Functional Domain Modelling Works

marcoemrich - last edited 6 days ago

Forked from /s/6tvWP1cCPsjwZJvvrWtcwe

Basic Building Blocks

Primitives

```
/* primitives (type alias) */
type name = string
type name = string;
type age = int
type age = int;
type size = float
type size = float;
type isAlive = bool
type isAlive = bool;


let a: int = 27
let a: int = 27;
let a: age = 27
let a: age = 27;
let a = 27
let a: int = 27;
let a = 5.7;
let a: float = 5.7;

27;
- : int = 27

"foo";
- : string = "foo"
```

Abstract Types

important usage:

1. reduce implementation detail
2. model top-down

```
17 type notYetSpecified;
   type notYetSpecified;
18
```

Algebraic Data Types (ADT)

Sum Types (OR)

(also Variants, Unions)

```
19 type color = Red | Green | Blue | Yellow;
   type color = Red | Green | Blue | Yellow;
20 type shape = Circle | Rectangle;
   type shape = Circle | Rectangle;
21
22 /* 4 possible Values (sum) */
23 Red;
   - : color = Red
```

Product Types (AND)

```
type coordinate = (int, int);
26 type coloredShape = (color, shape);
type coloredShape = (color, shape);
27
28 /* 4 Colors * 2 Shapes = 8 possible Values (product) */
29 let x: coloredShape = (Red, Circle);
let x: coloredShape = (Red, Circle);
30 (7, 3)
- : (int, int) = (7, 3)
31
32 /* Product: Record */
33 type coloredShape = {shape, color};
type coloredShape = { shape: shape, color: color, };
34
35 {shape: Circle, color: Red};
- : coloredShape = {shape: Circle, color: Red}
36 {color: Green, shape: Rectangle};
- : coloredShape = {shape: Rectangle, color: Green}
```

Sum Type with Payload

```
37 /* Sum with Payload aka Type Constructors */
38 type shape = Circle(int) | Square(int, int);
type shape = Circle(int) | Square(int, int);
39
40 Square(3, 4); /* (3) */
- : shape = Square(3, 4)
41 Circle(4); /* (3, 4) */
- : shape = Circle(4)
```

Functions

```
type areaSize = shape => size;
type areaSize = shape => size;
type add = (float, float) => float;
type add = (float, float) => float;
```

Primitive Anti-Pattern

```
let durationToNextRoom = 10.0;
let durationToNextRoom: float = 10.;
let wayToNextRoom = 5.0;
let wayToNextRoom: float = 5.;

let speed = (way: float, time: float) => way /. time;
let speed: (float, float) => float = <fun>;

speed(durationToNextRoom, wayToNextRoom);
- : float = 2.
```

Tagged Types (Sum Type of 1)

```
type secs = Secs(float);
type secs = Secs(float);
type meter = Meter(float);
type meter = Meter(float);

let speed: (meter, secs) => float =
  (Meter(m), Secs(s)) => m /. s;
let speed: (meter, secs) => float = <fun>;
/* let speed = (Meter(m), Secs(s)) => m /. s; */

```

```
let wayToNextRoom =  Meter(5.0);
let wayToNextRoom: meter = Meter(5.);

/* speed(durationToNextRoom, wayToNextRoom); */
```

*/# Lists

```
let x: list(shape) = [];
let x: list(shape) = [];

type simplePicture = { shape: list(shape),  backgroundColor: color};
type simplePicture = { shape: list(shape), backgroundColor: color,
```

Invariant Enforcement

Tennis Scoring Rules:
* Player points: Love, 15, 30, 40
* 40 points && win the ball ⇒ win game
* both player 40 ⇒ players are deuce
* deuce: the winner of a ball ⇒ advantage
* advantage && wins the ball ⇒ win game
* player without advantage wins ⇒ back at deuce

1st Try

```
type points = int;
type points = int;

```

```
  playerTwo: points,
};
type score = { playerOne: points, playerTwo: points, };

let s = {
  playerOne: 1000, /* 1000, -200 */
  playerTwo: -15,
};
let s: score = {playerOne: 1000, playerTwo: -15};
```

2nd Try

```
type points =
  | Love
  | Fifteen
  | Thirty
  | Forty;
type points = Love | Fifteen | Thirty | Forty;

type score = {
  playerOne: points,
  playerTwo: points,
};
type score = { playerOne: points, playerTwo: points, };

let s2 = {playerOne: Fifteen, playerTwo: Love};
let s2: score = {playerOne: Fifteen, playerTwo: Love};
let even = {playerOne: Forty, playerTwo: Forty};
let even: score = {playerOne: Forty, playerTwo: Forty};
```

Better Solution

```
  | PlayerTwo;
type player = PlayerOne | PlayerTwo;

type points =
  | Love
  | Fifteen
  | Thirty;
type points = Love | Fifteen | Thirty;

type score =
  | Points(points, points)
  | Forty(player, points /* of other player */)
  | Deuce
  | Advantage(player)
  | Game(player);
type score =
    Points(points, points)
  | Forty(player, points)
  | Deuce
  | Advantage(player)
  | Game(player);

let startScore: score = Points(Love, Love);
let startScore: score = Points(Love, Love);
let anotherScore: score = Forty(PlayerTwo, Thirty);
let anotherScore: score = Forty(PlayerTwo, Thirty);
let anotherScore2: score = Deuce;
let anotherScore2: score = Deuce;
let anotherScore3: score = Advantage(PlayerOne);
let anotherScore3: score = Advantage(PlayerOne);

/*
 let impossibleScore1: score = Points(Seven, Eleven);
 let impossibleScore2: score = Points(Forty, Forty);
 let impossibleScore3: score = Forty(PlayerTwo, Forty);
 */
```

Mistake

```
/* type option('a) = None | Some('a); */

let reciprocal = x => x == 0.0 ? None : Some(1.0 /. x);
let reciprocal: float => option(float) = <fun>;


type name = Name(string);
type name = Name(string);
type phone = Phone(string);
type phone = Phone(string);

type customer = {
  name,
  phone: option(phone),
};
type customer = { name: name, phone: option(phone), };
```

Tactical Design Pattern and idioms

Model Values explicitly!

ie. replace booleans with variants

```
/* Don't */

```

```
type livingState =
  | ALIVE
  | DEAD;
type livingState = ALIVE | DEAD;
/*  Do */
let cell = ALIVE;
let cell: livingState = ALIVE;
```

Smart Constructors

https://github.com/ostera/reason-design-patterns/blob/master/patterns/smart-constructors.md

```
type name = Name(string); /* Business Contraint: Length < 20 */
type name = Name(string);

type makeName = string => option(name);
type makeName = string => option(name);
let smallerThan20 = s => String.length(s) < 20
let smallerThan20: string => bool = <fun>;

let makeName: makeName = str => (smallerThan20(str)) ?
Some(Name(str)) : None
let makeName: makeName = <fun>;

/* Pretty:

let makeName: makeName = str => (str |> String.length < 20) ?
Some(Name(str)) : None

*/

makeName("Sven");
- : option(name) = Some((Name("Sven")))
```

```
- : option(name) = None
```

Model State Changes explicitly!

* Move Runtime to Compile Time
* Bool → Type

```
/* Bad Idea */
type emailAdress = {
  email: string,
  verified: bool,
};
type emailAdress = { email: string, verified: bool, };

/* Better */
type unverifiedEmail =
  | UnverifiedEmailAdress(string);
type unverifiedEmail = UnverifiedEmailAdress(string);
type verifiedEmail =
  | VerifiedEmailAdress(string);
type verifiedEmail = VerifiedEmailAdress(string);

type email =
  | UnverifiedEmailAdress(string)
  | VerifiedEmailAdress(string);
type email = UnverifiedEmailAdress(string) | VerifiedEmailAdress(stri

type verifyEmail = unverifiedEmail ⇒ option(verifiedEmail);
type verifyEmail = unverifiedEmail ⇒ option(verifiedEmail);
```

Model with Abstract Types;

```
169
170 type customer = {
171   name,
172   address,
173 };
```

```
type customer = { name: name, address: address, };
```

Generics

```
174 type numberOfPlayers = | NumberOfPlayers(int);
175 type duration = Minutes(int) | Hours(int);
176 type boardgame = {numberOfPlayers: int, duration: duration};
177
178 type cover = Leather | Cardboard;
179 type notebooks = {numberOfPages: int, cover: cover};
180
181 type id = int;
182
183 type product('productKind) = {
184   productId: id,
185   productSpecificData: 'productKind,
186 };
187
```

```
type numberOfPlayers = NumberOfPlayers(int);
type duration = Minutes(int) | Hours(int);
type boardgame = { numberOfPlayers: int, duration: duration, };
type cover = Leather | Cardboard;
type notebooks = { numberOfPages: int, cover: cover, };
type id = int;
type product('productKind) = {
  productId: id,
  productSpecificData: 'productKind,
};
```





```
let myNotebook: product(notebooks) =
  {productId: 27, productSpecificData: {numberOfPages: 300, cover: Le
let myBoardgame = {productId: 28, productSpecificData:
{numberOfPlayers: 5, duration: Minutes(90)}};
let myBoardgame: product(boardgame) =
  {productId: 28,
   productSpecificData: {numberOfPlayers: 5, duration: Minutes(90)}};


let productIdOf = product => product.productId;
let productIdOf: product('a) => id = <fun>;

let numberOfPlayersOf = product =>
product.productSpecificData.numberOfPlayers;
let numberOfPlayersOf: product(boardgame) => int = <fun>;
```