Type-Driven Development with Idris

by Edwin Brady

Chapter 13

brief contents

13 State machines: verifying protocols in types

This chapter covers

- Specifying protocols in types
- Describing preconditions and postconditions of operations
- Using dependent types in state

In the previous chapter, you saw how to manage mutable state by defining a type for representing sequences of commands in a system, and a function for running those commands. This follows a common pattern: the data type *describes* a sequence of operations, and the function *interprets* that sequence in a particular context. For example, `State` describes sequences of stateful operations, and `runState` interprets those operations with a specific initial state.

In this chapter, we'll look at one of the advantages of using a type for describing sequences of operations and keeping the execution function separate. It allows you to make the descriptions more precise, so that certain operations can only be run when the state has a specific form. For example, some operations require access to a resource, such as a file handle or database connection, before they're executed:

- You need an open file handle to read from a file successfully.
- You need a connection to a database before you can run a query on the database.

When you write programs that work with resources like this, you're really working with a *state machine*. A database client might have two states, such as `Closed` and `Connected`, referring to its connection status to a database. Some operations (such as querying the database) are only valid in the `Connected` state; some (such as connecting to the database) are only valid in the `Closed` state; and some (such as connecting and closing) also change the state of the system. Figure 13.1 illustrates this system.

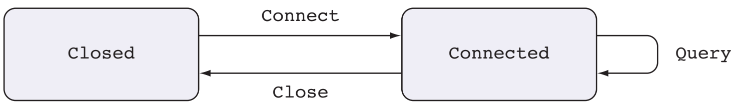

Figure 13.1 A state transition diagram showing the high-level operation of a database. It has two possible states, `Closed` and `Connected`. Its three operations, `Connect`, `Query`, and `Close`, are only valid in specific states.

State machines like the one illustrated in figure 13.1 exist, implicitly, in lots of real-world systems. When you're implementing communicating systems, for example, whether over a network or using concurrent processes, you need to make sure each party is following the same communication pattern, or the system could deadlock or behave in some other unexpected way. Each party follows a state machine where sending or receiving a message puts the overall system into a new state, so it's important that each party follows a clearly defined protocol. In Idris, we have an expressive type system, so if there's a model for a protocol, it's a good idea to express that in a type, so that you can use the type to help implement the protocol accurately.

In this chapter, you'll see how to make state machines like the one illustrated in figure 13.1*explicit* in types. In this way, you can be sure that any function that correctly describes a sequence of actions follows the protocol defined by a state machine. Not only that, you can take a type-driven approach to defining sequences of actions using holes and interactive development. We'll begin with some fairly abstract examples to illustrate how you can describe state machines in types, modeling the states and operations on a door and a vending machine.

13.1 State machines: tracking state in types

You've previously implemented programs with state by defining a type that describes commands for reading and writing state. With dependent types, you can make the types of these commands more precise and include any relevant details about the state of the system in the type itself.

For example, let's consider how to represent the state of a door with a doorbell. A door can be in one of two states, open (represented as `DoorOpen`) or closed (represented as `DoorClosed`), and we'll allow three operations:

- Opening the door, which moves the system from the `DoorClosed` state to the `DoorOpen` state
- Closing the door, which moves the system from the `DoorOpen` state to the `DoorClosed` state
- Ringing the doorbell, which we'll only allow when the door is in the `DoorClosed` state

Figure 13.2 is a state transition diagram that shows the states the system can be in and how each operation modifies the overall state.

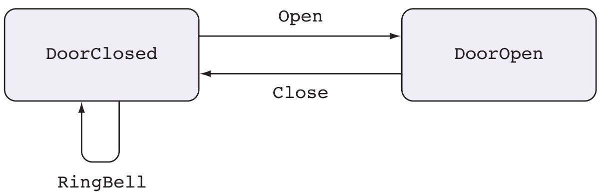

Figure 13.2 A state transition diagram showing the states and operations on a door

If you can define these state transitions in a type, then a well-typed description of a sequence of operations must correctly follow the rules shown in the state transition diagram. Furthermore, you'll be able to use holes and interactive editing to find out which operations are valid at a particular point in a sequence.

In this section, you'll see how to define state machines like the door in a dependent type. First, we'll implement a model of the door, and then we'll model more-complex states in a model of a simplified vending machine. In each case, we'll focus on the *model* of the state transitions, rather than a concrete implementation of the machine.

13.1.1 Finite state machines: modeling a door as a type

The state machine in figure 13.2 describes a *protocol* for correct use of a door by saying which operations are valid in which state, and how those operations affect the state. Listing 13.1 shows one way to represent the possible operations. This also includes a `(>>=)` constructor for sequencing and a `Pure` constructor for producing pure values.

Listing 13.1 Representing operations on a door as a command type (Door.idr)

```
data DoorCmd : Type where
     Open : DoorCmd ()
     Close : DoorCmd ()
     RingBell : DoorCmd ()

     Pure : ty -> DoorCmd ty
     (>>=) : DoorCmd a -> (a -> DoorCmd b) -> DoorCmd b
```

Changes the state of the door from DoorClosed to DoorOpen (Open)

Changes the state of the door from DoorOpen to DoorClosed (Close)

REMINDER: (>>=) AND DO NOTATION Remember that do notation translates into applications of `(>>=)`.

With DoorCmd, you can write functions like the following, which describes a sequence of operations for ringing a doorbell and opening and then closing the door, correctly following the door-usage protocol:

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

Unfortunately, you can also describe *invalid* sequences of operations that don't follow the protocol, such as the following, where you attempt to open a door twice, and then ring the doorbell when the door is already open:

```
doorProgBad : DoorCmd ()
doorProgBad = do Open
                 Open
                 RingBell
```

You can avoid this, and limit functions with DoorCmd to valid sequences of operations that do follow the protocol, by keeping track of the door's state in the type of the DoorCmd operations. The following listing shows how to do this, describing exactly the state transitions represented in figure 13.2 in the types of the commands.

Listing 13.2 Modeling the door state machine in a type, describing state transitions in the types of the commands (Door.idr)

```
data DoorState = DoorClosed | DoorOpen        -- Defines the two possible states of a door

data DoorCmd : Type ->                        -- The type of the result of the operation
               DoorState ->                   -- The state of the door before the operation
               DoorState ->                   -- The state of the door after the operation
               Type where
     Open : DoorCmd     () DoorClosed DoorOpen
     Close : DoorCmd    () DoorOpen   DoorClosed
     RingBell : DoorCmd () DoorClosed DoorClosed

     Pure : ty -> DoorCmd ty state state      -- Produces a value without affecting the state
     (>>=) : DoorCmd a state1 state2 ->
             (a -> DoorCmd b state2 state3) ->   -- Sequences two operations. The output state of the first gives the input state of the second.
             DoorCmd b state1 state3          -- Combined operation goes from the input state of the first operation to the output state of the second
```

Each command's type takes three arguments:

- The type of the value produced by the command
- The *input* state of the door; that is, the state the door must be in *before* you can execute the operation
- The *output* state of the door; that is, the state the door will be in *after* you execute the operation

An implementation of the following function would therefore describe a sequence of actions that begins and ends with the door closed:

```
doorProg : DoorCmd () DoorClosed DoorClosed
```

> **ARGUMENT ORDER IN DOORCMD** Notice that the type that a sequence of operations produces is the first argument to `DoorCmd`, and it's followed by the input and output states. This is a common convention when defining types for describing state transitions, and it will become important in chapter 14 when we look at more-complex state machines that deal with errors and feedback from the environment.

In general, if you have a value of type `DoorType ty beforeState afterState`, it describes a sequence of door actions that produces a value of type `ty`; it begins with the door in the state `beforeState`; and it ends with the door in the state `afterState`.

13.1.2 Interactive development of sequences of door operations

To see how the types in `DoorCmd` can help you write sequences of operations correctly, let's reimplement `doorProg`. We'll write this in the same way as before: ring the doorbell, open the door, and close the door.

If you write it incrementally, you'll see how the type shows the changes in the state of the door throughout the sequence of actions:

1. *Define*—Begin with the skeleton definition:

   ```
   doorProg : DoorCmd () DoorClosed DoorClosed
   doorProg = ?doorProg_rhs
   ```

2. *Refine, type*—Add an action to ring the doorbell:

   ```
   doorProg : DoorCmd () DoorClosed DoorClosed
   doorProg = do RingBell
                 ?doorProg_rhs
   ```

   If you check the type of `?doorProg_rhs` now, you'll see that it should be a sequence of actions that begins and ends with the door in the `DoorClosed` state:

   ```
   --------------------------------------
   doorProg_rhs : DoorCmd () DoorClosed DoorClosed
   ```

3. *Refine, type*—Next, add an action to open the door:

   ```
   doorProg : DoorCmd () DoorClosed DoorClosed
   doorProg = do RingBell
                 Open
                 ?doorProg_rhs
   ```

   If you check the type of `?doorProg_rhs` now, you'll see that it should begin with the door in the `DoorOpen` state instead:

   ```
   --------------------------------------
   doorProg_rhs : DoorCmd () DoorOpen DoorClosed
   ```

4 *Refine* failure—If you add an extra Open now, with the door already in the DoorOpen state, you'll get a type error:

```
doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = do RingBell
              Open
              Open
              ?doorProg_rhs
```

The error says that the type of Open is an operation that starts in the DoorClosed state, but the expected type starts in the DoorOpen state:

```
Door.idr:20:15:
When checking right hand side of doorProg with expected type
        DoorCmd () DoorClosed DoorClosed

When checking an application of constructor Main.>>=:
        Type mismatch between
                DoorCmd () DoorClosed DoorOpen (Type of Open)
        and
                DoorCmd a DoorOpen state2 (Expected type)

        Specifically:
                Type mismatch between
                        DoorClosed
                and
                        DoorOpen
```

5 *Refine*—Instead, complete the definition by closing the door:

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

Defining preconditions and postconditions in types

The type of doorProg includes input and output states that give preconditions and postconditions for the sequence (the door must be closed both before and after the sequence). If the definition violates either, you'll get a type error.

For example, you might forget to close the door:

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

In this case, you'll get a type error:

```
Door.idr:18:15:
When checking right hand side of doorProg with expected type
        DoorCmd () DoorClosed DoorClosed

When checking an application of constructor Main.>>=:
        Type mismatch between
                DoorCmd () DoorClosed DoorOpen (Type of Open)
```

(continued)

```
        and
                DoorCmd () DoorClosed DoorClosed (Expected type)

        Specifically:
                Type mismatch between
                        DoorOpen
                and
                        DoorClosed
```

The error refers to the final step and says that `Open` moves from `DoorClosed` to `DoorOpen`, but the expected type is to move from `DoorClosed` to `DoorClosed`.

By defining `DoorCmd` in this way, with the input and output states explicit in the type, you've defined what it means for a sequence of door operations to be valid. And by writing `doorProg` incrementally, with a sequence of steps and a hole for the rest of the definition, you can see the state of the door at each stage by looking at the type of the hole.

The door has exactly two states, `DoorClosed` and `DoorOpen`, and you can describe exactly when you change states from one to the other in the types of the door operations. But not all systems have an exact number of states that you can determine in advance. Next, we'll look at how you can model systems with an infinite number of possible states.

13.1.3 Infinite states: modeling a vending machine

In this section, we'll model a vending machine using type-driven development, writing types that explicitly describe the input and output states of each operation. As a simplification, the machine accepts only one type of coin (a £1 coin) and dispenses one product (a chocolate bar). Even so, there could be an arbitrarily large number of coins or chocolate bars in the machine, so the number of possible states is not finite.

Table 13.1 describes the basic operations of a vending machine, along with the state of the machine before and after each operation.

Table 13.1 Vending machine operations, with input and output states represented as `Nat`

Coins (before)	Chocolate (before)	Operation	Coins (after)	Chocolate (after)
pounds	chocs	Insert coin	S pounds	chocs
S pounds	S chocs	Vend chocolate	pounds	chocs
pounds	chocs	Return coins	Z	chocs

As with the door example, each operation has a precondition and a postcondition:

- *Precondition*—The number of coins and amount of chocolate that must be in the machine before the operation

- *Postcondition*—The number of coins and amount of chocolate in the machine after the operation.

You can represent the state of the machine as a pair of two `Nat`s, the first representing the number of coins in the machine and the second representing the number of chocolates:

```
VendState : Type
VendState = (Nat, Nat)
```

The next listing shows a representation of the vending machine state as an Idris type, with the state transitions from table 13.1 explicitly written in the types of the `MachineCmd` operations.

Listing 13.3 Modeling the vending machine in a type, describing state transitions in the types of commands (Vending.idr)

```
VendState : Type
VendState = (Nat, Nat)

data MachineCmd : Type ->
                  VendState ->

                  VendState ->

                  Type where
     InsertCoin : MachineCmd () (pounds, chocos)     (S pounds, chocs)
     Vend       : MachineCmd () (S pounds, S chocs) (pounds, chocs)
     GetCoins   : MachineCmd () (pounds, chocs)      (Z, chocs)
```

A type synonym for the machine state: a pair of the number of £1 coins and the number of chocolates

Machine state before the operation (precondition)

Machine state after the operation (postcondition)

To complete the model, you'll need to be able to sequence commands. You'll also need to be able to read user input: the commands you're defining describe what the *machine* does, but there's also a user interface that consists of the following:

- A coin slot
- A vend button, for dispensing chocolate
- A change button, for returning any unused coins

You can model these operations in a data type for describing possible user inputs. Listing 13.4 shows the complete model of the vending machine, including additional operations for displaying a message (`Display`), refilling the machine (`Refill`), and reading user actions (`GetInput`).

Listing 13.4 The complete model of vending machine state (Vending.idr)

```
data Input = COIN
           | VEND
           | CHANGE
           | REFILL Nat
```

Defines the possible user inputs

```
data MachineCmd : Type -> VendState -> VendState -> Type where
     InsertCoin : MachineCmd () (pounds, chocs)     (S pounds, chocs)
     Vend       : MachineCmd () (S pounds, S chocs) (pounds, chocs)
     GetCoins   : MachineCmd () (pounds, chocs)     (Z, chocs)
         Refill :(bars : Nat) ->
                  MachineCmd () (Z, chocs)          (Z, bars + chocs)

     Display : String -> MachineCmd () state state
     GetInput : MachineCmd (Maybe Input) state state

     Pure : ty -> MachineCmd ty state state
     (>>=) : MachineCmd a state1 state2 ->
             (a -> MachineCmd b state2 state3) ->
             MachineCmd b state1 state3

data MachineIO : VendState -> Type where
     Do : MachineCmd a state1 state2 ->
          (a -> Inf (MachineIO state2)) -> MachineIO state1

namespace MachineDo
     (>>=) : MachineCmd a state1 state2 ->
             (a -> Inf (MachineIO state2)) -> MachineIO state1
     (>>=) = Do
```

Refilling the machine is only valid if there are no coins in the machine.

Displaying a message doesn't affect the state.

Reading user input doesn't affect the state. Returns Maybe Input to account for possible invalid inputs.

An infinite sequence of machine state transitions. The type gives the starting state of the machine.

Supports do notation for infinite sequences of machine state transitions

13.1.4 A verified vending machine description

Listing 13.5 shows the outline of a function that describes verified sequences of operations for a vending machine using the state transitions defined by `MachineCmd`. As long as it type-checks, you know that you've correctly sequenced the operations, and you'll never execute an operation without its precondition being satisfied.

Listing 13.5 A main loop that reads and processes user input to the vending machine (Vending.idr)

```
mutual
  vend : MachineIO (pounds, chocs)
  vend = ?vend_rhs

  refill : (num : Nat) -> MachineIO (pounds, chocs)
  refill = ?refill_rhs

  machineLoop : MachineIO (pounds, chocs)
  machineLoop =
       do Just x <- GetInput
               | Nothing => do Display "Invalid input"
                               machineLoop
          case x of
              COIN => do InsertCoin
                         machineLoop
              VEND => vend
```

vend and refill need to check their preconditions are satisfied.

User input could be invalid, so check here.

A pattern-matching binding alternative (see chapter 5). This branch is executed if GetInput returns Nothing.

```
              CHANGE => do GetCoins
                           Display "Change returned"
                           machineLoop
              REFILL num => refill num
```

There are holes for vend and refill. In each case, you need to check that the number of coins and chocolates satisfy their preconditions. If you try to Vend without checking the precondition, Idris will report an error:

```
vend : MachineIO (pounds, chocs)
vend = do Vend
          Display "Enjoy!"
          machineLoop
```

Doesn't type-check because there may not be coins or chocolate in the machine

Idris will report an error because you haven't checked whether there's a coin in the machine and a chocolate bar available, so the precondition might not be satisfied:

```
Vending.idr:67:13:
When checking right hand side of vend with expected type
        MachineIO (pounds, chocs)

When checking an application of function Main.MachineDo.>>=:
        Type mismatch between
                MachineCmd ()
                           (S pounds1, S chocs2)
                           (pounds1, chocs2) (Type of Vend)
        and
                MachineCmd () (pounds, chocs) (pounds1, chocs2) (Expected type)

        Specifically:
                Type mismatch between
                        S chocs1
                and
                        chocs
```

The error says that the input state must be of the form (S pounds1, S chocs2), but instead it's of the form (pounds, chocs).

You can solve this problem by pattern matching on the implicit arguments, pounds and chocs, to ensure they're in the right form, or display an error otherwise. The following listing shows definitions of vend and refill that do this.

Listing 13.6 Adding definitions of `vend` and `refill` that check that their preconditions are satisfied (Vending.idr)

```
vend : MachineIO (pounds, chocs)
vend {pounds = S p} {chocs = S c}
      = do Vend
           Display "Enjoy!"
           machineLoop
vend {pounds = Z}
      = do Display "Insert a coin"
           machineLoop
vend {chocs = Z}
      = do Display "Out of stock"
```

A coin and a chocolate are available, so vend and continue.

No money in the machine; can't vend

No chocolate in the machine; can't vend

```
          machineLoop

refill : (num : Nat) -> MachineIO (pounds, chocs)
refill {pounds = Z} num
      = do Refill num
           machineLoop
refill _ = do Display "Can't refill: Coins in machine"
              machineLoop
```

Refill only allows restocking with chocolate when there are no coins in the machine.

With both the door and the vending machine, we've used types to *model* the states of a physical system. In each case, the type gives an abstraction of the state a system is in before and after each operation, and values in the type describe the valid sequences of operations. We haven't implemented a `run` function to execute the state transitions for either `DoorCmd` or `MachineCmd`, but in the code accompanying this book, which is available online, you'll find code that implements a console simulation of the vending machine.

In the next section, you'll see a more concrete example of tracking state in the type, implementing a stack data structure. I'll use this example to illustrate how you can execute commands in practice.

Exercises

1. Change the `RingBell` operation so that it works in any state, rather than only when the door is closed. You can test your answer by seeing that the following function type-checks:

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

2. The following (incomplete) type defines a command for a guessing game, where the input and output states are the number of remaining guesses allowed:

```
data GuessCmd : Type -> Nat -> Nat -> Type where
     Try : Integer -> GuessCmd Ordering ?in_state ?out_state

     Pure : ty -> GuessCmd ty state state
     (>>=) : GuessCmd a state1 state2 ->
             (a -> GuessCmd b state2 state3) ->
             GuessCmd b state1 state3
```

The `Try` command returns an `Ordering` that says whether the guess was too high, too low, or correct, and that changes the number of available guesses. Complete the type of `Try` so that you can only make a guess when there's at least one guess allowed, and so that guessing reduces the number of guesses available.

If you have a correct answer, the following definition should type-check:

```
threeGuesses: GuessCmd () 3 0
threeGuesses = do Try 10
```

```
                        Try 20
                        Try 15
                        Pure ()
```

Also, the following definition shouldn't type-check:

```
noGuesses : GuessCmd () 0 0
noGuesses = do Try 10
               Pure ()
```

3 The following type defines the possible states of matter:

```
data Matter = Solid | Liquid | Gas
```

Define a `MatterCmd` type in such a way that the following definitions type-check:

```
iceSteam : MatterCmd () Solid Gas
iceSteam = do Melt
              Boil

steamIce : MatterCmd () Gas Solid
steamIce = do Condense
              Freeze
```

Additionally, the following definition should *not* type-check:

```
overMelt : MatterCmd () Solid Gas
overMelt = do Melt
              Melt
```

13.2 Dependent types in state: implementing a stack

You've seen how to model state transitions in a type for two abstract examples: a door (representing whether it was open or closed in its type) and a vending machine (representing its contents in its type). Storing this abstract information in the type of the operations is particularly useful when you also have *concrete* data that relates to that abstract data. For example, if you're describing data of a specific size, and the type of an operation tells you how it changes the size of the data, you can use a `Vect` as a concrete representation. You'll know the required length of the input and output `Vect` from the type of each operation.

In this section, you'll see how this works by implementing operations on a *stack* data structure. A stack is a last-in, first-out data structure where you can add items to and remove them from the top of the stack, and only the top item is ever accessible. A stack supports three operations:

- `Push`—Adds a new item to the top of the stack
- `Pop`—Removes the top item from the stack, provided that the stack isn't empty
- `Top`—Inspects the top item on the stack, provided that the stack isn't empty

Like the operations on the vending machine, each of these operations has a precondition that describes the necessary input state and a postcondition describing the output state. Table 13.2 describes these, giving the required stack size before each operation and the resulting stack size after the operation.

Table 13.2 Stack operations, with input and output stack sizes represented as `Nat`

Stack size (before)	Operation	Stack size (after)
height	Push element	S height
S height	Pop element	height
S height	Inspect top element	S height

You'll express the preconditions and postconditions in the types of each operation. Once you've defined the operations on a stack, you'll implement a function to run sequences of stack operations using a concrete representation of a stack with its height in its type. Because you're using the stack's height in the state transitions, a good concrete representation of a stack is a `Vect`. You know, for example, that a stack of `Integer` of height `10`, contains exactly 10 integers, so you can represent this as a value of type `Vect 10 Integer`.

Finally, you'll see an example of a stack in action, implementing a stack-based interactive calculator.

13.2.1 Representing stack operations in a state machine

As with `DoorCmd` and `MachineCmd` in section 13.1, we'll describe operations on a stack in a dependent type and put the important properties of the input and output states explicitly in the type. Here, the property of the state that interests us is the height of the stack.

Listing 13.7 shows how you can express the operations in table 13.2 in code, describing how each operation affects the height of the stack. For this example, you'll only store `Integer` values on the stack, but you could extend `StackCmd` to allow generic stacks by parameterizing over the element type in the stack.

Listing 13.7 Representing operations on a stack data structure with the input and output heights of the stack in the type (Stack.idr)

You'll use a Vect to represent the stack, so import Data.Vect here.

Push increases the height of the stack by 1.

```
import Data.Vect

data StackCmd : Type -> Nat -> Nat -> Type where
     Push : Integer -> StackCmd () height (S height)
     Pop : StackCmd Integer (S height) height
     Top : StackCmd Integer (S height) (S height)

     Pure : ty -> StackCmd ty height height
     (>>=) : StackCmd a height1 height2 ->
                (a -> StackCmd b height2 height3) ->
                StackCmd b height1 height3
```

Pop requires there to be at least one element on the stack, and it decreases the height of the stack by 1.

Top requires there to be at least one element on the stack, and it preserves the height of the stack.

You're using a `Vect` to represent the stack, so every time you add an element to the vector or remove an element, you'll change the vector's type. You're therefore representing dependently typed mutable state by putting the relevant arguments to the type (the length of the `Vect`) in the `StateCmd` type itself.

Using `StackCmd`, you can write sequences of stack operations where the input and output heights of the stack are explicit in the types. For example, the following function pushes two integers, pops two integers, and then returns their sum:

```
testAdd : StackCmd Integer 0 0
testAdd = do Push 10
             Push 20
             val1 <- Pop
             val2 <- Pop
             Pure (val1 + val2)
```

The types of the constructors in `StackCmd` ensure that there will always be an element on the stack when you try to `Pop`. For example, if you only push one integer in `testAdd`, Idris will report an error:

```
testAdd : StackCmd Integer 0 0
testAdd = do Push 10
             val1 <- Pop
             val2 <- Pop    <- There's only one element on the
                               stack, so Pop doesn't type-check.
             Pure (val1 + val2)
```

When you try to define `testAdd` like this, Idris reports an error:

```
Stack.idr:27:22:
When checking right hand side of testAdd with expected type
        StackCmd Integer 0 0

When checking an application of constructor Main.>>=:
        Type mismatch between
                StackCmd Integer (S height) height (Type of Pop)
        and
                StackCmd a 0 height2 (Expected type)

        Specifically:
                Type mismatch between
                        S height
                and
                        0
```

This error, and particularly the mismatch between `S height` and `0`, means that you have a stack of height `0`, but `Pop` needs a stack that contains at least one element.

This approach is similar to the stateful functions defined in chapter 12, here using `Push` and `Pop` to describe how you're modifying and querying the state. As with the earlier descriptions of sequences of stateful operations, you'll need to write a separate function to run those sequences.

13.2.2 Implementing the stack using Vect

Listing 13.8 shows how to implement a function that executes stack operations. This is similar to `runState`, which you saw in chapter 12, but here you take an input `Vect` of the correct height as the contents of the stack.

Listing 13.8 Executing a sequence of actions on a stack, using a `Vect` to represent the stack's contents

```
runStack : (stk : Vect inHeight Integer) ->
           StackCmd ty inHeight outHeight ->
           (ty, Vect outHeight Integer)
runStack stk (Push val) = ((), val :: stk)
runStack (val :: stk) Pop = (val, stk)
runStack (val :: stk) Top = (val, val :: stk)

runStack stk (Pure x) = (x, stk)
runStack stk (cmd >>= next)
    = let (cmdRes, newStk) = runStack stk cmd in
          runStack newStk (next cmdRes)
```

The length of the input vector is the input height of the stack.

The length of the output vector is the output height of the stack.

The length of the output vector is the output height of the stack.

If you try `runStack` with `testAdd`, passing it an initial empty stack, you'll see that it returns the sum of the two elements that you push, and that the final stack is empty:

```
*Stack> runStack [] testAdd
(30, []) : (Integer, Vect 0 Integer)
```

You can also define functions like the following, which adds the top two elements on the stack, putting the result back onto the stack:

```
doAdd : StackCmd () (S (S height)) (S height)
doAdd = do val1 <- Pop
           val2 <- Pop
           Push (val1 + val2)
```

The input state `S (S height)` means that the stack must have at least two elements on it, but, otherwise, it could be any height. If you try executing `doAdd` with an initial stack containing two elements, you'll see that it results in a stack containing a single element that's the sum of the two input elements:

```
*Stack> runStack [2,3] doAdd
((), [5]) : ((), Vect 1 Integer)
```

If the input state contains more than two elements, you'll see that it results in a stack with a height one smaller than the input height. For example, an input stack of `[2, 3, 4]` results in an output stack with the value `[2 + 3, 4]`:

```
*Stack> runStack [2,3,4] doAdd
((), [5, 4]) : ((), Vect 2 Integer)
```

You can add the two elements on the resulting stack with another call to `doAdd`:

```
*Stack> runStack [2,3,4] (do doAdd; doAdd)
((), [9]) : ((), Vect 1 Integer)
```

But trying one more `doAdd` would result in a type error, because there's only one element left on the stack:

```
*Stack> runStack [2,3,4] (do doAdd; doAdd; doAdd)
(input):1:34:When checking an application of constructor Main.>>=:
        Type mismatch between
                StackCmd () (S (S height)) (S height) (Type of doAdd)
        and
                StackCmd ty 1 outHeight (Expected type)

        Specifically:
                Type mismatch between
                        S height
                and
                        0
```

This error means that you needed `S (S height)` elements on the stack (that is, at least two elements) but you only had `S height` (that is, at least one, but not necessarily any more). By putting the height of the stack in the type, therefore, you've explicitly specified the preconditions and postconditions on each operation, so you get a type error if you violate any of these.

13.2.3 Using a stack interactively: a stack-based calculator

If you add commands for reading from and writing to the console, you can write a console application for manipulating the stack and implement a stack-based calculator. A user can either enter a number, which pushes the number onto the stack, or add, which adds the top two stack items, pushes the result onto the stack, and displays the result. A typical session might go as follows:

```
*StackIO> :exec
> 3
> 4
> 5

> add
9
> add
12
> add
Fewer than two items on the stack
```

User pushes three values onto the stack: 3, 4 and 5 (annotation on `> 3`, `> 4`, `> 5`)

Adds the top two stack items, displays and pushes the result (annotation on the two `> add` lines)

Error, because there's only one item (12) on the (annotation on the final `> add`)

Figure 13.3 shows how each of the valid inputs in this session affects the contents of the stack. Every time the user enters an integer, the stack size grows by one, and every time the user enters `add`, the stack size decreases by one, as long as there are two items to add.

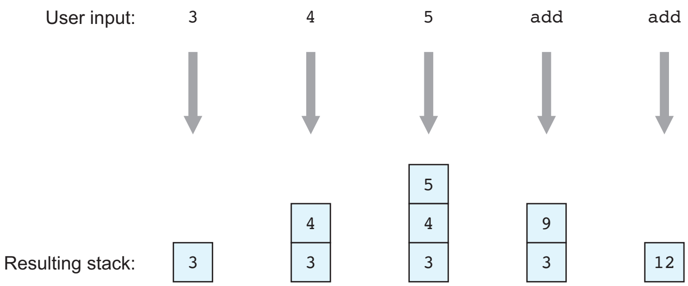

Figure 13.3 How each user input affects the contents of the stack

To implement this interactive stack program, you'll need to extend `StackCmd` to support reading from and writing to the console. The following listing shows `StackCmd` in a new file, StackIO.idr, extended with two commands: `GetStr` and `PutStr`.

Listing 13.9 Extending `StackCmd` to support console I/O with the commands `GetStr` and `PutStr` (StackIO.idr)

```
data StackCmd : Type -> Nat -> Nat -> Type where
     Push : Integer -> StackCmd () height (S height)
     Pop : StackCmd Integer (S height) height
     Top : StackCmd Integer (S height) (S height)

     GetStr : StackCmd String height height
     PutStr : String -> StackCmd () height height

     Pure : ty -> StackCmd ty height height
     (>>=) : StackCmd a height1 height2 ->
             (a -> StackCmd b height2 height3) ->
             StackCmd b height1 height3
```

Neither GetStr nor PutStr use the stack, so the height remains the same.

DEPENDENT STATES IN THE EFFECTS LIBRARY I mentioned the Effects library in chapter 12, which allows you to combine effects like state and console I/O without having to define a new type, like `StackCmd` here. The Effects library supports descriptions of state transitions and dependent state as in `StackCmd`. I won't describe the Effects library further in this book, but learning about the principles of dependent state here will mean that you'll be able to learn how to use the more flexible Effects library more readily.

You'll also need to update `runStack` to support the two new commands. Because `GetStr` and `PutStr` describe interactive actions, you'll need to update the type of `runStack` to return `IO` actions. Here's the updated `runStack`.

Listing 13.10 Updating `runStack` to support the interactive commands `GetStr` and `PutStr` (StackIO.idr)

```
runStack : (stk : Vect inHeight Integer) ->
           StackCmd ty inHeight outHeight ->
           IO (ty, Vect outHeight Integer)
runStack stk (Push val) = pure ((), val :: stk)
```

```
runStack (val :: stk) Pop = pure (val, stk)
runStack (val :: stk) Top = pure (val, val :: stk)
runStack stk GetStr = do x <- getLine
                         pure (x, stk)
runStack stk (PutStr x) = do putStr x
                             pure ((), stk)
runStack stk (Pure x) = pure (x, stk)
runStack stk (x >>= f) = do (x', newStk) <- runStack stk x
                            runStack newStk (f x')
```

As with the vending machine, you'll describe infinite sequences of `StackCmd` operations in total functions by defining a separate `StackIO` type for describing infinite streams of stack operations. The following listing shows how you can define `StackIO` and how to run `StackIO` sequences, given an initial state for the stack.

Listing 13.11 Defining infinite sequences of interactive stack operations (StackIO.idr)

```
data StackIO : Nat -> Type where
     Do : StackCmd a height1 height2 ->
          (a -> Inf (StackIO height2)) -> StackIO height1

namespace StackDo
     (>>=) : StackCmd a height1 height2 ->
             (a -> Inf (StackIO height2)) -> StackIO height1
     (>>=) = Do

data Fuel = Dry | More (Lazy Fuel)

partial
forever : Fuel
forever = More forever

run : Fuel -> Vect height Integer -> StackIO height -> IO ()
run (More fuel) stk (Do c f)
     = do (res, newStk) <- runStack stk c
          run fuel newStk (f res)
run Dry stk p = pure ()
```

- The Nat argument is the initial height of the stack for the infinite sequence. (annotates `data StackIO : Nat -> Type where`)
- Supports do notation for StackIO (annotates `namespace StackDo`)
- forever allows you to run a total program indefinitely by giving an infinite supply of Fuel. See chapter 11 for the full details. (annotates `data Fuel` and `forever : Fuel`)
- The input Vect must have a number of items given by the initial stack height. (annotates `run : Fuel -> ...`)

The interactive calculator follows a similar pattern to the implementation of the vending machine. The next listing shows an outline of the main loop, which reads an input, parses it into a command type, and processes the command if the input is valid.

Listing 13.12 Outline of an interactive stack-based calculator (StackIO.idr)

```
data StkInput = Number Integer
              | Add

strToInput : String -> Maybe StkInput

mutual
  tryAdd : StackIO height

  stackCalc : StackIO height
  stackCalc = do PutStr "> "
                 input <- GetStr
```

- Describes possible user inputs: entering a number or the add (annotates `data StkInput`)
- Parses the input read from the console. Returns Maybe because input could be invalid. (annotates `strToInput`)
- Adds two numbers at the top of the stack, if present, and then loops (annotates `tryAdd`)
- Main loop of the interactive calculator (annotates `stackCalc`)

```
                case strToInput input of
                     Nothing => do PutStr "Invalid input\n"
                                   stackCalc
                     Just (Number x) => do Push x
                                           stackCalc
                     Just Add => tryAdd

main : IO ()
main = run forever [] stackCalc
```

You still need to define `strToInput`, which parses user input, and `tryAdd`, which adds the two elements on the top of the stack, if possible. The following listing shows the definition of `strToInput`.

Listing 13.13 Reading user input for the stack-based calculator (StackIO.idr)

```
strToInput : String -> Maybe RPNInput
strToInput "" = Nothing
strToInput "add" = Just Add
strToInput x = if all isDigit (unpack x)
                  then Just (Number (cast x))
                  else Nothing
```

Empty input is considered invalid. (annotates `strToInput "" = Nothing`)

If the input is the string "add", parse as the Add command.

If the input consists entirely of digits, parse as Number.

Finally, the next listing shows the definition of `tryAdd`. Like `vend` and `refill` in the vending machine implementation, you need to match on the initial state to make sure that there are enough items on the stack to add.

Listing 13.14 Adding the top two elements on the stack, if they're present (StackIO.idr)

Adding is only valid if there are at least two elements on the stack.

doAdd, defined earlier, has a precondition in its type that there are two elements on the stack.

```
tryAdd : StackIO height
tryAdd {height = (S (S h))}
    = do doAdd
         result <- Top
         PutStr (show result ++ "\n")
         stackCalc
tryAdd
    = do PutStr "Fewer than two items on the stack\n"
         stackCalc
```

Inspects the top item on the stack so that you can display it as the result

If the earlier case doesn't match, there aren't enough items on the stack to add.

Continues with the main loop

You can check that `stackCalc` is total at the REPL:

```
*StackIO> :total stackCalc
Main.stackCalc is Total
```

By separating the looping component (`StackIO`) from the terminating component (`StackCmd`), and by giving precise types to the operations, you can be sure that `stackCalc` has at least the following properties, as long as it's total:

- It will continue running indefinitely.
- It will never crash due to user input that isn't handled.
- It will never crash due to a stack overflow.

Exercises

1 Add user commands to the stack-based calculator for `subtract` and `multiply`. You can test these as follows:

```
*ex_13_2> :exec
> 5
> 3
> subtract
2
> 8
> multiply
16
```

2 Add a `negate` user command to the stack-based calculator for negating the top item on the stack. You can test this as follows:

```
> 10
> negate
-10
```

3 Add a `discard` user command that removes the top item from the stack. You can test this as follows:

```
> 3
> 4
> discard
Discarded 4
> add
Fewer than two items on the stack
```

4 Add a `duplicate` user command that duplicates the top item on the stack. You can test this as follows:

```
> 2
> duplicate
Duplicated 2
> add
4
```

13.3 Summary

- Data types can model state machines by using each data constructor to describe a state transition.
- You can describe how a command changes the state of a system by giving the input and output states of the system as part of the command's type.
- Developing sequences of state transitions interactively, using holes, means you can check the required input and output states of a sequence of commands.

- Types can model infinite state spaces as well as finite states.
- Sequences of commands give verified sequences of state transitions because a sequence of commands will only type-check if it describes a valid sequence of state transitions.
- You can represent mutable dependently typed state by putting the arguments to the dependent type in the state transitions. For example, you can use the length of a vector to represent the height of a stack.