                              OK => LoggedIn
                              BadPassword => LoggedOut) (const LoggedOut)
```

We leave *execution* of a `Store` program abstract here; this depends on factors such as how to connect to the store, the security policy, and so on. Nevertheless, by defining an indexed monad for the operations on a data store, we can be sure that programs which access the store correctly follow a protocol of logging in before reading data. However, there are limitations to this approach. For example:

- What if we want to write programs which use other external resources, as well as the data store?
- Can we connect to an arbitrary number of stores, rather than being limited to one?
- `Store` describes a session with a *connected* store. How do we handle connecting and disconnecting?

In the rest of this paper, we will decribe a library, `ST`, written in Idris, which allows us to describe APIs such as those provided by `Store`. It will address the above limitations, allowing us to *compose* multiple stateful systems, implement systems in terms of other, lower level systems, and create and destroy resources as required.

## 3 `ST`: DESCRIBING STATE TRANSITIONS IN TYPES

In this section, we introduce the `ST` library. It is built around an indexed monad `STrans` which allows us to compose several stateful systems such as the `Store`. We will see how to describe *interfaces* of stateful systems, how to provide *implementations* of those interfaces which work in a particular monad, then how to combine multiple stateful systems in one program. Finally, we will introduce `ST`, a top level type which allows us to describe the behaviour of a program in terms of a list of *state transitions*.

### 3.1 Generalising Stateful Programs: Introducing `STrans`

We would like to *generalise* the approach taken in defining `Store` to support *multiple* state transition systems at once, and use types to describe not only the transitions in individual systems, but also to describe when new systems are *created* and *destroyed*. Furthermore, to be accessible to developers and practitioners, we aim to provide a *readable* API with helpful error messages. To summarise, we have the following high level requirements:

- Types should capture the *states* of resources. That is, they should: describe *when* operations are valid; and, describe *how* the results of operations affect resources.
- We should be able to use multiple stateful interfaces together. That is, stateful APIs shold *compose*.
- Types should be *readable*, and clearly express the behaviour of operations.
- Error messages should clearly describe when an operation violates the protocol given by the type.

The `ST` library is built around the following indexed monad, which describes sequences of state transitions and expresses in its type how the state transitions affect a collection of resources:

```
data STrans : (m : Type -> Type) -> (ty : Type) ->
              (in_ctxt : Context) -> (out_ctxt : ty -> Context) -> Type
```

The indices of `STrans` are:

- `m` — an underlying monad, in which the stateful program is to be executed
- `ty` — the result type of the program
- `in_ctxt` — an *input context*; a list of variables and their states *before* the program is executed
- `out_ctxt` — an *output context*, calculate from the result of the program, consisting of a list of variables and their states *after* the program is executed.

The input and output contexts correspond to the `Access` parameters of `Store`, except that they store *lists* of a variable name paired with a type, which reflect the current state of a *resource*. For example, a `Context` which contains a reference to a logged out data store and a file open for reading could be written as follows:

```
[store ::: Store LoggedOut, handle ::: File Read]
```

The names `store` and `handle` here are labels, of a type `Var` defined by `ST`, and are used to refer to specific states inside an `STrans` function. The idea, overall, is that we define *interfaces*[1] which describe operations for creating, modifying and updating resources as `STrans` programs.

Listing 3 shows how we can define an interface which supports access to a data store, like the `Store` indexed monad we defined in the previous section. In addition to operations for logging in and out, and reading the secret from the server, this interface provides two further operations for connecting to and disconnecting from a store:

- `connect` begins with no resources available and returns a new variable. The output context states that the variable refers to a store in the `LoggedOut` state:

  ```
  connect : STrans m Var [] (\store => [store ::: Store LoggedOut])
  ```

- `disconnect`, given a resource which *must* be in the `LoggedOut` state, removes that resource:

[1]An interface in Idris is, essentially, like a type class in Haskell, with some minor differences which do not concern us in this paper.

Listing 3. An interface for the data store, using `STrans` to describe the state transitions in each operation

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type

  connect : STrans m Var [] (\store => [store ::: Store LoggedOut])
  disconnect : (store : Var) -> STrans m () [store ::: Store LoggedOut] (const [])

  login : (store : Var) ->
          STrans m LoginResult
                   [store ::: Store LoggedOut]
          (\res => [store ::: Store (case res of
                                          OK => LoggedIn
                                          BadPassword => LoggedOut)])
  logout : (store : Var) ->
           STrans m () [store ::: Store LoggedIn]
                (const [store ::: Store LoggedOut])
  readSecret : (store : Var) ->
               STrans m String [store ::: Store LoggedIn]
                        (const [store ::: Store LoggedIn])
```

```
disconnect : (store : Var) ->
             STrans m () [store ::: Store LoggedOut] (const [])
```

The interface is parameterised by an underlying monad, `m`. To implement this interface[2], as we will do in Section 3.2, we need to explain how to implement the operations in a specific monad.

As before, the types of `login`, `logout` and `readSecret` explain how the operations affect a state, but now we provide a label `store` which refers to a specific state in the `Context`. Furthermore, by encapsulating the operations in an interface, we can constrain the types of functions to allow access to only the interfaces we need. Listing 4 shows how to implement `getData` in this setting. This also uses a `ConsoleIO` interface which provides operations `putStr` and `getStr` for writing to and reading from a console. Each one works in any `Context`, and does not modify any resources:

```
interface ConsoleIO (m : Type -> Type) where
  putStr : String -> STrans m () ctxt (const ctxt)
  getStr : STrans m String ctxt (const ctxt)
```

The constraints on the type of `getData` give the interfaces that it can use (namely `ConsoleIO` and `DataStore`). Also, the type states that there are *no* preconditions; in other words there are no resources on entry, and there are no resources available on exit:

```
getData : (ConsoleIO m, DataStore m) => STrans m () [] (const [])
```

Therefore, to use a `Store`, we must first `connect`, and we must always `disconnect` before returning, no matter which execution path we take. Listing 5 shows a more concise notation (the pattern matching binding alternative `| BadPassword => ...`) introduced by Brady (2014), for checking the result of `login`, and which we will use extensively. This is a convenient notation for working with sequences of operations which might fail, without the need for several nested case blocks. The main path through the program assumes that

[2]In Haskell terms, this would mean providing an instance of the class.

Listing 4. Implementing getData using the DataStore interface

```
getData : (ConsoleIO m, DataStore m) => STrans m () [] (const [])
getData = do st <- connect
             result <- login st
             case result of
                  OK => do secret <- readSecret st
                           putStr ("Secret: " ++ show secret ++ "\n")
                           logout st; disconnect st
                  BadPassword => do putStr "Failure\n"
                                    disconnect st
```

the result of `login` was `OK`, with an alternative path provided for the `BadPassword` case which cleans up by disconnecting. This program desugars into the original version of `getData` in Listing 4.

Listing 5. Implementing getData using the DataStore interface, with a pattern matching bind alternative

```
getData : (ConsoleIO m, DataStore m) => STrans m () [] (const [])
getData = do st <- connect
             OK <- login st | BadPassword => do putStr "Failure\n"
                                                disconnect st
             secret <- readSecret st
             putStr ("Secret: " ++ show secret ++ "\n")
             logout st; disconnect st
```

## 3.2 Executing Stateful Programs

In order to execute a program described by `STrans`, we use the following `run` function:

```
run : Applicative m => ST m a [] -> m a
```

This requires `m` to be `Applicative` for reasons we will see when we discuss the implementation of `ST` in Section 4. We can try to `run getData` in a `main` program, as before:

```
main : IO ()
main = run getData
```

Unfortunately, this doesn't work quite yet, because we have no implementation of the `DataStore` interface for `IO`. The `DataStore` interface allows us to describe how each operation affects the state, but in order to execute the operations, we need to explain how each is implemented concretely. An `implementation` of an interface in Idris corresponds to an `instance` of a type class in Haskell:

```
implementation DataStore IO where
  ...
```

To implement the operations, we need to be able to access the resources in the context directly, so that we can create new concrete resources and read and write existing resources. Listing 6 shows how `Context` is defined. A `Context` is, essentially, a list of `Resource`s, each of which is a pair of a label and a state.

Listing 7 gives the types for the four primitive operations for manipulating contexts: `new`, `delete`, `read` and `write`. Since the `delete` and `write` operations update the context at the type level, they rely on a predicate

Listing 6. Resources, contexts and context membership

```
data Resource : Type where
     (:::) : label -> state -> Resource

data Context : Type where
     Nil : Context
     (::) : Resource -> Context -> Context
```

`InState`. A value of type `InState lbl ty ctxt` means that the resource with label `lbl` in a context `ctxt` has type `ty`. Informally, these operations do the following:

- `new`, given a value of type `state`, returns a new label, and adds an entry `lbl ::: State state` to the context.
- `delete`, given a label and a proof that the label is available, removes that label from the context.
- `read`, given a label and a proof that the label is available, returns the value stored for that label.
- `write`, given a label, a proof that the label is available, and a new value, updates the value.

In the types, the notation `{auto prf : t}` means that Idris will attempt to construct the argument `prf` automatically, by searching recursively through the constructors of `t`, up to a certain depth limit, and use the first value which type checks. This is a notational convenience for programmers, and avoids the need for writing proof terms directly when the proof can be determined statically.

Listing 7. Primitive operations in `STrans` for creating, deleting, reading and writing resources

```
new : (val : state) -> STrans m Var ctxt (\lbl => (lbl ::: State state) :: ctxt)
delete : (lbl : Var) -> {auto prf : InState lbl (State st) ctxt} ->
         STrans m () ctxt (const (drop ctxt prf))
read : (lbl : Var) -> {auto prf : InState lbl (State ty) ctxt} ->
       STrans m ty ctxt (const ctxt)
write : (lbl : Var) -> {auto prf : InState lbl ty ctxt} ->
        (val : ty') -> STrans m () ctxt (const (updateCtxt ctxt prf (State ty')))
```

The reason we wrap types in a type constructor `State` is that it allows implementations of interfaces to *hide* the implementation details of resources. It is defined as follows:

```
data State : Type -> Type where
     Value : ty -> State ty
```

When we implement `DataStore`, for example, we will need to give a concrete type for `Store`. In the simplest case, we can use a `String` for this, to represent the secret data:

```
implementation DataStore IO where
  Store x = State String
  ...
```

Since we can *only* use `delete` and `read` on resources with a type wrapped in `State`, we can be sure that we can *only* use `delete` and `read` when we know the concrete monad in which we are running. For example, `getData` is constrained rather than in a concrete monad:

```
getData : (ConsoleIO m, DataStore m) => STrans m () [] (const [])
```

There is no way for `getData` to get access to the secret data through `read` or delete it with `delete`, since it does not know which implementation is being used, and therefore does not know that `Store` is necessarily implemented by a type wrapped in `State`. This suggests that, for safety, it is good practice for `STrans` programs to use interface constraints rather than concrete underlying monads in their types.

For reference, listing 8 gives the types of `InState`, `drop` and `updateCtxt`. These are fairly standard: `InState` is essentially an index into the context; `drop` removes the context entry given by the index; and `updateCtxt` updates the type at the location given by the index.

Listing 8. The `InState` predicate, which describes the type of a context entry, and functions for manipulating the context

```
data InState : lbl -> state -> Context -> Type where
     Here : InState lbl st ((lbl ::: st) :: rs)
     There : InState lbl st rs -> InState lbl st (r :: rs)

drop : (ctxt : Context) -> (prf : InState lbl st ctxt) -> Context
updateCtxt : (ctxt : Context) -> InState lbl st ctxt -> state -> Context
```

We are now in a position to complete the implementation of `DataStore` for `IO`. When we `connect`, we create a new data store with `new` and initialise it with some secret data:

```
connect = do store <- new "Secret Data"
             pure store
```

The type of `connect` requires that, on exit, there is a new entry in the context with type `State String`. Similarly, the type of `disconnect` requires that, on exit, the context entry has been removed. To `disconnect`, we merely remove the `store` from the context:

```
disconnect store = delete store
```

To `login`, we prompt the user for a password (hard coded here), and return success if it is correct:

```
login store = do putStr "Enter password: "
                 p <- getStr
                 if p == "Mornington Crescent"
                    then pure OK else pure BadPassword
```

Since the data has type `State String` whether the user is logged in or logged out, `logout` in this case does not need to do anything:

```
logout store = pure ()
```

Finally, to implement `readSecret`, we use `read` to access the store, which is allowed because in this implementation we know that the `Store` is implemented as a `State String`:

```
readSecret store = read store
```

## 3.3 Programs with Multiple States and Resources

So far, we have only used one resource, a data store connection. Realistically, we will need to deal with multiple states and resources in a single program. For example, instead of printing the value we have read from the data store, we might want to write it to a local file, assuming we have a `File` resource available and open for writing. We could try to do this as follows:

```
writeToFile : (FileIO m, DataStore m) =>
              (h : Var) -> (st : Var) ->
              STrans m () [h ::: File {m} Write, st ::: Store {m} LoggedIn]
                   (const [h ::: File {m} Write, st ::: Store {m} LoggedIn])
writeToFile h st = do secret <- readSecret st
                      writeLine h secret
```

Note that the implicit argument `{m}` needs to be passed to `File` and `Store` so that interface resolution can see which implementation to use for each. The pre- and post-conditions of `writeToFile` both say that we have a file `h` which is open for writing, and a store `st` which is logged in. Unfortunately, Idris reports an error[3]:

```
Error in state transition:
      Operation has preconditions: [store ::: Store LoggedIn]
      States here are: [h ::: File Write,
                        st ::: Store LoggedIn]
```

The problem is that `readSecret` requires a single data store to be available, but we have a larger set of states available. We can resolve this problem using the `call` function, which reduces the available states to only those required for the operation, then rebuilds the states once the operation is complete:

```
call : STrans m t sub new_f -> {auto ctxt_prf : SubCtxt sub old} ->
       STrans m t old (\res => updateWith (new_f res) old ctxt_prf)
```

This has some similarities with the frame rule in Separation Logic (O'Hearn et al. 2001), in that it permits us to reason locally about a subprogram's effect on resources, without any effect on other resources. `SubCtxt sub old` is a proof that the context `sub` is smaller than the context `old` (reordering is allowed), and `updateWith` rebuilds the parent context using the context updates `new_f` given by the called function. We will look at the implementation of `call`, and especially `SubCtxt`, in Section 4.3. Using `call`, we can correct `writeToFile` as follows:

```
writeToFile h st = do secret <- call (readSecret st)
                      call (writeLine h secret)
```

Optionally, to reduce the syntactic noise, we can allow `call` to be implicitly inserted by Idris to correct any such errors, using the `implicit` keyword:

```
implicit call : STrans m t sub new_f -> {auto ctxt_prf : SubCtxt sub old} ->
                STrans m t old (\res => updateWith (new_f res) old ctxt_prf)
```

While this makes our original version of `writeToFile` work, and improves readability, it does come at the expense of making error messages potentially difficult to understand, because it is unpredictable whether the error message refers to code with or without the implicit `call`. Therefore, by default `call` must be explicit, and a user of `STrans` needs to import an additional module to make it implicit. Nevertheless, for readability, in the rest of this paper we will leave `call` *implicit*.

## 3.4 Cleaning up the types: Introducing `ST`

The type of `STrans` expresses precisely what the input and output states of an operation are, and therefore we can express complete preconditions and explain how the result of an operation affects the state. However, these can be quite verbose, especially when the function does not change a state. So, instead, we define a type level function `ST` which allows us to express state changes in terms of complete transitions on individual resources:

[3]The error message is rewritten using error reflection, which we discuss briefly in Section 4.5

```
ST : (m : Type -> Type) -> (ty : Type) -> List (Action ty) -> Type
ST m ty xs = STrans m ty (in_res xs) (\result : ty => out_res result xs)
```

This converts a list of `Actions` on labelled resources into separate *input* resources and *output* resources. An `Action` is defined as follows:

```
data Action : Type -> Type where
     Stable : lbl -> Type -> Action ty
     Trans : lbl -> Type -> (ty -> Type) -> Action ty
     Remove : lbl -> Type -> Action ty
     Add : (ty -> Context) -> Action ty
```

`Action` is parameterised by the result type of an operation, and an `Action` says one of the following about the effect an operation has on a resource:

- `Stable lbl st` says that the resource `lbl` is in the state `st` before and after the operation
- `Trans lbl st st_f` says that the resource `lbl` begins in the state `st` and has an output state computed from the result of the operation by `st_f`.
- `Remove lbl st` says that the resource `lbl` begins in the state `st` and is deleted during execution of the operation
- `Add st_f` says that the operation introduces new resources as described by the function `st_f`.

The functions `in_res` and `out_res` compute the relevant input and output contexts from a list of actions:

```
in_res : (as : List (Action ty)) -> Context
out_res : ty -> (as : List (Action ty)) -> Context
```

To provide a consistent and readable notation which makes the state transitions visible directly in the type, we define infix operators using Idris' type-directed overloading mechanism. Listing 9 shows the final definition of the `DataStore` interface, in which each state transition is written in one of the following forms:

- Using `Add`, to describe which resources are added.
- Using `Remove`, to describe which resources are removed.
- `store ::: State`, which expresses that `store` begins and ends in `State` (i.e. `Stable store State`)
- `store ::: InState :-> OutState`, which expresses that `store` begins in the state `InState` and ends in `OutState`, no matter what the result of the operation (i.e. `Trans store InState (const Outstate)`)
- `store ::: InState :-> (\res => OutState)`, which expresses that `store` begins in the state `InState` and ends in `OutState`, which may be computed from the result of the operation `res` (i.e. `Trans store InState (\res => Outstate)`)

This notation is defined by using overloaded operators `:::` and `:->` to translate the state transition written using infix operators into an `Action`. In the rest of this paper, we will use this notation to describe state transition interfaces, using `ST`.

## 4 IMPLEMENTATION

As we have seen, `STrans` allows us to write programs which describe sequences of state transitions, creating and destroying resources as necessary. In this section, we will describe some implementation details of `STrans`. In particular, we will describe how a `Context` corresponds to an environment containing concrete values at run time, and see the concrete definition of `STrans` itself. We will show the mechanism for calling subprograms with smaller sets of resources, and show how to build *composite* resources, consisting of a list of independent resources. Finally, we will briefly describe how Idris' error reflection mechanism (Christiansen 2016) lets us rewrite type

Listing 9. An interface for the data store, using ST to describe the state transitions in each operation

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type
  connect : ST m Var [Add (\store => [store ::: Store LoggedOut])]
  disconnect : (store : Var) -> ST m () [Remove store (Store LoggedOut)]
  login : (store : Var) ->
        ST m LoginResult [store ::: Store LoggedOut :->
                            (\res => Store (case res of
                                                OK => LoggedIn
                                                BadPassword => LoggedOut))]
  logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
  readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
```

error messages to describe errors in terms of the problem domain, rather than in terms of the implementation language.

## 4.1 Environments and Execution

The indices of STrans describe how a sequence of operations affects the types of elements in a context. Correspondingly, when we run an STrans program, we need to keep track of the *values* of those elements in an environment, defined as follows:

```
data Env : Context -> Type where
     Nil : Env []
     (::) : ty -> Env xs -> Env ((lbl ::: ty) :: xs)
```

Then, when running a program, we provide an *input* environment, and a continuation which explains how to process the result of the program and the output environment, whose type is calculated from the result:

```
runST : Env invars -> STrans m a invars outfn ->
        ((x : a) -> Env (outfn x) -> m b) -> m b
```

At the top level, we can run a program with no resources on input and output:

```
run : Applicative m => ST m a [] -> m a
run prog = runST [] prog (\res, _ => pure res)
```

The Applicative constraint on run is required so that we can inject the result of the computation into the underlying computation context m. In the special case that m is the identity function, we can return the result directly instead:

```
runPure : ST id a [] -> a
runPure prog = runST [] prog (\res, _ => res)
```

Finally, in some cases we might want to execute an STrans program with an initial environment and read the resulting environment, which we can achieve using runWith provided that we are executing the program in the IO monad.

```
runWith : Env ctxt -> STrans IO a ctxt (\res => ctxtf res) ->
          IO (res ** Env (ctxtf res))
runWith env prog = runST env prog (\res, env' => pure (res ** env'))
```

It is important to restrict this to a specific monad, rather than allowing `runWith` to be parameterised over any monad `m` like `run`. The reason is that the intention of `STrans` is to control all accesses to state via `read` and `write`, but `runWith` gives us a convenient "escape hatch" if we need more flexibility to modify the environment in a concrete `IO` setting. We will need this when implementing asynchronous programs in Section 5.4. By restricting `runWith` to work in a *concrete* monad, we know that it is at least impossible to use it in programs which are written in a generic context. For example, we saw `writeToFile` earlier:

```
writeToFile : (FileIO m, DataStore m) => (h : Var) -> (st : Var) ->
              ST m () [h ::: File {m} Write, st ::: Store {m} LoggedIn]
```

We know that it is impossible for `writeToFile` to call `runWith`, and possibly introduce new items into the environment, because it is parameterised over some monad `m`.

## 4.2 Defining `STrans`

`STrans` itself is defined as an algebraic data type, describing operations for reading, writing, creating and destroying resources, and sequencing stateful operations. Additionally, there are operations for manipulating the context in some more advanced ways. The complete definition is shown in Listing 10.

Listing 10. The complete definition of `STrans` as an Idris data type

```
data STrans : (m : Type -> Type) -> (ty : Type) ->
              Context -> (ty -> Context) -> Type where
     Pure : (result : ty) -> STrans m ty (out_fn result) out_fn
     Bind : STrans m a st1 st2_fn ->
            ((result : a) -> STrans m b (st2_fn result) st3_fn) ->
            STrans m b st1 st3_fn
     Lift : Monad m => m t -> STrans m t ctxt (const ctxt)
     New : (val : state) ->
           STrans m Var ctxt (\lbl => (lbl ::: State state) :: ctxt)
     Delete : (lbl : Var) -> (prf : InState lbl (State st) ctxt) ->
              STrans m () ctxt (const (drop ctxt prf))
     DropSubCtxt : (prf : SubCtxt ys xs) -> STrans m (Env ys) xs (const (kept prf))
     Split : (lbl : Var) -> (prf : InState lbl (Composite vars) ctxt) ->
             STrans m (VarList vars) ctxt
                    (\vs => mkCtxt vs ++ updateCtxt ctxt prf (State ()))
     Combine : (comp : Var) -> (vs : List Var) ->
               (prf : VarsIn (comp :: vs) ctxt) ->
               STrans m () ctxt (const (combineVarsIn ctxt prf))
     Call : STrans m t ys ys' -> (ctxt_prf : SubCtxt ys xs) ->
            STrans m t xs (\res => updateWith (ys' res) xs ctxt_prf)
     Read : (lbl : Var) -> (prf : InState lbl (State ty) ctxt) ->
            STrans m ty ctxt (const ctxt)
     Write : (lbl : Var) -> (prf : InState lbl ty ctxt) -> (val : ty') ->
             STrans m () ctxt (const (updateCtxt ctxt prf (State ty')))
```

The operations we have described so far, in Section 3, are implemented by using the corresponding constructor of `STrans`. The main difference is that proof terms (such as the `prf` arguments of `Delete` and `Read`) are explicit, rather than marked as implicit with `auto`. There are four constructors we have not yet encountered. These are `Lift`, `DropSubCtxt`, `Split` and `Combine`:

- `Lift` allows us to use operations in the underlying monad `m`. We will need this to implement interfaces in terms of existing monads.
- `DropSubCtxt` allows us to remove a subset of resources, and returns their values as an environment. The output context is calculated using a function `kept`, which returns the resources which are not part of the subset. We will use this to share resources between multiple threads in Section 5.4.
- `Split` and `Combine` allow us to work with *composite* resources. We will discuss these shortly in Section 4.4, and we will need them to be able to implement one stateful interface in terms of a collection of others. For example, in Section 5.2, we will implement a small high level graphics API in terms of resources describing a low level graphics context, and some internal state.

The implementation of `runST` is by pattern matching on `STrans`, and largely uses well known implementation techniques for well-typed interpreters with dependent types (Augustsson and Carlsson 1999; Pašalic et al. 2002). Most of the implementation is about managing the environment, particularly when interpreting `Call`, `Split` and `Combine`, which involve restructuring the environment according to the proofs. When we interpret `New`, we need to provide a new `Var`. The `Var` type is defined as follows, and perhaps surprisingly has only one value:

```
data Var = MkVar
```

We do not need any more than this because internally all references to variables are managed using proofs of context membership, such as `InState` and `SubCtxt`. The variable of type `Var` gives a human readable name, and a way of disambiguating resources by Idris variable names, but by the time we execute a program with `runST`, these variables have been resolved into proofs of context membership so we do not need to construct any unambiguous concrete values.

## 4.3 Calling subprograms

We often need to call subprograms with a *smaller* set of resources than the caller, as we saw in Section 3.3. To do this, we provide a proof that the input resources required by the subprogram (`sub`) are a subset of the current input resources (`old`):

```
Call : STrans m t sub new_f -> (ctxt_prf : SubCtxt sub old) ->
       STrans m t old (\res => updateWith (new_f res) old ctxt_prf)
```

A value of type `SubCtxt sub old` is a proof that every element in `sub` appears exactly once in `old`. In other words, it is a proof that the resources in `sub` are a subset of those in `old`. To show this, we need to be able to show that a specific `Resource` is an element of a `Context`:

```
data ElemCtxt : Resource -> Context -> Type where
     HereCtxt : ElemCtxt a (a :: as)
     ThereCtxt : ElemCtxt a as -> ElemCtxt a (b :: as)
```

Then, a proof of `SubCtxt sub old` essentially states, for each entry in `old`, either where it appears in `sub` (using `InCtxt` and a proof of its location using `ElemCtxt`) or that it is not present in `sub` (using `Skip`):

```
data SubCtxt : Context -> Context -> Type where
     SubNil : SubCtxt [] []
     InCtxt : (el : ElemCtxt x ys) -> SubCtxt xs (dropEl ys el) ->
              SubCtxt (x :: xs) ys
     Skip : SubCtxt xs ys -> SubCtxt xs (y :: ys)
```

A context is a subcontext of itself, so we can say that `[]` is a subcontext of `[]`. To ensure that there is no repetition in the sub-context, the recursive argument to `InCtxt` explicitly states that the resource cannot appear in the remaining sub-context, using `dropEl`:

```
dropEl : (ys: _) -> ElemCtxt x ys -> Context
dropEl (x :: as) HereCtxt = as
dropEl (x :: as) (ThereCtxt p) = x :: dropEl as p
```

The type of `Call` relies on the following function, which calculates a new context for the calling function, based on the updates made by the subprogram:

```
updateWith : (new : Context) -> (old : Context) -> SubCtxt sub old -> Context
```

The result of `updateWith` is the contents of `new`, adding those values in `old` which were previously `Skip`ped as described by the proof of `SubCtxt sub old`. Correspondingly, there is a function used by `runST` which rebuilds an environment after evaluating the `Call`ed subprogram:

```
rebuildEnv : Env new -> Env old -> (prf : SubCtxt sub old) ->
             Env (updateWith new old prf)
```

When we use `Call`, the contexts described by `SubCtxt` are known at compile time, from the type of the subprogram and the state of the `Context` at the call site. Idris' built in proof search, which searches constructors up to a depth limit until it finds an application which type checks, is strong enough to find these proofs without programmer intervention, so in the top level `call` function we mark the proof argument as `auto`:

```
call : STrans m t sub new_f -> {auto ctxt_prf : SubCtxt sub old} ->
       STrans m t old (\res => updateWith (new_f res) old ctxt_prf)
```

## 4.4 Composite Resources

A *composite* resource is built from a list of other resources, and is essentially defined as a heterogeneous list:

```
data Composite : List Type -> Type
```

If we have a composite resource, we can split it into its constituent resources, and create new variables for each of those resources, using the following top level function which is defined using `Split`, again using proof search to find the label in the context:

```
split : (lbl : Var) -> {auto prf : InState lbl (Composite vars) ctxt} ->
        STrans m (VarList vars) ctxt
                 (\vs => mkCtxt vs ++ updateCtxt ctxt prf (State ()))
```

A `VarList` is a list of variable names, one for each resource in the composite resource. We use `mkCtxt` to convert it into a fragment of a context where the composite resource has been split into independent resources:

```
VarList : List Type -> Type
mkCtxt : VarList tys -> Context
```

After splitting the composite resource, the original resource is replaced with unit `()`. We can see the effect of this in the following code fragment, in a function which is intended to swap two integers in a composite resource:

```
swap : (comp : Var) -> ST m () [comp ::: Composite [State Int, State Int]]
swap comp = do [val1, val2] <- split comp
               ?rest
```

If we check the type of the hole `?rest`, we see that `val1` and `val2` are now individual resources:

```
rest : STrans m () [val1 ::: State Int, val2 ::: State Int, comp ::: State ()]
            (const [comp ::: Composite [State Int, State Int]])
```

The type of `?rest` shows that, on exit, we need a composite resource again. We can build a composite resource from individual resources using `combine`, implemented in terms of the corresponding `STrans` constructor:

```
combine : (comp : Var) -> (vs : List Var) ->
          {auto prf : InState comp} -> {auto var_prf : VarsIn (comp :: vs) ctxt} ->
          STrans m () ctxt (const (combineVarsIn ctxt var_prf))
```

Similar to `SubCtxt`, `VarsIn` is a proof that every variable in a list appears once in a context. Then, `combineVarsIn` replaces those variables with a single `Composite` resource. Correspondingly, in the implementation of `runST`, `rebuildVarsIn` updates the environment:

```
rebuildVarsIn : Env ctxt -> (prf : VarsIn (comp :: vs) ctxt) ->
                Env (combineVarsIn ctxt prf)
```

Using `combine`, we can reconstruct the context in `swap` with the resources swapped:

```
swap : (comp : Var) -> ST m () [comp ::: Composite [State Int, State Int]]
swap comp = do [val1, val2] <- split comp
               combine comp [val2, val1]
```

## 4.5 Improving Error Messages with Error Reflection

Helpful error messages make an important contribution to the usability of any system. We have used Idris' error message reflection (Christiansen 2016) to rewrite the errors produced by Idris to explain the relevant part of specific errors, namely, the preconditions and postconditions on operations. Consider the following incorrect code fragment, using the data store defined in Section 3.1 but attempting to read without first logging in:

```
badGet : DataStore m => ST m () []
badGet = do st <- connect
            secret <- readSecret st
            ?more
```

By default, Idris reports the following as part of the error message:

```
When checking an application of function Control.ST.>>=:
        Type mismatch between
                STrans m String [st ::: Store LoggedIn]
                       (\result =>
                          [st ::: Store LoggedIn]) (Type of readSecret st)
        and     STrans m String [st ::: Store LoggedOut]
                       (\result => [st ::: Store LoggedIn]) (Expected type)
```

This includes the relevant information, that the store needs to be `LoggedIn` but in fact is `LoggedOut`, but the important parts are hidden inside a larger term. However, recognising that errors arising from running an operation when the preconditions do not match always have the same form, we can extract the pre- and postconditions from the message, and rewrite it to the following using an *error handler*:

```
Error in state transition:
    Operation has preconditions: [st ::: Store LoggedIn]
    States here are: [st ::: Store LoggedOut]
    Operation has postconditions: \result => [st ::: Store LoggedIn]
    Required result states here are: \result => [st ::: Store LoggedIn]
```

The full details are beyond the scope of this paper, but they rely on the following function which inspects a reflected error message, and returns a new message if the original message has a particular form:

```
st_precondition : Err -> Maybe (List ErrorReportPart)
```

## 5 EXAMPLES

The ST library can be used in any situation which calls for tracking of the states of resource in types. In this section, we present several diverse examples which show how the library allows us to compose stateful systems both horizontally (that is, using multiple stateful systems at once) and vertically (that is, implementing one stateful system in terms of one or more others).

### 5.1 A Graphics Interface

Listing 11 shows an interface for a small graphics library which supports creating a window and drawing into that window. The flip operation supports double buffering. We create a state of type Surface using the following function:

```
initWindow : Int -> Int -> ST m (Maybe Var) [addIfJust Surface]
```

This uses a type level function addIfJust, provided by the ST library, which allows us to write concisely in a type that the function will add a resource when it successfully returns a variable:

```
addIfJust : Type -> Action (Maybe a)
addIfJust ty = Add (maybe [] (\var => [var ::: ty]))
```

Listing 11. The Draw interface which supports drawing lines in a window

```
interface Draw (m : Type -> Type) where
  Surface : Type

  initWindow : Int -> Int -> ST m (Maybe Var) [addIfJust Surface]
  closeWindow : (win : Var) -> ST m () [Remove win Surface]
  flip : (win : Var) -> ST m () [win ::: Surface]
  filledRectangle : (win : Var) -> (Int, Int) -> (Int, Int) -> Col ->
                    ST m () [win ::: Surface]
  drawLine : (win : Var) -> (Int, Int) -> (Int, Int) -> Col ->
             ST m () [win ::: Surface]
```

We can provide an implementation for this interface using the SDL graphics library[4], for example:

```
implementation Draw IO where
  Surface = State SDLSurface
  ...
```

Having defined a primitive interface for graphics operations, we can use it as a basis for more high level interfaces. For example, we can build a library for turtle graphics by building a *composite* resource from a Surface and the state of a turtle (including location, direction and pen colour).

### 5.2 Turtle Graphics

Turtle graphics involves a "turtle" manoeuvring around a screen, drawing lines as it moves. It has attributes describing its location, direction, and pen colour. There are commands for moving the turtle forwards, turning through an angle, and changing colour. Listing 12 gives one possible interface using ST.

[4]https://www.libsdl.org/

Listing 12. An interface for turtle graphics, supporting creating a turtle, drawing lines, and rendering the result

```
interface TurtleGraphics (m : Type -> Type) where
  Turtle : Type

  start : Int -> Int -> ST m (Maybe Var) [addIfJust Turtle]
  end : (t : Var) -> ST m () [Remove t Turtle]
  fd : (t : Var) -> Int -> ST m () [t ::: Turtle]
  rt : (t : Var) -> Int -> ST m () [t ::: Turtle]
  col : (t : Var) -> Col -> ST m () [t ::: Turtle]
  render : (t : Var) -> ST m () [t ::: Turtle]
```

The `start` function initialises the system with window dimensions. Then, the state of the turtle can be updated by issuing commands. Finally, the `render` function displays the picture drawn so far in a window. The following function, for example, initialises a turtle and, if successful, draws a coloured square:

```
square : (ConsoleIO m, TurtleGraphics m) => ST m () []
square = do Just t <- start 640 480 | Nothing => putStr "Can't make turtle\n"
            col t yellow; fd t 100; rt t 90; col t green; fd t 100; rt t 90
            col t red; fd t 100; rt t 90; col t blue; fd t 100; rt t 90
            render t; end t
```

We can implement the interface using `Draw`, and render using a `Surface` but we need additional state to represent the current state of the turtle, and to record what we need to draw when `render` is called. We can achieve this using a composite resource:

```
implementation Draw m => TurtleGraphics m where
  Turtle = Composite [Surface {m}, State Col, State (Int, Int, Int),
                      State (List Line)]
  ...
```

The components are: the `Surface` to draw on; the current pen colour, the current turtle location and direction, and a list of lines to draw. When we initialise the system, we create each component of the composite resource then `combine` them:

```
start x y = do Just srf <- initWindow x y | Nothing => pure Nothing
               col <- new white; pos <- new (320, 200, 0)
               lines <- new []; turtle <- new ()
               combine turtle [srf, col, pos, lines]
               pure (Just turtle)
```

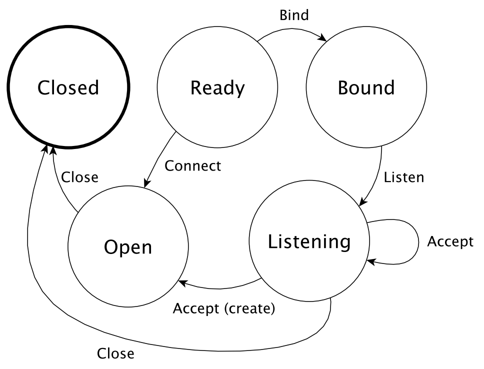
In each operation, to access an individual element of the composite resource, we need to `split` the resource, then `combine` the components when done. For example, to turn right:

```
rt t angle = do [srf, col, pos, lines] <- split t
                (x, y, d) <- read pos
                write pos (x, y, d + angle `mod` 360)
                combine t [srf, col, pos, lines]
```

In this way, we can implement a larger system as a *hieararchy* of smaller systems. The `Turtle`, as far as the programmer who writes `square` is concerned, is an individual system, but internally there are state machines all the way down to the concrete implementations of `Draw` and `State`.

## 5.3 Socket Programming, with Error Handling

The POSIX sockets API supports communication between processes across a network. A *socket* represents an endpoint of a network communication, and can be in one of several states: `Ready`, the initial state; `Bound`, meaning that it has been bound to an address ready for incoming connections; `Listening`, meaning that it is listening for incoming connections; `Open`, meaning that it is ready for sending and receiving data; and `Closed` meaning that it is no longer active. The diagram on the right shows how the operations provided by the API modify the state, where `Ready` is the initial state.



Listing 13. The POSIX sockets API, written as in interface for `ST` describing how each operation affects socket state

```
data SocketState = Ready | Bound | Listening | Open | Closed

interface Sockets (m : Type -> Type) where
  Sock : SocketState -> Type

  socket : SocketType -> ST m (Either () Var) [addIfRight (Sock Ready)]
  bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
    ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Bound)]
  listen : (sock : Var) ->
    ST m (Either () ()) [sock ::: Sock Bound :-> (Sock Closed `or` Sock Listening)]
  accept : (sock : Var) ->
    ST m (Either () Var) [addIfRight (Sock Open), sock ::: Sock Listening]
  connect : (sock : Var) -> SocketAddress -> Port ->
    ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Open)]
  close : (sock : Var) -> {auto prf : CloseOK st} ->
    ST m () [sock ::: Sock st :-> Sock Closed]
  remove : (sock : Var) -> ST m () [Remove sock (Sock Closed)]
  send : (sock : Var) -> String ->
    ST m (Either () ()) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
  recv : (sock : Var) ->
    ST m (Either () String) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
```

Listing 13 shows how these operations can be given precise types which describe how the operations affect socket state, using `ST`. By convention, each operation returns something with a type of the form `Either a b`, representing the possibility of failure. The `ST` library provides two helper functions which allow us to write concise types for these operations:

```
addIfRight : Type -> Action (Either a b)
addIfRight ty = Add (either (const []) (\var => [var ::: ty]))

or : a -> a -> Either b c -> a
or x y = either (const x) (const y)
```

Note, in particular, that `accept` explicitly creates a new `Open` socket specifically for processing an incoming connection, keeping the existing socket in a `Listening` state. This could allow, for example, processing an incoming connection in a different thread:

```
accept : (sock : Var) ->
     ST m (Either () Var) [addIfRight (Sock Open), sock ::: Sock Listening]
```

We could use `Sockets` to write an "echo" server which repeatedly accepts an incoming connection from a client, and echoes a message back to the client. We can define a function `echoServer` which accepts connections on a `Listening` socket, and logs messages to the console:

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
                ST m () [Remove sock (Sock {m} Listening)]
```

Then, before we start up `echoServer`, we need to create a socket, bind it to a port, then begin listening for incoming connections. Each of these operations could fail, so we include pattern matching alternatives to clean up the resources if necessary:

```
startServer : (ConsoleIO io, Sockets io) => ST io () []
startServer = do Right sock <- socket Stream         | Left err => pure ()
                 Right ok <- bind sock Nothing 9442 | Left err => remove sock
                 Right ok <- listen sock            | Left err => remove sock
                 echoServer sock
```

We can begin writing `echoServer` as follows, accepting a connection or cleaning up the resources and returning if `accept` fails:

```
echoServer sock =
  do Right new <- accept sock | Left err => do close sock; remove sock
     ?rest
```

Checking the type of `?rest` here shows us that we have a `new` socket, which is `Open` and therefore ready to communicate with the client, as well as `sock` which remains listening for new connections:

```
rest : STrans m () [new ::: Sock Open, sock ::: Sock Listening]
                   (\result1 => [])
```

## 5.4 Asynchronous Programming with Threads

In `ST`, resources are linear, in that there is exactly one reference to each, and once a resource has been overwritten the old value is no longer available. So, if we spawn a thread, we need to consider how to preserve linearity, and maintain only one reference to each resource. Listing 14 shows one way to do this, where the `fork` function takes a thread described in `STrans`, and a proof that the forked thread uses a subcontext of the parent thread. Then, the parent thread keeps only the resources which are not used by the child thread.

Listing 14. An interface supporting asynchronous programming, dividing resources between a child and a parent thread

```
interface Conc (m : Type -> Type) where
  fork : (thread : STrans m () thread_res (const [])) ->
         {auto tprf : SubCtxt thread_res all} ->
         STrans m () all (const (kept tprf))
```

An implementation of `Conc` then needs to divide the resources appropriately. We can achieve this with `dropSubCtxt` (to remove the subcontext from the current thread, returning the environment) and `runWith` (to pass that environment to the spawned thread):

```
implementation Conc IO where
  fork thread = do threadEnv <- dropSubCtxt
                   lift (spawn (do runWith threadEnv thread
                                   pure ()))
                   pure ()
```

The Idris library defines `spawn` to create a new thread. It returns a process identifier, to allow communication between threads, but we leave communication between threads for future work.

## 5.5 Managing Sessions: A Random Number Server

In the turtle graphics example, we implemented a high level graphics API in terms of lower level drawing operations. Similarly, we can implement a high level network application protocol in terms of sockets. Listing 15 shows an interface for a server which replies to requests from a client for a random number within a bound.

Listing 15. An interface for a server which returns random numbers within a given bound

```
data SessionState = Waiting | Processing | Done

interface RandomSession (m : Type -> Type) where
  Connection : SessionState -> Type
  Server : Type

  recvReq : (conn : Var) ->
    ST m (Maybe Integer) [conn ::: Connection Waiting :->
                          \res => Connection (case res of
                                                   Nothing => Done
                                                   Just _ => Processing)]
  sendResp : (conn : Var) -> Integer ->
             ST m () [conn ::: Connection Processing :-> Connection Done]
  start : ST m (Maybe Var) [addIfJust Server]
  quit : (srv : Var) -> ST m () [Remove srv Server]
  done : (conn : Var) -> ST m () [Remove conn (Connection Done)]
  accept : (srv : Var) ->
           ST m (Maybe Var) [srv ::: Server, addIfJust (Connection Waiting)]
```

The interface defines two types:

- `Connection`, which is the current state of a session with a client. A session is either `Waiting` for the client to send a request, `Processing` the request from the client, or `Done` and ready to close.
- `Server`, which is the type of a server which processes incoming client requests.

Overall, a server listens for an incoming request from a client, and when it receives a request, it initialises a session with the client and continues waiting for further incoming requests. In a session, we can call one of:

- `recvReq`, which, if we're in the `Waiting` state, receives a request from a client, and if it is valid moves into the `Processing` state.

- `sendResp`, which, if we're in the `Processing` state, sends a random number within the given bound. Note that `sendResp` itself is intended to do the work of generating the random number.
- `done`, which closes the session and removes the `Connection` state.

To implement a random number server, we'll use this interface, as well as `ConsoleIO` for console logging, and `Conc` to process requests asynchronously. To avoid repetition in every function signature, Idris provides a notation to allow us to state that every function in a block is constrained by the same interfaces:

```
using (ConsoleIO m, RandomSession m, Conc m)
```

We implement a session with `rndSession` which, given a `Connection` in the `Waiting` state will process a client request and eventually delete the connection, using `recvReq`, `sendResp` and `done`:

```
rndSession : (conn : Var) -> ST m () [Remove conn (Connection {m} Waiting)]
```

The main loop of the server calls `accept` to receive an incoming connection. If it fails, it reports an error. Otherwise, it uses `fork` to process the incoming connection in a separate thread. The resources are divided between `rndSession` (whose type states that it receives the `Connection` variable `conn`) and the parent thread, which retains the `Server` variable `srv`:

```
rndLoop : (srv : Var) -> ST m () [srv ::: Server {m}]
rndLoop srv = do Just conn <- accept srv | Nothing => putStr "accept failed\n"
                 putStr "Connection received\n"
                 fork (rndSession conn)
                 rndLoop srv

rndServer : ST m () []
rndServer = do Just srv <- start | Nothing => putStr "Can't start server\n"
               rndLoop srv; quit srv
```

Finally, we can implement the interface using composite resources for both the `Connection` and the `Server`. Each one carries an `Integer` seed for the random number generator, and a socket for receiving incoming connections. In the case of `Connection`, the socket will be in a different state depending on the current state of the session. In particular, once the session is `Done`, the socket will need to be `Closed`.

```
implementation (ConsoleIO m, Sockets m) => RandomSession m where
  Connection Waiting = Composite [State Integer, Sock {m} Open]
  Connection Processing = Composite [State Integer, Sock {m} Open]
  Connection Done = Composite [State Integer, Sock {m} Closed]
  Server = Composite [State Integer, Sock {m} Listening]
  ...
```

## 6 RELATED WORK

This paper builds on previous work on algebraic effects in Idris (Brady 2013b, 2014), and the implementation of `STrans` follows many of the ideas used in these earlier implementations. However, this earlier work had several limitations, most importantly that it was not possible to implement one effectful API in terms of others, and that it was difficult to describe the relationship between separate resources, such as the fact that `accept` creates a new resource in a specific initial state. The work in the present paper is influenced in particular by previous work on Separation Logic, Linear Types and Session Types, and has connections with other type systems and tools for formal verification. In this section we discuss these and other connections.

*Separation Logic and Indexed Monads:* The representation of state transition systems using dependent types owes much to Atkey's study of indexed monads (Atkey 2006), and McBride's implementation of dynamic interaction in Haskell (McBride 2011). The state transitions given by operations in `ST`, like in this previous work on indexed monads, are reminiscent of Hoare Triples (Hoare 1969). Separation Logic (Reynolds 2002), an extension of Hoare Logic, permits local reasoning by allowing a heap to be split into two disjoint parts, like the `call` function in `ST`. This has previously been implemented as Ynot, an axiomatic extension to Coq (Nanevski et al. 2008). Separation Logic allows us to reason about the state of a program's heap. The key distinction between this previous work and `ST` is that we reason about individual resources, such as files and network sockets, rather than about the entire heap, and list these resources in a context. While Separation Logic can prove more complex properties, our approach has a clean embedding in the host language, which leads to readable types and, because resources are combined in predictable ways, minimises the proof obligations for the programmer. Indeed, the examples in Section 5 require *no* additional effort from the programmer to prove that resources are used correctly.

*Linear Types:* Indexed monads, as used in `ST`, are a way of encoding linearity (Abramsky 1993; Wadler 1990) in a dependently typed host language. Recent work by Krishnaswami et al. (2015) and by McBride (2016) has shown ways of integrating linear and dependent types, and the latter has been implemented as an experimental extension to Idris. Our experience has been that linear types can provide the same guarantees as the `ST` library, but that, subjectively, they have a more "low-level" feel than the indexed monad approach. A particular notational inconvenience is the need for a function which updates a linear resource to return a new reference to the resource, a detail which is abstracted away by the indexed monad. Nevertheless, we hope to explore the connection further in future work. In particular, linear dependent types may give us an efficienct method for implementing `ST`.

*Types and State Machines:* Earlier work has recognised the importance of state transition systems in describing applications (Harel 1987). In this paper, we have used `ST` 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 (Walker 2000), by creating DSLs for resource management (Brady and Hammond 2010a), or with Typestate (Aldrich et al. 2009; Strom and Yemini 1986). In these approaches, however, it is difficult to compose systems from multiple resources or to implement a resource in terms of other resources. In `ST`, we can combine resources by using additional interfaces, and implement those interfaces in terms of other lower level resources.

*Types for Communication:* A strong motivation for the work in the present paper is to be able to incorporate a form of Session Types (Honda 1993; Honda et al. 2008) into dependently typed applications, while also supporting other stateful components. Session Types describe the state of a communication channel, and recent work has shown the correspondence between propositions and sessions (Wadler 2012) and how to implement this in a programming language (Lindley and Morris 2015). More recently, Toninho and Yoshida (2016) have shown how to increase the expressivity of multi-party session to capture invariants on data. We expect to be able to encode similar properties in `ST` and in doing so, be able to encode security properties of communicating systems with dependent types, following (Guenot et al. 2015). Type systems in modern programming languages, such as Rust (Jespersen et al. 2015) and Haskell (Pucella and Tov 2008), 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 (Gordon and Jeffrey 2003; Kaloper-Mersinjak et al. 2015).

*Effects and Modules:* Algebraic effects (Bauer and Pretnar 2015; Plotkin and Pretnar 2009) are increasingly being proposed as an alternative to monad transformers for structuring Haskell applications (Kammar et al. 2013; Kiselyov et al. 2013). Earlier work in Idris (Brady 2013b, 2014) has used a system based on algebraic effects to track resource state, like `ST` in this paper. Koka (Leijen 2017) uses row polymorphism to describe the allowed effects of a function, which is similar to the `Context` of an `ST` program in that a function's type lists only the parts of the overall state it needs.

In ST, rather than using effects and handlers, we use *interfaces* to constrain the operations on a resource, and provide *implementations* of those interfaces. This does not have the full power of handlers of algebraic effects—in particular, we cannot reorder handlers to implement exception handling or other control flow—but it does allow us to express the state transitions precisely. Indeed, we do not want handlers to be *too* flexible: for example, a handler which implements non-determinism by executing an operation several times would violate linear access to a resource. Interfaces in Idris are similar to Haskell type classes, but with support for *named* overlapping instances. This means that we can give multiple different interpretations to an interface in different implementations, and in this sense they are similar in power to ML modules (Dreyer 2005). Interfaces in Idris are first class, meaning that implementations can be calculated by a function, and thus provide the same power as first-class modules in 1ML (Rossberg 2015).

## 7 CONCLUSIONS AND FURTHER WORK

We have shown how to describe APIs in terms of type-dependent state transition systems, using a library ST and evaluated the library by presenting several diverse examples which show how ST can be used in practice. In particular, we have shown how to 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).

A significant strength of this approach to structuring programs is that state machines are ubiquitous, if implicit, in realistic APIs, especially when dealing with external resources like surfaces on which to draw graphics, endpoints in network communication, and databases. The ST library makes these state machines explicit in types, meaning that we can be sure by type checking that a program correctly follows the state machine. As the Sockets interface shows, we can give precise types to an existing API (in this case, the POSIX sockets API), and use the operations in more or less the same way, with additional confidence in their correctness. Furthermore, as we briefly discussed in Section 2.3, and saw throughout the paper, we can use *interactive*, type-driven, program development and see the internal state of a program at any point during development.

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

There are several avenues for further work. For example, we have not discussed the efficiency of our approach in this paper. There is a small overhead due to the interpreter for ST but we expect to be able to eliminate this using a combination of partial evaluation (Brady and Hammond 2010b) and a finally tagless approach to interpretation (Carette, Kiselyov, and Shan Carette et al.). We may also be able to use linear types for the underlying implementation (McBride 2016) using an experimental extension to Idris.

Most importantly, however, we 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 (Honda 1993; Honda et al. 2008) embedded in Idris, generalising the random number server presented in Section 5.5.

The ST library provides a generic interface for implementing 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

Samson Abramsky. 1993. Computational Interpretations of Linear Logic. *Theor. Comput. Sci.* 111, 1-2 (April 1993), 3–57. DOI:http://dx.doi.org/10.1016/0304-3975(93)90181-R

J Aldrich, J Sunshine, D Saini, and Z Sparks. 2009. Typestate-oriented programming. In *Proceedings of the 24th conference on Object Oriented Programming Systems Languages and Applications*. 1015–1012. http://dl.acm.org/citation.cfm?id=1640073

Robert Atkey. 2006. Parameterised Notions of Computation. In *Proceedings of Workshop on Mathematically Structured Functional Programming (MSFP 2006) (BCS Electronic Workshops in Computing)*.

L. Augustsson and M. Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*. Citeseer. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2895

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. DOI:http://dx.doi.org/10.1016/j.jlamp.2014.02.001

Edwin Brady. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (September 2013), 552–593. Issue 05. DOI:http://dx.doi.org/10.1017/S095679681300018X

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

Edwin Brady. 2014. Resource-dependent Algebraic Effects. In *Trends in Functional Programming (TFP '14) (LNCS)*, Jurriaan Hage and Jay McCarthy (Eds.), Vol. 8843. Springer.

E Brady and K Hammond. 2010a. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae* 102 (2010), 145–176. http://iospress.metapress.com/index/Q3118847K1351421.pdf

Edwin Brady and Kevin Hammond. 2010b. 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*. ACM, New York, NY, USA, 297–308. DOI:http://dx.doi.org/10.1145

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (????), 509–543. http://journals.cambridge.org/abstract_S0956796809007205

Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 609–622. DOI:http://dx.doi.org/10.1145/2676726.2677003

David Raymond Christiansen. 2016. *Practical Reflection and Metaprogramming for Dependent Types*. Ph.D. Dissertation. IT University of Copenhagen.

D Dreyer. 2005. *Understanding and Evolving the ML Module System*. Ph.D. Dissertation. Carnegie Mellon University.

A.D. Gordon and A. Jeffrey. 2003. Authenticity by Typing for Security Protocols. *Journal of computer security* 11, 4 (2003), 451–520.

Nicolas Guenot, Daniel Gustafsson, and Nicola Pouillard. 2015. Dependent Communication in Type Theory. (2015). https://nicolaspouillard.fr/publis/ptt1.pdf

David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274. DOI:http://dx.doi.org/10.1016/0167-6423(87)90035-9

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

Kohei Honda. 1993. Types for Dyadic Interaction. In *International Conference on Concurrency Theory*. 509–523.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. 273–284.

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming (WGP 2015)*. ACM, New York, NY, USA, 13–22. DOI:http://dx.doi.org/10.1145/2808098.2808100

David Kaloper-Mersinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. In *USENIX Security 2015*.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th International Conference on Functional Programming (ICFP '13)*. ACM.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, New York, NY, USA, 59–70. DOI:http://dx.doi.org/10.1145/2503778.2503791

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. *SIGPLAN Not.* 50, 1 (Jan. 2015), 17–30. DOI:http://dx.doi.org/10.1145/2775051.2676969

Daan Leijen. 2017. Type Directed Compilation of Row-typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 486–499. DOI:http://dx.doi.org/10.1145/3009837.3009872

Sam Lindley and J. Garrett Morris. 2015. A semantics for propositions as sessions. In *ESOP '15)*.

Conor McBride. 2011. Kleisli Arrows of Outrageous Fortune. (2011).

Conor McBride. 2016. 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.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/1411204.1411237

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, London, UK, UK, 1–19. http://dl.acm.org/citation.cfm?id=647851.737404

Emir Pašalic, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming*, Vol. 37. 218–229. DOI: http://dx.doi.org/10.1145/583852.581499

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*. 80–94.

J. Postel. 1981. Transmission Control Protocol. RFC 793 (Standard). (Sept. 1981). http://www.ietf.org/rfc/rfc793.txt Updated by RFCs 1122, 3168.

Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/1411286.1411290

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.

Andreas Rossberg. 2015. 1ML: Core and Modules United (F-ing First-Class Modules). In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. 35–47. DOI: http://dx.doi.org/10.1145/2784731.2784738

RE Strom and S Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6312929

Bernardo Toninho and Nobuko Yoshida. 2016. Certifying data in multiparty session types. In *A List of Successes that can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (LNCS)*. Springer.

Philip Wadler. 1990. Linear types can change the world!. In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, M. Broy and C. Jones (Eds.). North Holland, 347–359.

Philip Wadler. 2012. Propositions As Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 273–286. DOI: http://dx.doi.org/10.1145/2364527.2364568

David Walker. 2000. A Type System for Expressive Security Policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, 254–267. DOI: http://dx.doi.org/10.1145/325694.325728