A Theory of Functional Programming

November 2020
Eric Normand
grokking
Simplicity
Taming complex software with functional thinking

Eric Normand

lispcast.com/gs

TSSIMPLICITY (50% off)
What is functional programming?
Functional programming is a set of skills
Why functional programming?
Distributed systems
3 Levels of Functional Programming

1. Distinguishing Actions, Calculations, Data

2. First-class abstractions

3. Building composable models
Part 1: Distinguishing Actions, Calculations, Data

- What are actions, calculations, and data?
- How the spreading rule makes actions pernicious
- Recognizing implicit inputs and outputs to functions
- Immutability
- Stratified design
stars indicate you need to be careful

```javascript
{ "firstname": "Eric", "lastname": "Normand" }
```

information about a person

```javascript
sendEmail(to, from, subject, body)
```

be careful with this one, it sends an email

```javascript
sum(numbers)
```

a handy function for adding up some numbers

```javascript
saveUserDB(user)
```

once you save it to the db, other parts of the system can see it

```javascript
string_length(str)
```

if you pass it the same string twice, it returns the same length twice

```javascript
getCurrentTime()
```

each time you call it, you get a different time

```
[1, 10, 2, 45, 3, 98]
```

just a list of numbers
actions depend on when they are called

* sendEmail(to, from, subject, body)
* saveUserDB(user)
* getCurrentTime()

everything else does not depend on when it is called

{“firstname”: “Eric”,
 “lastname”: “Normand”}

sum(numbers)

string_length(str)

[1, 10, 2, 45, 3, 98]
actions depend on when they are called

calculations are computations from inputs to outputs

data is recorded facts about events

**Actions**
- sendEmail(to, from, subject, body)
- saveUserDB(user)
- getCurrentTime()

**Calculations**
- sum(numbers)
- string_length(str)

**Data**
- [1, 10, 2, 45, 3, 98]
- {"firstname": "Eric",
  "lastname": "Normand"}
Actions

the process of doing something, typically to achieve an aim

- Typically: *impure functions, effects, or side-effects*

- Depend on
  - when you run them — *ordering*
  - how many times you run them — *repetition*
Calculations

*computation from inputs to outputs*

- Typically: *pure functions*
- Eternal — outside of time
- Referentially transparent
Data

*facts about events used as a basis for reasoning, discussion, or calculation*

- Inert
- Serializable
- Require interpretation
universal

Calculation

Data

Actions

Haskell
universal

Lisp

Data

Actions

Calculations
function `figurePayout` (affiliate) {
    var owed = affiliate.sales * affiliate.commission;
    if (owed > 100) // don’t send payouts less than $100
        sendPayout(affiliate.bank_code, owed);
}

function `affiliatePayout` (affiliates) {
    for (var a = 0; a < affiliates.length; a++)
        figurePayout(affiliates[a]);
}

function `main` (affiliates) {
    affiliatePayout(affiliates);
}
function figurePayout(affiliate) {
    var owed = affiliate.sales * affiliate.commission;
    if(owed > 100) // don’t send payouts less than $100
        sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
    for(var a = 0; a < affiliates.length; a++)
        figurePayout(affiliates[a]);
}

function main(affiliates) {
    affiliatePayout(affiliates);
}
function figurePayout(affiliate) {
    var owed = affiliate.sales * affiliate.commission;
    if(owed > 100) // don’t send payouts less than $100
        sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
    for(var a = 0; a < affiliates.length; a++)
        figurePayout(affiliates[a]);
}

function main(affiliates) {
    affiliatePayout(affiliates);
}
function figurePayout(affiliate) {
    var owed = affiliate.sales * affiliate.commission;
    if(owed > 100) // don’t send payouts less than $100
        sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
    for(var a = 0; a < affiliates.length; a++)
        figurePayout(affiliates[a]);
}

function main(affiliates) {
    affiliatePayout(affiliates);
}
function figurePayout(affiliate) {
    var owed = affiliate.sales * affiliate.commission;
    if(owed > 100) // don’t send payouts less than $100
        sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
    for(var a = 0; a < affiliates.length; a++)
        figurePayout(affiliates[a]);
}

function main(affiliates) {
    affiliatePayout(affiliates);
}

it’s all actions
Oh, I see. I thought I had one action, but really, all of my code is actions.

```javascript
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout() {
  var affiliates = fetchAffiliates();
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main() {
  affiliatePayout();
}
```
function calc_cart_total() {
    calc_total();
    set_cart_total_dom();
    update_shipping_icons();
    update_tax_dom();
}

function calc_total() {
    shopping_cart_total = 0;
    for(var i = 0; i < shopping_cart.length; i++) {
        var item = shopping_cart[i];
        shopping_cart_total += item.price;
    }
}
current

function calc_cart_total() {
  calc_total();
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
}

eliminated outputs

function calc_cart_total() {
  shopping_cart_total = calc_total();
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total() {
  var total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    total += item.price;
  }
  return total;
}

move the assignment outside to the caller
use the return value to set the global variable
convert it to a local variable
operate on the local variable
return the local
Current

function calc_cart_total() {
  shopping_cart_total = calc_total();
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total() {
  var total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    total += item.price;
  }
  return total;
}

Eliminated inputs

function calc_cart_total() {
  shopping_cart_total = calc_total(shopping_cart);
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total(cart) {
  var total = 0;
  for(var i = 0; i < cart.length; i++) {
    var item = cart[i];
    total += item.price;
  }
  return total;
}
function calc_cart_total() {
    shopping_cart_total = calc_total(shopping_cart);
    set_cart_total_dom();
    update_shipping_icons();
    update_tax_dom();
}

function calc_total(cart) {
    var total = 0;
    for(var i = 0; i < cart.length; i++) {
        var item = cart[i];
        total += item.price;
    }
    return total;
}
Copy-on-write

Mutating

function drop_first(array) {
    array.shift();
}

1. Make a shallow copy
2. Modify the copy
3. Return the copy

Copy-on-write

function drop_first(array) {
    var array_copy = array.slice();
    array_copy.shift();
    return array_copy;
}
Defensive copying

1. Make a **deep copy** as data **leaves** the safe zone
2. Make a **deep copy** as data **enters** the safe zone
Part 2: First-class abstractions

- First-class values help you abstract
- Higher-order iteration (map, filter, reduce) helps clarify your for loops
- Chaining map, filter, and reduce gives you data transformation in steps
- Timelines help you understand how your code might execute
- Higher-order actions help you control your execution
First-class values

function setPriceByName(cart, name, price) {
  var item = cart[name];
  var updatedItem = objectSet(item, 'price', price);
  return objectSet(cart, name, updatedItem);
}

function setQuantityByName(cart, name, quantity)
function setDiscountByName(cart, name, discount)
First-class values

function set\textbf{Price}\textbf{ByName}(\textit{cart}, \textit{name}, \textit{price}) { 
    \text{var item} = \text{cart}[\textit{name}]; 
    \text{var updatedItem} = \text{objectSet}(\text{item}, \textit{price}, \textit{price}); 
    \text{return objectSet(\text{cart}, \textit{name}, \text{updatedItem});}
}

function set\textbf{Quantity}\textbf{ByName}(\textit{cart}, \textit{name}, \textit{quantity})
function set\textbf{Discount}\textbf{ByName}(\textit{cart}, \textit{name}, \textit{discount})
First-class values

function setFieldByName(cart, name, field, value) {
    var item = cart[name];
    var updatedItem = objectSet(item, field, value);
    return objectSet(cart, name, updatedItem);
}
for(var i = 0; i < foods.length; i++) {
    var food = foods[i];
    cook(food);
    eat(food);
}

for(var i = 0; i < dishes.length; i++) {
    var dish = dishes[i];
    wash(dish);
    dry(dish);
    putAway(dish);
}
for(var i = 0; i < foods.length; i++) {
    var food = foods[i];
    cook(food);
    eat(food);
}

for(var i = 0; i < dishes.length; i++) {
    var dish = dishes[i];
    wash(dish);
    dry(dish);
    putAway(dish);
}
Replace body with callback

```javascript
function forEach(array, f) {
  for(var i = 0; i < array; i++) {
    f(array[i]);
  }
}

forEach(foods, function(food) {
  cook(food);
  eat(food);
});
forEach(dishes, function(dish) {
  wash(dish);
  dry(dish);
  putAway(dish);
});
```
Chaining map, filter, reduce

// Average order of a good customer

var sum = 0;

var count = 0;

for(var i = 0; i < customers.length; i++) {
    var customer = customers[i];
    if(customer.purchases.length > 5) {
        for(var p = 0; p < customer.purchases.length; p++) {
            var purchase = customer.purchases[p];
            sum += purchase.total;
            count += 1;
        }
    }
}

var average = sum / count;
Chaining map, filter, reduce

// Average order of a good customer

var goodCustomers = filter(customers, function(customer) {
  return customer.purchases.length > 5;
});

var purchases = flatMap(goodCustomers, function(customer) {
  return customer.purchases;
});

var purchasePrices = map(purchases, function(purchase) {
  return purchase.price;
});

var sum = reduce(purchasePrices, 0, function(a, b) { return a+b; });

var average = sum / purchasePrices.length;
Timelines

MAKING A CHEESE PIZZA

order comes in

make dough

grate cheese

make sauce

roll out dough

spread sauce

spread cheese

put in oven

wait 10 minutes

serve

three robots working in parallel, so three different timelines

Operations on different timelines can interleave. You don’t know what order they happen in.
DOUGH TAKES LONGER

order comes in

grate cheese

make sauce

roll out dough

spread sauce

spread cheese

put in oven

wait 10 minutes

serve

make dough

dough is not ready until after the other robot tried to roll it out
order comes in

make dough

make sauce
  roll out dough
  spread sauce
  spread cheese
  put in oven
  wait 10 minutes
  serve

grate cheese

cheese not ready until after the other robot tried to spread it
ALL 6 POSSIBLE ORDERINGS

- make dough, make sauce, grate cheese
- make dough, make sauce, grate cheese
- make sauce, make dough, grate cheese
- make sauce, make dough, grate cheese
- grate cheese, make dough, make sauce
- grate cheese, make dough, make sauce

It only works when "make sauce" is last.
3-ROBOT SETUP WITH COORDINATION

order comes in

- make dough
- grate cheese
- make sauce

roll out dough
- spread sauce
- spread cheese
- put in oven
- wait 10 minutes
- serve

I can have each robot wait for the others to finish prepping, then one of them can finish the assembly.

That way, it doesn't matter what order the preparation steps are done in.
```javascript
function cut(num, callback) {
    var num_finished = 0;
    return function() {
        num_finished += 1;
        if (num_finished === num) {
            callback();
        }
    };
}
```

- The function `cut(num, callback)` is defined to wait for `num` timelines to finish.
- `var num_finished = 0;` initializes the count to 0.
- The returned function increments `num_finished` each time it's called.
- `if (num_finished === num)` checks if all timelines have finished.
- `callback();` is called when all timelines are done.
A simple example

```javascript
var done = Cut(3, function() {
    console.log("3 timelines are finished");
});

done();
done();
done();

console.log("3 timelines are finished");
```

wait for 3 calls to `done()`, then print a message

- `num_finished = 0`
- `num_finished = 1`
- `num_finished = 2`
- `num_finished = 3`

After third call to `done()`, the message is printed
Grokking Simplicity
Taming complex software with functional thinking

Eric Normand

lispcast.com/gs

TSSIMPLICITY (50% off)
Where to find me

Blog: lispcast.com

Podcast: lispcast.com/podcast

Clojure & functional programming video courses: purelyfunctional.tv

Clojure Newsletter: purelyfunctional.tv/newsletter

Twitter: @ericnormand