purescript-bonsai Documentation

Release 1.0.0-beta01

Juergen Gmeiner

Jul 04, 2018

Contents:

Functional web programming in Purescript. Heavily inspired by Elm & using the Elm Virtual DOM.

CHAPTER 1

Introduction

purescript-bonsai is a functional web programming framework for *purescript*. It uses Elm's VirtualDom implementation (the part written in javascript, anyway) and adds the necessary plumbing to make it work with purescript.

The Elm Virtual Dom is tied pretty tightly to how Elm works, so Bonsai follows Elm in a lot of basic design decisions. In particular, the general structure of an application is the same: *TEA* as in "The Elm Architecture".

There is a *Message* type that defines what actions are possible on the *Model*. The *Model* is another type that defines the whole state of the application. All changes to this *Model* go through an *update function*. The *update function* applies messages to the current model and produces a new model.

When the model changes, a *view function* will be called that produces a tree of *Virtual Dom nodes*. This *Virtual Dom tree* is then rendered in the browser.

A classical example for a functional web app is a *Counter*. It displays a number, a "+" button and a "-" button. Clicking the buttons changes the number that is displayed.

In this case, the *Model* is simply an `Int`. The messages can be `Inc` or `Dec`:

```
type Model = Int

data Msg
  = Inc
  | Dec
```

The *update function* applies these messages to the current count. It returns a *Tuple* of command and model. Here the command is empty, but it could also return commands to apply more messages:

```
update msg model = Tuple empty $
  case msg of
    Inc ->
      model + 1
    Dec ->
      model - 1
```

The *view function* produces a tree of *Virtual DOM nodes*. Note that the model never changes, rather a new model (in this case a new number) is produced. The view function always paints the whole state of the application:

```
view model =
  render $ div_ $ do
    text $ show model
    button ! onClick Inc $ text "+"
    button ! onClick Dec $ text "-"
```

I've glossed over some things like imports or types or how the application is started. These will be discussed later on, but you can also look at the complete source code of the counter example:

https://github.com/grmble/purescript-bonsai-docs/blob/master/src/Examples/Basic/Counter.purs

CHAPTER 2

Messages and Commands

Messages are applied to the model. The message type defines the possible actions that can change the model.

Commands are a wrapper around these messages, they encode how the messages are delivered. There are pure commands and tasks.

Commands come from two sources: event handlers, and the update function.[1]

Let's look at an example: you are asked if you want to download some content. If the button is pressed, a progress bar is displayed. Once the animation plays out, the question is shown again.

Lets start with the message type. There are 2 states we track: if there is a download active, and the progress of that download.

```
data Msg
  = Progress Number
  | InProgress Boolean
```

The model is simple as well, it holds the information from the commands:

```
type Model =
  { progress :: Number
  , inProgress :: Boolean
  }
```

The update function applies the messages to the model. This update functions simply applies the incoming messages to the model. But it could issue commands as well:

```
update msg model =
  Tuple empty
    case msg of
      Progress p ->
        model { progress = p }
      InProgress b ->
        model { inProgress = b }
```

[1] You can also arrange for commands to be issued from outside via `issueCommand`

So where are the commands and their messages coming from? As I said, the update function could issue commands, it just does not in this example. In this example, the simulated download is started when the user clicks a button.

```
view m =
  render $
    div_ $
      if m.inProgress
        then do
          p $ text "Downloading all the things!"
          meter ! cls "pure-u-1-2" ! value (show m.progress) $ text (show (100.0*m.
↪progress) <> "%")
        else do
          p $ text "Would you like to download some cat pictures?"
          div_ $ button
            ! cls "pure-button"
            ! typ "button"
            ! disabled m.inProgress
            ! on "click" (const $ pure $ emittingTask simulateDownload)
            $ text "Start Download"
```

We have seen examples with `onClick`. `onClick` is a convenience function that takes a message and issues a command for it - a pure Command, meaning it will emit that particular message and nothing else.

Here we we don't want to emit just one message, we want several, with delays in between. So we have to use `on`. It takes the name of an event ("click") and an event handling function. This is a function that takes a DOM event and produces a `F (Cmd msg)`. `F` is from `Foreign`, it handles failures and gives you do-notation.

`(const $ pure $ emittingTask simulateDownload)` means: our function will ignore the event (`const`) and always produce a sucessful `F`. `emittingTask` is the `Cmd`: it is an `Aff` (think of it like a Thread in other programming languages) that can emit as many messages as it wants because it has a `TaskContext`:

```
simulateDownload :: TaskContext Msg -> Aff Unit
simulateDownload ctx = do
  emitMessage ctx (InProgress true)
  for_ (range 1 100) \i -> do
    delay (Milliseconds 50.0)
    emitMessage ctx (Progress $ 0.01 * toNumber i)
  emitMessage ctx (InProgress false)
```

The other types of tasks are `unitTask``(a task that will not emit any messages, it is useful only because of its side effects) and ``simpleTask` (can emit exactly one message).

The source code for this example is at https://github.com/grmble/purescript-bonsai-docs/blob/master/src/Examples/Basic/Animation.purs

CHAPTER 3

The Update Function

In Elm (or Bonsai), the *model* of an application contains the complete state of the application at any point in time. The model is immutable. The *view function* displays the complete model in the browser.

This means that any observable change must be caused by a change in the current model. How does that work, given that the model is immutable?

Bonsai maintains mutable references for the application:

- a queue of outstanding messages that should be applied to the current model
- the current model

When commands are emitted, their messages will be queued immediately, and Bonsai will try to apply those messages to the current model as soon as possible. It will call the applications *update function* with the then current model and the next outstanding message. The *update function* is responsible for producing the next model state and, optionally, another command.

Updating the model state without issuing any new commands is the common case. The idiom here is `Tuple empty`. An example would be the update function from our earlier counter example:

```
update msg model = Tuple empty $
  case msg of
    Inc ->
      model + 1
    Dec ->
      model - 1
```

A `Dec` message will subtract 1 from the current counter, a `Inc` message will add 1. No additional commands have to be emitted, so it wraps the new model in `Tuple empty`.

In an old version of the animation example, we saw an additional case: a command was issued from the update function. This is accomplished by not returning a plain result, but a real one containing the new model and a (possibly empty) command:

```
case msg of
 SetText str ->
   Tuple (pureCommand EndAnimation) (model { text = str } )
```

Bonsai tries hard to apply as many messages as possible between rendering. Once it has applied all queued messages (and all messages emitted by the updates), and has received no additional messages in the mean time, it will schedule a render via `requestAnimationFrame`. If there still are no unapplied messages in that animation frame, Bonsai will render the model using the view function.

CHAPTER 4

The View Function

The *view function* is responsible for displaying the model in the browser. It always renders the whole model of the application. Because DOM operations in the browser are moderately expensive, a *Virtual DOM* is used.

A *Virtual DOM* is a representation of the DOM tree without any interface to the browser. This representation of the DOM tree can be produced very fast. The *Virtual DOM* also supports computing a *diff* between two virtual DOM trees. This diff can then be applied to the real browser's DOM. Only changed DOM nodes will be touched by this patching operation.

Bonsai provides a Smolder-style syntax to produce virtual DOM nodes.[1] View code is expected to import `Bonsai.Html`, `Bonsai.Html.Attributes` and `Bonsai.Html.Events`. These modules provide helper functions for easily representing HTML content:

```
view :: Model -> VNode Msg
view model =
  render $ div_ $ do
    text $ show model
    button ! onClick Inc $ text "+"
    button ! onClick Dec $ text "-"
```

`render` produces a virtual DOM node from the Smolder-style DSL. `div_` and `button` come from `Bonsai.Html`, they produce their corresponding HTML elements. Child elements are simply nested in a `do` block. Attributes and event handlers are specified with `!` (this is different from Smolder - Smolder uses different syntax for event handlers).

If an attribute or event handler is not always needed, a `Maybe (Property msg)` can be put on the element with `!?`. Elm's virtual DOM has special helpers for styles, and there is an unconditional and a conditional operator for styles as well: `#!` and `#!?` - this is from an old version of the animation example:

```
p #!? (map (\(Color c) -> style "background-color" c) m.color) $
  text m.text
```

The styles helper just makes it possible to not provide a single style attribute, but many different styles (some of them conditional). The DSL takes care of producing the final style attribute for you.

[1] The HTML Api is optional, you can also work with the VirtualDom directly.

Note that with the conditional operators, you usually need `map` because you have to lift over the structure of the `Maybe`.

Also note that class properties (`cls`) are special - if multiple class properties are present, the virtual DOM will join them (separated by a spaces). With all other properties/attributes, later ones overwrite earlier ones.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search