# The Transit library

Tutorial: Build fast and type-safe state machines with PureScript

Michael Bock

January 2026

# Contents

# Introduction

**Transit** is a PureScript library for building type-safe state machines. It provides a type-level DSL for specifying state transitions. You define your state machine once using this specification, and the compiler ensures your implementation matches it — eliminating bugs from invalid transitions, missing cases, or documentation drift.

This tutorial will guide you through the basics of **Transit** by showing its main features based on examples. We'll start with a simple door state machine and gradually add more complexity.

### About This Documentation

All code examples in this documentation are extracted from actual, type-checked PureScript source files. Also, whenever you find an assertion or a full unit test, we ensure that it ran and passed. In this sense, this text is not just documentation, but also a test suite. At the bottom of every code example you can find a link to the actual source file, so you can get a better picture of the context and see information about the imports used.

### Similarities to Haskell's Servant library

If you're familiar with Servant[1] from Haskell, **Transit** follows a similar philosophy: just as Servant uses a REST API type-level specification to ensure type-safe routing functions and generate OpenAPI documentation, **Transit** uses a state machine graph type-level specification to ensure type-safe update functions and generate state diagrams.

# Example 1: A Simple Door

Full source code: *test/Examples/Door.purs*

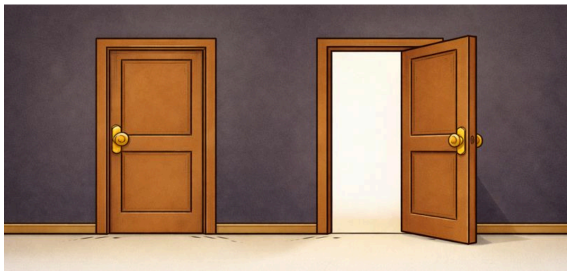

Figure 1: Open and Closed Door

Let's start with a simple door state machine to demonstrate **Transit**'s core concepts. This example will show you how to define a state machine using **Transit**'s type-level DSL, implement a type-safe update function, and generate documentation automatically. We'll compare the traditional approach with **Transit**'s approach to highlight the benefits of the latter.

Think of a door that can be either open or closed. When it's open, you can close it. When it's closed, you can open it. That's it — no other actions make sense. You can't open a door that's already open, and you can't close a door that's already closed. This simple behavior is what we're modeling in this example.

## The State Machine

Before diving into the code, let's visualize our simple door state machine. This will help you understand the structure we're about to implement.

### State Diagram

The state diagram below shows all possible states and the valid transitions between them:

In this diagram, you can see:

- **Two states**: `DoorOpen` and `DoorClosed` (shown as rectangles)
- **Two transitions**: `Close` and `Open`
- **Arrows**: The direction of each arrow shows which state changes are valid

[1]Servant is a Haskell library for building type-safe web APIs.

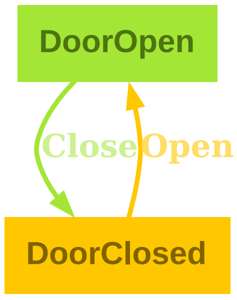

Figure 2: Simple Door state diagram

### Transition Table

For a more structured view, here's the corresponding transition table:

| State | | Message | | State |
|---|---|---|---|---|
| DoorOpen | $\longrightarrow$ | Close | $\longrightarrow$ | DoorClosed |
| DoorClosed | $\longrightarrow$ | Open | $\longrightarrow$ | DoorOpen |

Each row shows one valid transition: which state you start in, which action you take, and which state you end up in. Notice that invalid actions — like trying to open an already open door — simply don't appear in the table.

Now let's see how we represent this in PureScript code.

## Classic Approach

Before diving into **Transit**, let's first look at how state machines are typically implemented in PureScript using State and Message data types and an update function performing pattern matching on both. This classic approach is familiar to most PureScript developers and serves as a baseline for understanding what **Transit** improves upon.

### States and Message types

To represent our door in code, we need two major types: the states the door can be in, and the actions that can change those states. In PureScript, we define these as simple data types.

```
data State
  = DoorOpen
  | DoorClosed

data Msg
  = Close
  | Open
```

test/Examples/Classic/Door.purs (lines 14-20)

The `State` type captures the two possible states we saw in the diagram: `DoorOpen` and `DoorClosed`. The `Msg` type represents the two actions: `Close` and `Open`. These correspond directly to what we visualized earlier — each state and each transition from the diagram has a corresponding value in these types.

### The update function

Now that we have our types, we need a function that takes the current state and a message, and returns the new state. The traditional way to implement this is with a pattern-matching function:

```
update :: State -> Msg -> State
update state msg =
  case state, msg of
    DoorOpen, Close -> DoorClosed
    DoorClosed, Open -> DoorOpen
    _, _ -> state
```

test/Examples/Classic/Door.purs (lines 22-27)

We pattern match on both the current state and the message at once. It could also be written as a nested pattern match. The `update` function handles the two valid transitions we saw in the diagram: closing an open door and opening a closed door. The catch-all case `_, _ -> state` handles any invalid combinations (like trying to open an already open door) by returning the current state unchanged.

While this approach works and is straightforward, it has some drawbacks:

- **Implicit state machine specification**: The state machine's structure is only defined implicitly within the update function's pattern matching and return values.
- **Documentation drift**: If you maintain a state diagram for documentation purposes, there's nothing ensuring the code stays in sync — you have to remember to update both manually.
- **Limited analysis capabilities**: There's no way to analyze the state machine's structure or behavior statically — you can only inspect its behavior by running the code.

## Implementation using Transit

Now let's see how **Transit** can help us to improve this.

### State and Message Types

**Transit** uses `Variant`[2] types for both `State` and `Msg` instead of traditional ADTs. This design choice is crucial for **Transit**, but for now let's just focus on the fact that it's another way to represent sum types.[3]

```
type State = Variant
  ( "DoorOpen" :: {}
  , "DoorClosed" :: {}
  )

type Msg = Variant
  ( "Close" :: {}
  , "Open" :: {}
  )
```

test/Examples/Door.purs (lines 27-35)

The empty record `{}` is used to represent the absence of any data (payload) associated with the state or message.

### Transit Specification

Once the types are defined, we can define the state machine structure using **Transit**'s type-level DSL. Let's see what it looks like:

[2]The `purescript-variant` library provides row-polymorphic sum types. See the documentation for more details.

[3]This design choice is crucial for **Transit**'s type-level machinery. The key advantage is that **Transit** can filter the possible cases (both input states/messages and output states) for each handler function. Variants are perfect for this. There is no way to express a subset of cases from a traditional ADT.

```
type DoorTransit =
  Transit
    :* ("DoorOpen" :@ "Close" >| "DoorClosed")
    :* ("DoorClosed" :@ "Open" >| "DoorOpen")
```

test/Examples/Door.purs (lines 37-40)

Breaking down the syntax:

- `Transit` initializes an empty transition list
- `:*` is an infix operator that appends each transition to the list
- The `:@` operator connects a state to a message, and `>|` indicates the target state

For instance, we read the first transition as: in state `DoorOpen`, when receiving message `Close`, transition to state `DoorClosed`.

This type-level specification fully defines the state machine's structure. The compiler can now use it to ensure our implementation of the update function matches the specification.

### The Update Function

Based on the above specification, we create an update function using `mkUpdate`:

```
update :: State -> Msg -> State
update = mkUpdate @DoorTransit
  (match @"DoorOpen" @"Close" \_ _ -> return @"DoorClosed")
  (match @"DoorClosed" @"Open" \_ _ -> return @"DoorOpen")
```

test/Examples/Door.purs (lines 42-45)

Here's how this works:

- `mkUpdate @DoorTransit` creates an update function based on the `DoorTransit` specification. The `@` symbol is type application[4], passing the specification to the function.
- Each `match` line handles one transition from the specification. The first two arguments (`@"DoorOpen"` and `@"Close"`) are type-level symbols (type applications) that specify which state and message to match on. The lambda function defines what happens when that transition occurs.
- `return @"DoorClosed"` specifies which state to transition to. The `return` function is part of **Transit**'s DSL for specifying the target state, and the `@` symbol again indicates a type-level symbol.
- **Important**: The order of match handlers must match the order of transitions in the DSL specification.

## Testing the update function

Before we proceed, let's verify that our implementation of the update function works as we expect it to. We'll do this by writing some tests.

### Creating Variant Values

For the tests we need to create `Variant` values. To create values of type `Variant`, **Transit** provides the `v` function from `Transit.VariantUtils`. [5]

[4]In PureScript, the `@` symbol is used for explicit type application, allowing you to pass type-level arguments to functions.

[5]It's a convenience wrapper around `Variant`'s `inj` function that uses type application (no Proxy needed) and allows omitting empty record arguments.

```
doorOpen :: State
doorOpen = v @"DoorOpen" {}

doorClosed :: State
doorClosed = v @"DoorClosed" {}

close :: Msg
close = v @"Close" {}

open :: Msg
open = v @"Open" {}
```

Since having no data associated with a case is very common, the `v` function has a shortcut for this: You can just omit the empty record argument:

```
doorOpenShort :: State
doorOpenShort = v @"DoorOpen"
```

Now we are well prepared to start testing the update function that we've implemented previously.

### Testing State Transitions

To test our update function, we'll use two functions from the `Data.Array` module:

- `foldl :: forall a b. (b -> a -> b) -> b -> Array a -> b`
- `scanl :: forall a b. (b -> a -> b) -> b -> Array a -> Array b`

The simplest way to test the update function is to use `foldl` to apply a sequence of messages and check if the final state matches what we expect:

```
specWalk1 :: Spec Unit
specWalk1 =
  it "follows the walk and ends in expected final state" do
    foldl update (v @"DoorOpen") [ v @"Close", v @"Open", v @"Close" ]
      `shouldEqual`
        (v @"DoorClosed")
```

test/Examples/Door.purs (lines 54-59)

This test starts with the door open, closes it, opens it, then closes it again. It checks that we end up with the door closed, as expected.

This test only checks the final result. To be more thorough, we should also verify that each step along the way works correctly. The `scanl` function is perfect for this — it shows us all the intermediate states, not just the final one.

```
specWalk2 :: Spec Unit
specWalk2 =
  it "follows the walk and visits the expected intermediate states" do
    scanl update (v @"DoorOpen") [ v @"Close", v @"Open", v @"Close" ]
      `shouldEqual`
        [ v @"DoorClosed", v @"DoorOpen", v @"DoorClosed" ]
```

test/Examples/Door.purs (lines 61-66)

This test is similar to the previous one. But instead of just checking the final result, it verifies each step along the way:

after closing, the door is closed; after opening, the door is open; and after closing again, the door remains closed. This makes sure each transition works correctly.

#### More ergonomic testing with `assertWalk` helper function

Since we'll want to write more of these tests for further examples, it's helpful to define a reusable helper function. The `assertWalk` function takes an update function, an initial state, and a list of message/state pairs representing the expected walk through the state machine:

```
assertWalk
  :: forall msg state
   . Eq state
  => Show state
  => (state -> msg -> state)
  -> state
  -> Array (msg /\ state)
  -> Aff Unit
assertWalk updateFn initState walk = do
  let
    msgs :: Array msg
    msgs = map fst walk

    expectedStates :: Array state
    expectedStates = map snd walk

    actualStates :: Array state
    actualStates = scanl updateFn initState msgs

  actualStates `shouldEqual` expectedStates
```

test/Examples/Common.purs (lines 40-59)

The function extracts the messages from the pairs, applies them sequentially using `scanl`, and verifies that the resulting states match the expected ones. Here's how we use it:

```
specWalk3 :: Spec Unit
specWalk3 =
  it "follows the walk and visits the expected intermediate states" do
    assertWalk update
      (v @"DoorOpen")
      [ v @"Close" ~> v @"DoorClosed"
      , v @"Open" ~> v @"DoorOpen"
      , v @"Close" ~> v @"DoorClosed"
      , v @"Close" ~> v @"DoorClosed"
      , v @"Open" ~> v @"DoorOpen"
      , v @"Open" ~> v @"DoorOpen"
      , v @"Open" ~> v @"DoorOpen"
      ]
```

test/Examples/Door.purs (lines 68-80)

The `~>` operator is an infix alias for `Tuple`. So `v @"Close" ~> v @"DoorClosed"` is equivalent to `Tuple (v @"Close") (v @"DoorClosed")`.

We read it like: Starting from state `DoorOpen`, when receiving message `Close`, we expect the next state to be `DoorClosed`. From there, when receiving message `Open`, we expect the next state to be `DoorOpen`. And so on.

## Generating Documentation

**Transit** can generate both state diagrams and transition tables directly from your type-level specification. Both generation processes use the same approach: `reflectType` converts your type-level DSL specification to a term-level equivalent, which can then be used to generate the documentation.

```
doorTransit :: TransitCore
doorTransit = reflectType (Proxy @DoorTransit)
```

test/Examples/Door.purs (lines 47-48)

### State Diagrams

**Transit** can generate state diagrams using Graphviz[6]. For this we'll use the following function from the `Transit.Render.Graphviz` module:

`generate :: TransitCore -> (Options -> Options) -> GraphvizGraph`

It takes the `TransitCore` value which we've created in the previous step and a function that takes the default options and returns the options we want to use. Now we have everything in place to generate the state diagram:

```
generateGraphDark :: Effect Unit
generateGraphDark =
  let
    graph :: GraphvizGraph
    graph = TransitGraphviz.generate doorTransit \def -> def
      { theme = themeHarmonyDark
      , layout = Portrait
      }
  in
    FS.writeTextFile UTF8 "renders/door_graph-dark.dot" (Graphviz.toDotStr graph)
```

test/Examples/Door.purs (lines 104-113)

**Generated Output**: This code produces the diagram we examined at the beginning of this example.

Link: View diagram on GraphvizOnline

The options we're using above are:

- The `theme` option we're using above controls the color scheme. **Transit** provides a couple of built-in themes, but you can also provide your own. See themes.md for more details.
- The `layout` option controls the layout of the graph. We're using `Portrait` here, which is the default layout. But you can also use `Landscape`, `Circular`, or `Manual` to position the nodes manually, as we'll see later.

Finally, to convert the `.dot` file to an SVG (or other formats), use the Graphviz command-line tools:

```
dot -Tsvg renders/door_graph.dot -o renders/door.svg
```

### Transition Tables

In addition to state diagrams, you can also generate transition tables from the same specification. This provides a tabular view of all state transitions, which can be easier to read for some use cases.

[6] Graphviz is a graph visualization software that uses the DOT language. The `.dot` files generated by Transit can be rendered into various formats (SVG, PNG, PDF, etc.) using Graphviz's command-line tools.

```
generateTable :: Effect Unit
generateTable = do
  let
    table :: Table
    table = TransitTable.generate doorTransit \def -> def

  FS.writeTextFile UTF8 "renders/door_table.md" (Table.toMarkdown table)
```

test/Examples/Door.purs (lines 115-121)

Here we provide no options to the `generate` function, so we use the identity function `\def -> def` to pass the default options.

**Generated Output**: This generates the transition table shown earlier, with rows for each valid state transition.

### Conclusion

In this example, we've seen how **Transit** helps you build type-safe state machines. We started with a simple door that can be open or closed, and learned the core workflow:

1. Use `Variant` types for both `State` and `Msg` instead of traditional ADTs.
2. **Define the state machine** using **Transit**'s type-level DSL specification
3. **Implement the update function** using `mkUpdate` with `match` clauses that the compiler verifies against the specification
4. **Generate documentation** automatically — both state diagrams and transition tables — from the same specification

The key advantage is that your specification, implementation, and documentation all stay in sync because they share the same source of truth. The compiler ensures your code matches your specification, and your documentation is generated directly from it.

While this example was simple, it demonstrates **Transit**'s fundamental approach. In the next example, we'll see how **Transit** handles more complex scenarios with states that contain data and conditional transitions.

## Example 2: Door with Pin

Full source code: *test/Examples/DoorPin.purs*

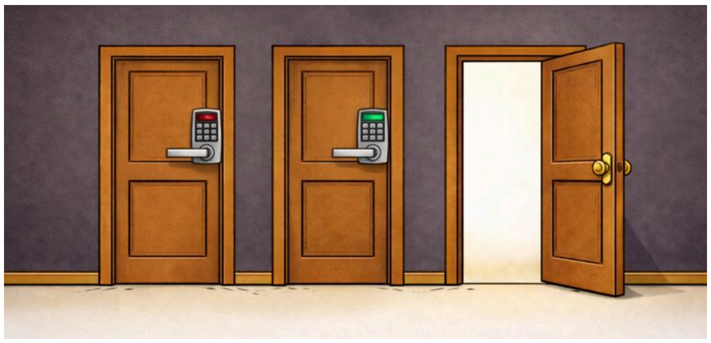

Figure 3: Door with Pin

Now let's extend our door to support PIN-based locking. In this enhanced version, you can lock the door with a PIN code, and then only unlock it by entering the correct PIN. This introduces two important concepts: **states with data** and **conditional transitions**.

## The State Machine

We add a new state `DoorLocked` and the new messages `Lock` and `Unlock`:

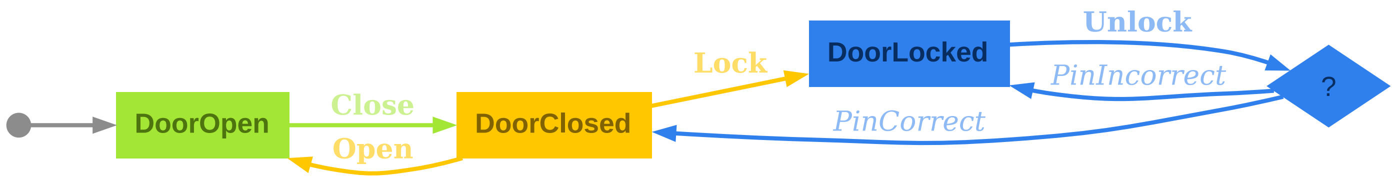

Figure 4: Door with Pin state diagram

Notice the diamond node in the state diagram — this represents a conditional transition where the outcome depends on runtime data: The unlock operation can succeed (transitioning to `DoorClosed`) if the condition `PinCorrect` is met - or fail (staying in `DoorLocked`) when the condition `PinIncorrect` is met.

In the transition table the conditional transitions are expressed by the new "Guard" column. For most transitions however, this column is empty — these are unconditional transitions that always succeed.

| State | | Message | | Guard | | State |
|---|---|---|---|---|---|---|
| DoorOpen | ⟶ | Close | | | ⟶ | DoorClosed |
| DoorClosed | ⟶ | Open | | | ⟶ | DoorOpen |
| DoorClosed | ⟶ | Lock | | | ⟶ | DoorLocked |
| DoorLocked | ⟶ | Unlock | ? | PinIncorrect | ⟶ | DoorLocked |
| DoorLocked | ⟶ | Unlock | ? | PinCorrect | ⟶ | DoorClosed |

Guard labels are not strictly required. Transitions can have multiple target states without explicit labels - like in the Countdown example. But the labels can be very useful to make the code and the state diagram more readable.

## The Classic Approach

Let's briefly recap how we would implement this using the classic approach.

### States and Message types

The PureScript types now include data in both states and messages:

```
data State
  = DoorOpen
  | DoorClosed
  | DoorLocked { storedPin :: String }

data Msg
  = Close
  | Open
  | Lock { newPin :: String }
  | Unlock { enteredPin :: String }
```

test/Examples/Classic/DoorPin.purs (lines 10-27)

### The update function

Accordingly the update function now needs to handle state and message data:

```
update :: State -> Msg -> State
update state msg = case state, msg of
```

```
    DoorOpen, Close -> DoorClosed
    DoorClosed, Open -> DoorOpen
    DoorClosed, Lock { newPin } -> DoorLocked { storedPin: newPin }
    DoorLocked { storedPin }, Unlock { enteredPin } ->
      if storedPin == enteredPin then
        DoorClosed
      else
        DoorLocked { storedPin }
    _, _ -> state
```

test/Examples/Classic/DoorPin.purs (lines 29-39)

### Implementation using Transit

#### State and Message Types

Also in the **Transit** approach we define `State` and `Msg` types. This time some cases of those types have data attached to them:

```
  type State = Variant
    ( "DoorOpen" :: {}
    , "DoorClosed" :: {}
    , "DoorLocked" :: { storedPin :: String }
    )

  type Msg = Variant
    ( "Close" :: {}
    , "Open" :: {}
    , "Lock" :: { newPin :: String }
    , "Unlock" :: { enteredPin :: String }
    )
```

test/Examples/DoorPin.purs (lines 31-42)

#### Transit Specification

In the DSL specification, we express conditional transitions by listing multiple possible target states:

```
  type DoorPinTransit =
    Transit
      :* ("DoorOpen" :@ "Close" >| "DoorClosed")
      :* ("DoorClosed" :@ "Open" >| "DoorOpen")
      :* ("DoorClosed" :@ "Lock" >| "DoorLocked")
      :*
        ( "DoorLocked" :@ "Unlock"
            >| ("PinCorrect" :? "DoorClosed")
            >| ("PinIncorrect" :? "DoorLocked")
        )
```

test/Examples/DoorPin.purs (lines 44-53)

The syntax `("PinCorrect" :? "DoorClosed") >| ("PinIncorrect" :? "DoorLocked")` indicates that the `Unlock` message from `DoorLocked` can transition to either state, depending on runtime conditions. The `:?` operator associates a condition label (like `"PinCorrect"`) with a target state, and `>|` chains multiple conditional outcomes together.

**The Update Function**

The handlers in the update function now have access to both the matching state and message data, allowing you to implement the conditional runtime logic for the transition.

```
update :: State -> Msg -> State
update = mkUpdate @DoorPinTransit
  ( match @"DoorOpen" @"Close" \_ _ ->
      return @"DoorClosed"
  )
  ( match @"DoorClosed" @"Open" \_ _ ->
      return @"DoorOpen"
  )
  ( match @"DoorClosed" @"Lock" \_ msg ->
      return @"DoorLocked" { storedPin: msg.newPin }
  )
  ( match @"DoorLocked" @"Unlock" \state msg ->
      let
        isCorrect = state.storedPin == msg.enteredPin
      in
        if isCorrect then
          returnVia @"PinCorrect" @"DoorClosed"
        else
          returnVia @"PinIncorrect" @"DoorLocked" { storedPin: state.storedPin }
  )
```

test/Examples/DoorPin.purs (lines 55-74)

The order of match handlers in `mkUpdate` must match the order of transitions in the DSL specification. The compiler *can* detect if the returned state of a handler is legal for a given transition. However, it *cannot* detect if an implementation forgets to return a possible case. For example, if a transition can return either `DoorClosed` or `DoorLocked`, but your handler always returns `DoorClosed`, the compiler would not detect this error. The compiler cannot verify whether your handler implements the conditional logic correctly, so missing a case is just one of many possible errors.

## Testing the update function

We'll use the same test function which we used in the previous example. Let's recap how it works quickly by looking at its type signature:

```
assertWalk
  :: forall msg state
   . Eq state
  => Show state
  => (state -> msg -> state)
  -> state
  -> Array (msg /\ state)
  -> Aff Unit
```

We want to start the state machine in the `DoorOpen` state and then follow this sequence of transitions:

1. `Close` the door, expect transition to `DoorClosed`
2. `Lock` the door with PIN "1234", expect transition to `DoorLocked` with the stored PIN
3. Attempt to `Unlock` with the wrong PIN "abcd", expect to stay in `DoorLocked` with the original PIN
4. `Unlock` with the correct PIN "1234", expect transition to `DoorClosed`
5. `Open` the door, expect transition to `DoorOpen`

In code this looks like this:

```
specWalk :: Spec Unit
specWalk =
  it "should follow the walk and visit the expected intermediate states" do
    assertWalk update
      (v @"DoorOpen")
      [ v @"Close" ~> v @"DoorClosed"
      , v @"Lock" { newPin: "1234" } ~> v @"DoorLocked" { storedPin: "1234" }
      , v @"Unlock" { enteredPin: "abcd" } ~> v @"DoorLocked" { storedPin: "1234" }
      , v @"Unlock" { enteredPin: "1234" } ~> v @"DoorClosed"
      , v @"Open" ~> v @"DoorOpen"
      ]
```

test/Examples/DoorPin.purs (lines 83-93)

Since this test passes, we can be pretty confident that the update function is correct.

## Generating Documentation

For generating the state diagram, we now add some more options to the `generate` function:

- `entryPoints`: The state machine will start in the `DoorOpen` state.
- `layout`: The state diagram will be displayed in landscape mode.

```
generateGraphLight :: Effect Unit
generateGraphLight = do
  let
    graph :: GraphvizGraph
    graph = TransitGraphviz.generate doorPinTransit \cfg -> cfg
      { theme = themeHarmonyLight
      , entryPoints = [ "DoorOpen" ]
      , layout = Landscape
      }

  FS.writeTextFile UTF8 "renders/door-pin_graph-light.dot" (Graphviz.toDotStr graph)
```

test/Examples/DoorPin.purs (lines 103-113)

**Generated Output**: This creates the state diagram we saw earlier in this example.

Link: View diagram on GraphvizOnline

The generation of the transition table works exactly the same as in the previous example.

## Conclusion

This example demonstrates how **Transit** extends beyond simple state machines to handle real-world complexity:

- **States and messages with data**: Both states and messages can carry data (like `storedPin` in `DoorLocked` or `newPin` in `Lock`), and handlers receive this data.
- **Conditional transitions**: The DSL supports transitions with multiple possible outcomes using guard labels (`PinCorrect` and `PinIncorrect`). The type system ensures that conditional transitions can only return valid target states, and each outcome must be associated with its corresponding guard label.

# Example 3: Bridges of Königsberg

Full source code: *test/Examples/BridgesKoenigsberg.purs*

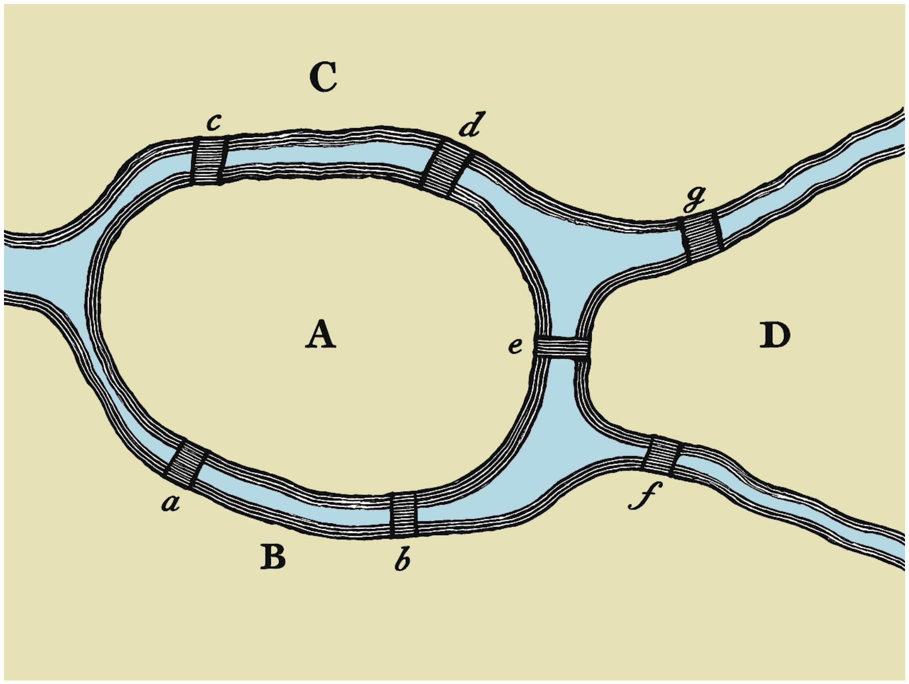

Figure 5: Map of Königsberg

So far, we've seen how **Transit** helps you build type-safe state machines and generate state diagrams and transition tables. But the power of **Transit** extends far beyond documentation generation. The reflected data structure — the term-level representation of your type-level DSL specification — can be converted into a general-purpose graph data structure, enabling sophisticated graph analysis.

This example demonstrates this capability using the famous Seven Bridges of Königsberg problem. In 1736, the mathematician Leonhard Euler was asked whether it was possible to walk through the city of Königsberg crossing each of its seven bridges exactly once.[7]

In the picture above you see the topography of the historic city of Königsberg. The city is divided into four land areas (A, B, C, and D) and seven bridges (a, b, c, d, e, f, and g) connect them.

## The State Machine

While not immediately obvious, the map of the city can be represented as a graph:

- **Nodes** represent the four land areas
- **Edges** represent the seven bridges connecting them

Note that we drew an undirected graph here. This is due to the fact that the bridges are bidirectional. We could also have drawn a directed graph with two edges for each bridge, but this would look more cluttered.

The same is true for the transition table. Instead of two rows for each bridge, we have one row for each bridge:

| State | | Message | | State |
|---|---|---|---|---|
| A | $\longleftarrow$ | a | $\longrightarrow$ | B |
| A | $\longleftarrow$ | b | $\longrightarrow$ | B |
| A | $\longleftarrow$ | c | $\longrightarrow$ | C |
| A | $\longleftarrow$ | d | $\longrightarrow$ | C |
| A | $\longleftarrow$ | e | $\longrightarrow$ | D |
| B | $\longleftarrow$ | f | $\longrightarrow$ | D |
| C | $\longleftarrow$ | g | $\longrightarrow$ | D |

[7]The Seven Bridges of Königsberg problem was solved by Leonhard Euler (1707–1783), a Swiss mathematician, physicist, and engineer who made fundamental contributions to mathematics and physics. His work on this problem is considered the foundation of graph theory.

| State | Message | State |
|---|---|---|

## Implementation using Transit

### State and message types

The state machine represents the four land areas as states (uppercase letters `A`, `B`, `C`, and `D`) and the seven bridges as messages (lowercase letters `a` through `g`). Each message represents crossing a specific bridge, which transitions between the corresponding land areas.

```
type State = Variant
  ( "A" :: {}
  , "B" :: {}
  , "C" :: {}
  , "D" :: {}
  )

type Msg = Variant
  ( "a" :: {}
  , "b" :: {}
  , "c" :: {}
  , "d" :: {}
  , "e" :: {}
  , "f" :: {}
  , "g" :: {}
  )
```

test/Examples/BridgesKoenigsberg.purs (lines 25-40)

### Type-level specification

Since bridges can be crossed in both directions, each bridge creates a bidirectional connection between two land areas. In the type-level specification, we define transitions using the syntax `"State1" |< "Message" >| "State2"`, which effectively defines two transitions: one from `State1` to `State2` and one from `State2` to `State1`.

```
type BridgesTransit =
  Transit
    :* ("A" |< "a" >| "B")
    :* ("A" |< "b" >| "B")
    :* ("A" |< "c" >| "C")
    :* ("A" |< "d" >| "C")
    :* ("A" |< "e" >| "D")
    :* ("B" |< "f" >| "D")
    :* ("C" |< "g" >| "D")
```

test/Examples/BridgesKoenigsberg.purs (lines 42-50)

### The Update Function

However, in the update function we need to explicitly handle both directions of each bridge as shown below.

```
update :: State -> Msg -> State
update = mkUpdate @BridgesTransit
  (match @"A" @"a" \_ _ -> return @"B")
  (match @"B" @"a" \_ _ -> return @"A")
```

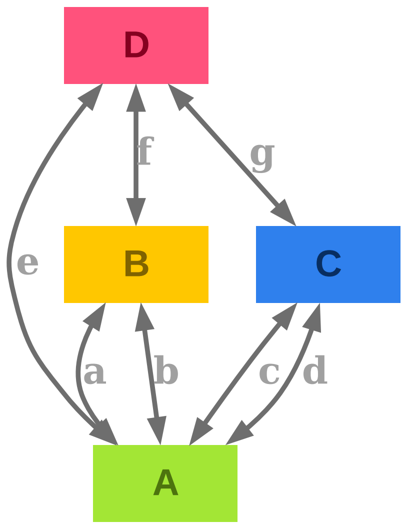

Figure 6: Seven Bridges of Königsberg state diagram

```
    (match @"A" @"b" \_ _ -> return @"B")
    (match @"B" @"b" \_ _ -> return @"A")

    (match @"A" @"c" \_ _ -> return @"C")
    (match @"C" @"c" \_ _ -> return @"A")

  -- And so on ... (13 lines omitted)
```

test/Examples/BridgesKoenigsberg.purs (lines 52-73)

## Testing the update function

The picture shows one randomly chosen walk through the city of Königsberg. Unfortunately, it does not visit all bridges exactly once as required by Euler. The red circle indicates where bridge `g` is crossed twice. Let's test the walk anyway before we move on.

```
specSampleWalk :: Spec Unit
specSampleWalk =
  it "should follow the sample walk and visit the expected intermediate states" do
    assertWalk update
      (v @"A")
      [ v @"a" ~> v @"B"
      , v @"f" ~> v @"D"
      , v @"g" ~> v @"C"
      , v @"c" ~> v @"A"
      , v @"e" ~> v @"D"
      , v @"g" ~> v @"C"
      , v @"d" ~> v @"A"
      , v @"b" ~> v @"B"
      ]
```

test/Examples/BridgesKoenigsberg.purs (lines 82-95)

We could try many other walks the same way. But — spoiler alert — none of them will visit all bridges exactly once. And in the next section we'll see a way to prove this for our state machine.

## Graph Analysis

The real power of **Transit** becomes apparent when we convert the reflected data structure into a general-purpose graph. Using `mkStateGraph`, we transform the **Transit** specification into a `StateGraph` — a specialized `Graph` type configured with edge and node labels suitable for state machine analysis.

```
bridgesGraph :: StateGraph
bridgesGraph = mkStateGraph bridgesTransit
```

test/Examples/BridgesKoenigsberg.purs (lines 97-98)

### Eulerian trail

Once we have this graph data structure, we can perform sophisticated analysis using standard graph algorithms. For the Seven Bridges problem, we want to determine if the graph has an **Eulerian trail**: A path that visits every edge exactly once but doesn't necessarily return to the start.

Let's assume our trail would start and end at the same node. Is there some property that must hold for each node? As we know, a bridge can only be crossed once, so we can conclude that whenever we visit a piece of land via a bridge we must leave it via a *different* bridge again. That means that the number of bridges connected to each land must be 2, or 4, or 6, or 2000, … in other words, an even number.

However, our trail does not have to start and end at the same piece of land. If this is the case, the start and end nodes can have an odd number of bridges connected to them. This is because we never enter the start node, and we never leave the end node.

This is what Euler formalized in his theorem: An undirected graph has an Eulerian trail if and only if it *is connected* and has exactly *zero or two* vertices of odd degree.

### Degree of a node

For simplicity we'll assume that our graphs are always connected. The degree of a node is the number of edges connected to it. Since our graph is undirected, we can obtain the degree of a node by counting the number of outgoing edges:

```
nodeDegree :: StateGraph -> StateNode -> Int
nodeDegree graph node = Set.size (Graph.getOutgoingEdges node graph)
```

test/Examples/Common.purs (lines 21-22)

And we can easily see that all of our nodes have odd degree:

```
specNodeDegree :: Spec Unit
specNodeDegree = do
  it "should each node have the expected degree" do
    nodeDegree bridgesGraph "A" `shouldEqual` 5
    nodeDegree bridgesGraph "B" `shouldEqual` 3
    nodeDegree bridgesGraph "C" `shouldEqual` 3
    nodeDegree bridgesGraph "D" `shouldEqual` 3
```

test/Examples/BridgesKoenigsberg.purs (lines 100-106)

And this already shows that our graph does *not* have an Eulerian trail.

### Checking for Eulerian trail

But we want to create a function that can tell this for any state graph. Let's do that now, we'll use this function in the next example again:

```
hasEulerTrail :: StateGraph -> Boolean
hasEulerTrail graph =
  let
    nodes :: Array StateNode
    nodes = fromFoldable (Graph.getNodes graph)

    countEdgesByNode :: Array Int
    countEdgesByNode = map (nodeDegree graph) nodes

    sumOddEdges :: Int
    sumOddEdges = Array.length (Array.filter Int.odd countEdgesByNode)
  in
    sumOddEdges == 2 || sumOddEdges == 0
```

test/Examples/Common.purs (lines 24-36)

The implementation is pretty straightforward. We get all nodes, count the number of edges connected to each node, filter the odd ones and count them. If the count is 2 or 0, we have an Eulerian trail, otherwise we don't. We can use this function to test our graph:

```
specEulerTrail :: Spec Unit
specEulerTrail = do
  it "should not have an Eulerian trail" do
    hasEulerTrail bridgesGraph `shouldEqual` false
```

test/Examples/BridgesKoenigsberg.purs (lines 108-111)

#### Graph analysis with the classic state machine approach

If we wanted to perform similar analysis with the classic state machine approach, we would need to generate all possible 5040 walks and empirically check if any of them visit all bridges exactly once. This is due to the fact that the specification of state machine transitions is buried inside the update function.

### Generating Documentation

For generating the state diagram we add some more options to the `generate` function:

- `undirectedEdges`: The state diagram will be displayed as an undirected graph.
- `fontSize`: Since our edges and labels consist of only single characters, we can make their font size a bit larger.

```
generateGraphLight :: Effect Unit
generateGraphLight = do
  let
    graph :: GraphvizGraph
    graph = TransitGraphviz.generate bridgesTransit _
      { theme = themeHarmonyLight
      , undirectedEdges = true
      , fontSize = 14.0
      }
  FS.writeTextFile UTF8 "renders/bridges-koenigsberg_graph-light.dot" (Graphviz.toDotStr graph)
```

test/Examples/BridgesKoenigsberg.purs (lines 124-133)

**Generated Output**: This produces the graph visualization we examined earlier in this example.

Link: View diagram on GraphvizOnline

The transition table is generated the same way as before, but we also add the `undirectedEdges` option to the options:

```
generateTable :: Effect Unit
generateTable = do
  let
    table :: Table
    table = TransitTable.generate bridgesTransit _
      { undirectedEdges = true
      }
  FS.writeTextFile UTF8 "renders/bridges-koenigsberg_table.md" (Table.toMarkdown table)
```

test/Examples/BridgesKoenigsberg.purs (lines 146-153)

**Generated Output**: This creates the transition table in the undirected format, showing all possible bridge crossings between the land areas.

### Conclusion

This example demonstrates that **Transit**'s value extends far beyond state machine documentation. By reflecting the type-level specification to a term-level graph data structure, you gain access to a rich ecosystem of third-party graph algorithms and analysis tools.

In the next example, we'll see a graph that **does** have an Eulerian trail.

## Example 4: House of Santa Claus

Full source code: *test/Examples/HouseOfSantaClaus.purs*

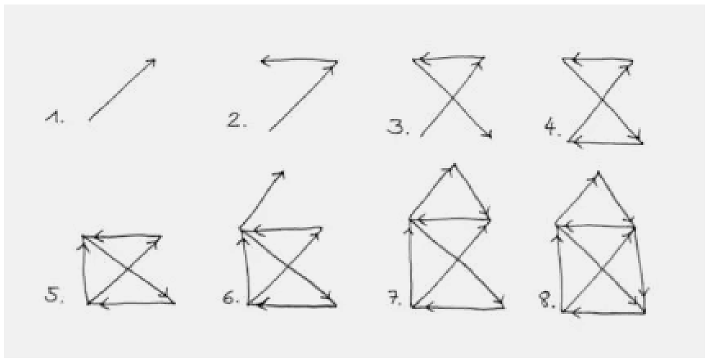

Figure 7: House of Santa Claus drawing game

Do you remember the puzzle where you try to draw the house of Santa Claus in one continuous line — without lifting your pen and without retracing any line? And while doing that you were supposed to say out loud the 8 syllables "This-Is-The-House-Of-San-Ta-Claus" — one syllable for each line.

### The State Machine

Of course we can model this puzzle as a state machine. We have states (1 through 5) and 8 transitions (a through h). Again, this is an undirected graph:

Accordingly, the transition table looks like this:

| State |  | Message |  | State |
|---|---|---|---|---|
| 1 | ⟵ | a | ⟶ | 2 |
| 2 | ⟵ | b | ⟶ | 3 |
| 3 | ⟵ | c | ⟶ | 5 |

| State |  | Message |  | State |
|---|---|---|---|---|
| 5 | ⟵ | d | ⟶ | 4 |
| 4 | ⟵ | e | ⟶ | 1 |
| 1 | ⟵ | f | ⟶ | 3 |
| 2 | ⟵ | g | ⟶ | 4 |
| 3 | ⟵ | h | ⟶ | 4 |

## Implementation using Transit

### State and message types

Nothing special here. We define the state and message types like we did in the previous examples. By using Variants, we have no constraint on how the labels are named; we can use any type-level string we want: numbers, lowercase letters, etc. Traditional ADTs wouldn't give us this flexibility.

```
type State = Variant
  ( "1" :: {}
  , "2" :: {}
  , "3" :: {}
  , "4" :: {}
  , "5" :: {}
  )

type Msg = Variant
  ( "a" :: {}
  , "b" :: {}
  , "c" :: {}
  , "d" :: {}
  , "e" :: {}
  , "f" :: {}
  , "g" :: {}
  , "h" :: {}
  )
```

test/Examples/HouseSantaClaus.purs (lines 27-44)

### Type-level specification

The transit specification follows the same pattern as in the previous example.

```
type SantaTransit =
  Transit
    :* ("1" |< "a" >| "2")
    :* ("2" |< "b" >| "3")
    :* ("3" |< "c" >| "5")
    :* ("5" |< "d" >| "4")
    :* ("4" |< "e" >| "1")
    :* ("1" |< "f" >| "3")
    :* ("2" |< "g" >| "4")
    :* ("3" |< "h" >| "4")
```

test/Examples/HouseSantaClaus.purs (lines 46-55)

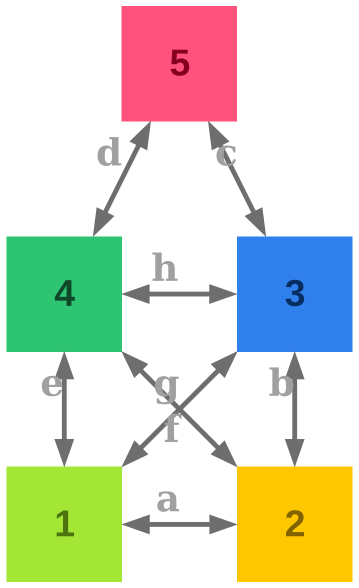

Figure 8: House of Santa Claus state diagram

### The Update Function

Until now, we have always manually defined the update function. In most cases this will be the way to go. But you may have noticed that in some cases this is sheer boilerplate. We can let the compiler generate the update function for us by using the `mkUpdateAuto` function. This works if the following conditions are met:

- There are no conditional transitions in the state machine.
- State transitions don't change the type of the state payload.

Both conditions are met in our case, so we can use `mkUpdateAuto` to generate the update function for us. We could have used it in the Door example and the Bridges of Königsberg example as well.

```
update :: State -> Msg -> State
update =
  mkUpdateAuto @SantaTransit
```

test/Examples/HouseSantaClaus.purs (lines 57-59)

## Testing the state machine

At the beginning of this chapter, we saw an image of one possible solution to the puzzle. Let's write a test to verify that the update function follows this solution:

```
specWalk :: Spec Unit
specWalk =
  it "should follow the walk and visit the expected intermediate states" do
    assertWalk update
      (v @"1")
      [ v @"f" ~> v @"3"
      , v @"h" ~> v @"4"
      , v @"g" ~> v @"2"
      , v @"a" ~> v @"1"
      , v @"e" ~> v @"4"
      , v @"d" ~> v @"5"
      , v @"c" ~> v @"3"
      , v @"b" ~> v @"2"
    ]
```

test/Examples/HouseSantaClaus.purs (lines 68-81)

Since this test passes, we know that the state machine has an Eulerian trail. We can also assert that with the `hasEulerTrail` function we defined earlier:

```
specEulerTrail :: Spec Unit
specEulerTrail =
  it "should have an Eulerian trail" do
    let
      graph :: StateGraph
      graph = mkStateGraph santaTransit

    hasEulerTrail graph `shouldEqual` true
```

test/Examples/HouseSantaClaus.purs (lines 83-90)

## Generating documentation

Until now we always used automatic layouts for the state diagram. This is super convenient because you don't have to worry about the layout at all. Sometimes you want more control over the layout. Luckily we can also position the nodes manually by using the `Manual` layout. We'll do this here to make our state diagram look exactly like the drawing of the house of Santa Claus.

Let's do a quick sketch in a 2D grid. The nodes are positioned at the following coordinates:

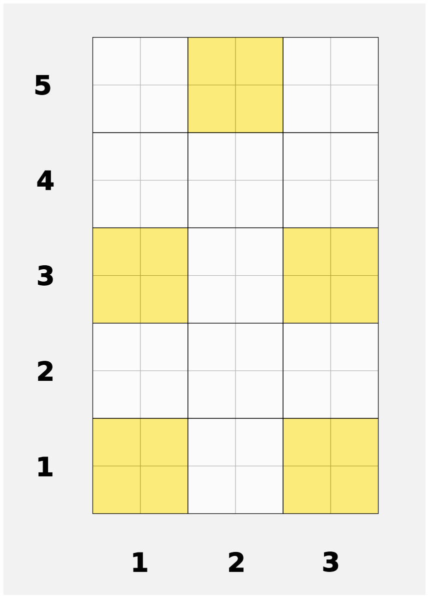

Figure 9: House of Santa Claus layout positions

For historical reasons, the Graphviz renderer wants positions to be defined in inches. We want to use 0.6 inches as the base unit and have all positions be a multiple of that. This number is an arbitrary choice and you can adjust it to increase or decrease the size of the graph.

```
baseUnit :: Inch
baseUnit = Inch 0.6
```

test/Examples/HouseSantaClaus.purs (lines 102-103)

Since we'll need vectors of inches to position the nodes, we define a helper function to create them as multiples of the base unit.

```
units2D :: Int -> Int -> { x :: Inch, y :: Inch }
units2D x y =
  { x: Inch (Int.toNumber x * unwrap baseUnit)
  , y: Inch (Int.toNumber y * unwrap baseUnit)
  }
```

test/Examples/HouseSantaClaus.purs (lines 105-109)

Finally we can generate the graph with the `generateGraphDark` function and define the nodes positions in the options of the `Manual` layout. The `exact` boolean means that the position is exact and not meant as just a hint for the layout algorithm.

```
generateGraphDark :: Effect Unit
generateGraphDark = do
  let
    graph :: GraphvizGraph
    graph = TransitGraphviz.generate santaTransit _
      { undirectedEdges = true
      , theme = themeHarmonyDark
      , layout = Manual
          [ { node: "1", pos: units2D 0 0, exact: true }
          , { node: "2", pos: units2D 2 0, exact: true }
          , { node: "3", pos: units2D 2 2, exact: true }
          , { node: "4", pos: units2D 0 2, exact: true }
          , { node: "5", pos: units2D 1 4, exact: true }
          ]
      , fixedNodeSize = pure $ units2D 1 1
      , fontSize = 14.0
      }

  FS.writeTextFile UTF8
    "renders/house-santa-claus_graph-dark.dot"
    (Graphviz.toDotStr graph)
```

test/Examples/HouseSantaClaus.purs (lines 133-153)

**Generated Output**: This generates the diagram we saw at the beginning of this example.

Link: View diagram on GraphvizOnline

Note that the size of the nodes is also fixed to a multiple of the base unit. In this way we can better control the position in the grid we drew at the beginning.

## Conclusion

This example demonstrated several advanced features of **Transit**:

- **Automatic update function generation**: When your state machine has no conditional transitions and preserves state payload types, you can use `mkUpdateAuto` to let the compiler generate the update function for you, eliminating boilerplate code.
- **Manual layout control**: Unlike the previous examples that used automatic layouts, we showed how to precisely position nodes using the `Manual` layout option, giving you complete control over the visual representation of your

state machine.

- **Graph analysis verification**: We verified that this graph has an Eulerian trail using the same `hasEulerTrail` function from the previous example, demonstrating how **Transit**'s graph analysis capabilities work consistently across different state machines.

Together with the previous examples, we've seen how **Transit** provides a comprehensive solution for building type-safe state machines, generating documentation, and performing graph analysis — all from a single type-level specification.

## Advanced Update Patterns

So far we've used `mkUpdate` to create pure update functions that simply transform state based on messages. However, real-world applications often need more sophisticated behavior: logging transitions, performing side effects, or explicitly handling invalid transitions. **Transit** provides three advanced patterns to cover these scenarios:

- **Monadic Updates** (`mkUpdateM`) - perform effects during transitions
- **Error Handling** (`mkUpdateMaybe`) - explicitly track valid vs invalid transitions
- **Combining Both** (`mkUpdateMaybeM`) - effects with error handling

Let's explore each pattern in detail.

### Monadic Updates

Full source code: *test/Examples/Monadic.purs*

In some cases, you may want your update function to perform effects or collect information during state transitions. **Transit** supports monadic update functions through `mkUpdateM` and `matchM`, which allow you to work within any monad context.

This is useful for scenarios like:

- Logging state transitions
- Accumulating data during updates
- Performing other side effects

Let's see how to create an update function that logs each transition using the `Writer` monad. This is just an example, you can use any monad.

```
type Accum = Array String

update :: forall m. MonadWriter Accum m => State -> Msg -> m State
update = mkUpdateM @DoorTransit
  ( matchM @"DoorOpen" @"Close" \_ _ -> do
      tell [ "You just closed the door" ]
      pure $ return @"DoorClosed"
  )
  ( matchM @"DoorClosed" @"Open" \_ _ -> do
      tell [ "You just opened the door" ]
      pure $ return @"DoorOpen"
  )
```

test/Examples/Monadic.purs (lines 13-24)

The key differences from the non-monadic version are:

- We use `mkUpdateM` instead of `mkUpdate`
- We use `matchM` instead of `match`
- The update function returns `m State` instead of just `State`
- Inside each match handler, we can perform monadic operations (like `tell` for the Writer monad)
- We wrap the return value with `pure $ return @"..."` to lift it into the monad

Now we can use this monadic update function to process a sequence of messages while collecting logs:

```
walk :: Writer Accum State
walk = do
  let s0 = v @"DoorOpen"
  s1 <- update s0 (v @"Close")
  s2 <- update s1 (v @"Open")
  s3 <- update s2 (v @"Close")
  pure s3
```

test/Examples/Monadic.purs (lines 26-32)

Here we chain multiple state updates in a do-notation, just like any other monadic computation. Each call to `update` not only returns the next state, but also accumulates logs in the Writer monad.

Let's verify that the logs are collected correctly:

```
specLogs :: Spec Unit
specLogs = do
  it "should return the correct state" do
    let
      logs :: Accum
      logs = execWriter walk

    logs `shouldEqual`
      [ "You just closed the door"
      , "You just opened the door"
      , "You just closed the door"
      ]
```

test/Examples/Monadic.purs (lines 34-45)

## Error Handling

Full source code: *test/Examples/ErrorHandling.purs*

With **Transit**'s `mkUpdate` function, you define valid transitions in your type-level specification, and the compiler ensures you implement handlers for each one. When an invalid state-message combination (one not in your specification) is encountered at runtime, `mkUpdate` silently returns the unchanged state. This is sometimes exactly what you want - the state machine simply ignores invalid messages and stays in its current state.

However, sometimes you need to explicitly know whether a transition was valid or not. For these cases, **Transit** provides `mkUpdateMaybe`, which wraps the result in a `Maybe` type. Valid transitions return `Just state`, while invalid transitions return `Nothing`, allowing you to handle the error explicitly.

Here's an update function that only handles valid transitions:

```
update :: State -> Msg -> Maybe State
update = mkUpdateMaybe @DoorTransit
  ( match @"DoorOpen" @"Close" \_ _ ->
      return @"DoorClosed"
  )
  ( match @"DoorClosed" @"Open" \_ _ ->
      return @"DoorOpen"
  )
```

test/Examples/ErrorHandling.purs (lines 13-20)

The key difference is the return type: `State -> Msg -> Maybe State` instead of `State -> Msg -> State`. The function uses `mkUpdateMaybe` instead of `mkUpdate`, but the match handlers remain the same.

When you call this update function with a valid state-message combination, it returns `Just` with the new state:

```
specSuccess :: Spec Unit
specSuccess = do
  it "should return the correct state" do
    update (v @"DoorOpen") (v @"Close") `shouldEqual` Just (v @"DoorClosed")
```

test/Examples/ErrorHandling.purs (lines 22-25)

But when you call it with an invalid combination (like trying to open a door that's already open), it returns `Nothing`:

```
specFailure :: Spec Unit
specFailure = do
  it "should return the correct state" do
    update (v @"DoorOpen") (v @"Open") `shouldEqual` Nothing
```

test/Examples/ErrorHandling.purs (lines 27-30)

## Combining Monads and Error Handling

Full source code: *test/Examples/ErrorHandlingMonadic.purs*

For more advanced scenarios, you can combine error handling with monadic effects using `mkUpdateMaybeM`.

Here's an update function that logs transitions AND returns `Maybe`. Again we use the `Writer` as example, but you can use any monad you want.

```
type Accum = Array String

update :: forall m. MonadWriter Accum m => State -> Msg -> m (Maybe State)
update = mkUpdateMaybeM @DoorTransit
  ( matchM @"DoorOpen" @"Close" \_ _ -> do
      tell [ "Closing door" ]
      pure $ return @"DoorClosed"
  )
  ( matchM @"DoorClosed" @"Open" \_ _ -> do
      tell [ "Opening door" ]
      pure $ return @"DoorOpen"
  )
```

test/Examples/ErrorHandlingMonadic.purs (lines 15-26)

The return type is `m (Maybe State)` - combining the monad `m` (e.g. for effects) with `Maybe` (for error handling). Inside each handler, we can perform monadic operations like `tell` to log transitions.

Now let's use this in a scenario where we attempt an invalid transition in the middle of a sequence:

```
walk :: MaybeT (Writer Accum) State
walk = do
  let s0 = v @"DoorOpen"
  s1 <- MaybeT $ update s0 (v @"Close")
  s2 <- MaybeT $ update s1 (v @"Open")
  s3 <- MaybeT $ update s2 (v @"Close")
  s4 <- MaybeT $ update s3 (v @"Close") -- here we request illegal transition
```

```
    s5 <- MaybeT $ update s4 (v @"Open")
    pure s5
```

test/Examples/ErrorHandlingMonadic.purs (lines 28-36)

Here we use the `MaybeT` transformer to work with the `Writer` monad while handling potential failures. Notice that somewhere along the way, we attempt an invalid transition (closing an already closed door). This will cause the computation to short-circuit - the subsequent transitions won't be executed.

Let's verify that the logs only contain the transitions that actually occurred:

```
spec :: Spec Unit
spec = do
  describe "ErrorHandlingMonadic" do
    it "should return the correct state" do
      let
        logs :: Array String
        logs = execWriter (runMaybeT walk)
      logs `shouldEqual`
        [ "Closing door"
        , "Opening door"
        , "Closing door"
        ]
```

test/Examples/ErrorHandlingMonadic.purs (lines 38-49)

Indeed, the logs stop after the invalid transition.