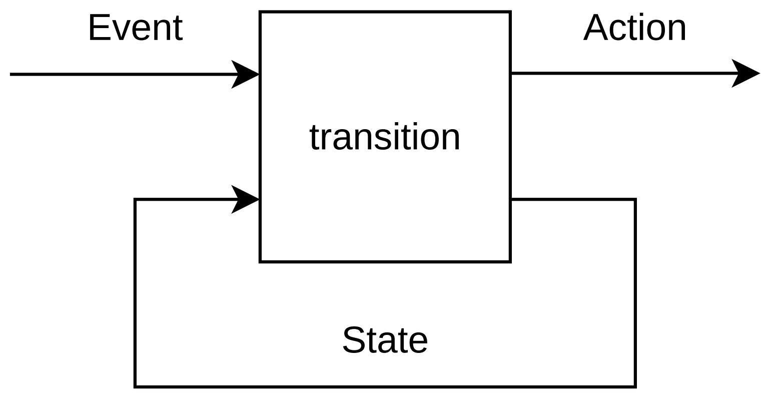
Model View Update

Feedback Loop

A mealy machine consists of:

- a set of states,
- a set of events,
- a set of actions,
- a start state,
- a transition function.



When to use it

- User interfaces
- Embedded systems
- Cyber-physical systems
- Computer games

```
while (1) { switch(event) { ... } }
```

Advantages:

- No deadlocks
- No races
- Testability
- Provability
- Resource bounds

Elm

https://elm-lang.org/

```
import Html exposing (text)

main =
  text "Hello!"
```

Designed by Evan Czaplicki, first appeared 2013.

Elm compiles to JavaScript.

Totality

Total functions are defined for every value in their domain.

Partial functions need not be defined for every value in their domain.

In programming partiality is modeled with exceptions and panics.

Elm is a total functional programming language: there are no exceptions.

(Disregarding non-termination)

demo/Hello.elm

Feedback Loop in Elm

```
type Model

init : Model

update : Event -> Model -> Model

view : Model -> Html Event
```

This idea has various names:

- The Elm Architecture
- Reactor pattern
- Event-driven architecture
- Universe pattern (c.f. Praktische Informatik 1)
- Unidirectional data flow

demo/Increment.elm

Events in Elm

https://package.elm-lang.org/packages/elm/html/1.0.1/Html

```elm
text : String -> Html msg

button : List (Attribute msg) -> List (Html msg) -> Html msg

input : List (Attribute msg) -> List (Html msg) -> Html msg
```

https://package.elm-lang.org/packages/elm/html/1.0.1/Html-Events

```elm
onClick : msg -> Attribute msg

onInput : (String -> msg) -> Attribute msg
```

demo/Payload.elm

Actions in Elm

https://package.elm-lang.org/packages/elm/core/latest/Platform-Cmd#Cmd

```
type Cmd msg
```

> A command is a way of telling Elm, "Hey, I want you to do this thing!" [...]
> Every `Cmd` specifies (1) which effects you need access to and (2) the type of messages that will come back into your application.

```
type Model

init : () -> ( Model, Cmd Event )

update : Event -> Model -> ( Model, Cmd Event )

view : Model -> Html Event
```

Tasks in Elm

https://package.elm-lang.org/packages/elm/core/latest/Task#perform

```
perform : (a -> msg) -> Task Never a -> Cmd msg
```

> Like I was saying in the Task documentation, just having a Task does not mean it is done.

https://package.elm-lang.org/packages/elm/core/latest/Process#sleep

```
sleep : Float -> Task Never ()
```

> Block progress on the current process for the given number of milliseconds.

demo/Sleep.elm

Fetch in Elm

https://package.elm-lang.org/packages/elm/http/latest/Http#get

```
get : { url : String, expect : Expect msg } -> Cmd msg
```

> Create a GET request.

https://package.elm-lang.org/packages/elm/http/latest/Http#expectString

```
expectString : (Result Error String -> msg) -> Expect msg
```

> Expect the response body to be a `String`.

demo/Fetch.elm

Subscriptions in Elm

```
subscriptions : Model -> Sub Event
```

https://package.elm-lang.org/packages/elm/core/latest/Platform.Sub

```
type Sub msg
```

> A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!" [...] Every Sub specifies (1) which effects you need access to and (2) the type of messages that will come back into your application.

https://package.elm-lang.org/packages/elm/time/latest/Time#every

```
every : Float -> (Posix -> msg) -> Sub msg
```

> Get the current time periodically.

demo/Ticker.elm

Composition

We want to compose models, views, and updates of different components.

Product type of models:

```
type alias Model = { model1 : Model1, model2 : Model2 }
```

Sum type of events:

```
type Event = Event1 Event1 | Event2 Event2
```

Event routing in update:

```
update event model = case event of
  Event1 event1 -> ... update1 event1 model.model1 ...
  Event2 event2 -> ... update2 event2 model.model2 ...
```

Visual composition in view:

```
view model =
  ... view1 model.model1 ... view2 model.model2 ...
```

demo/Composition.elm

Virtual Document Object Model

The browser maintains a tree of visual components: the document object model.

Each change to this tree is expensive.

To minimize and batch changes, we keep a virtual copy of this tree.

We calculate the difference to this virtual tree and only apply those changes.

demo/Virtual.elm

Summary and Outlook

Model View Update is simple but verbose.

Next week: Remote Procedures