Algebraically Refactoring a Weak API

1

Imagine a program which takes a ToDoItem and sets all its attributes. What will happen if an attribute gets renamed, e.g.: ”colour” to ”color”?

Answer

It will set the non-existent field ”colour”; the rest of the program will continue using the unchanged ”color” field.

2

Via the external view, it is possible to describe all states of a ToDoItem without referencing the implementation: a state of a ToDoItem is a call to the constructor with default values followed by a sequence of calls to updateItem such that no key is set twice. Any sequence of calls can be ”normalized” into a sequence of this form by removing all but the last call to each key and sorting the calls by key. Using this as a definition of representable state, how does the implementation of ToDoItem violate the Representable/Valid principle?

Answer

Depending on your perspective, adding calls to updateItem for a nonexistent attribute either represent an invalid `ToDoItem`, or represents the same `ToDoItem` as without that call. Also, no enforcement that the values of updateItem are valid for that attribute.

3

Write down the type of this function (i.e.: the sets of valid inputs and outputs) using the notation from lecture. Change it using algebraic laws to an equivalent type which helps avoid the problems raised in the earlier parts of this question. Update the code accordingly.

Answer

While there is nothing in Python enforcing the types of the arguments, the signature/specification for the `updateItem` function is as follows.

```
updateItem ::
(S("dueDate")*Date + S("description")*String + S("status")*Status +
 ↪  S("colour")*Color + S("isPublic") * bool) -> unit
```

Where `S('<name>')` is the singleton type, so that `updateItem` may only be called with certain strings.

The `updateItem` method can be written in a more algebraic notation as

```
// A function from certain pairs of values to unit (unit = 1).
1^ (S("dueDate")       * Date    +
    S("description")   * String  +
    S("status")        * Status  +
    S("colour")        * RGBA    +
    S("isPublic")      * Boolean)
```

We can multiply out the terms in the exponent following the law `a^(b+c) = a^b * a^c` .

```
1^(S("dueDate")      * Date)     *
1^(S("description")  * String)   *
1^(S("status")       * Status)   *
1^(S("colour")       * RGBA)     *
1^(S("isPublic")     * Boolean))
```

This algebraic expression corresponds to multiple functions from pairs to unit. In Python, this would look like this

```
setDueDate(s: "dueDate", date: Date): unit
...
```

The constant strings only have one valid member of the type so we can replace it with them with a 1

```
1^(1 * Date)    *
1^(1 * String)  *
1^(1 * Status)  *
1^(1 * RGBA)    *
1^(1 * Boolean))
```

Having a constant value is *isomorpic* to not having that value, `1 * x = x` .

```
1^Date    *
1^String  *
1^Status  *
1^RGBA    *
1^Boolean
```

This refactoring corresponds to having multiple functions of one argument to unit

```
setDueDate(d:Date):unit
...
```

There is still at least one improvement we can do `Boolean` is not extensible, so let's change the representation to something better. `Boolean` is not the only type with two members, we can make our own, that later can be extended to three members if we need.

```
Boolean = True + False = 1 + 1 = Public + Private = Visibility
```

The final algebraic expression ends up being

```
1^Date    *
1^String  *
1^Status  *
1^RGBA    *
1^Visibility
```

Which corresponds to this refactoring

```
class ToDoItem:
  ...

  def updateDueDate(self, date):
    setattr(self, "dueDate", date)

  def updateDescription(self, description):
    setattr(self, "description", description)

  def updateStatus(self, status):
    setattr(self, "status", status)

  def updateColor(self, rgba):
    setattr(self, "color", rgba)

  def setVisibility(self, visibility):
    setattr(self, "visibility", visibility)
```

Mechanically Refactoring a Weak API

1

Suppose you wanted to write another function which printed the graphical settings associated with each mode. How would you reuse this code to do so?

Answer

Cannot (except maybe by redefining setColorDepth/drawRect)

More on that last idea: There's actually a legitimate technique in FP folklore similar to this, overriding the core operators. Simon Peyton-Jones and friends used it to extract the dependencies of build tasks (https://www.youtube.com/watch?v=BQVT6wiwCxM). I used it at Apptimize for a similar purpose: getting a list of images to pre-fetch.

2

Suppose someone who hadn't read this code and didn't speak English saw the "mode" argument being passed in. What information would this argument convey to them?

Answer

Nothing, save that it's a string and related to other things named "mode"

3

Refactor the function to replace the "mode" argument with something more semantically meaningful. Doing so should also eliminate the conditional.

Answer

```
enum Config {
  SMALL(8, 1024, 768),
  MEDIUM(16, 1600, 1200);

  private int colorDepth;
  private int width;
  private int height;
}


public void displayGame(Config cfg) {
  setColorDepth(cfg.getColorDepth());
  drawRect(screen, cfg.getWidth(), cfg.getHeight());
}
```

4

Show how to do this refactoring through a sequence of mechanical steps, as in lecture.

Answer

We start from this code:

```
public void displayGame(String mode) {
  if (mode.equals("small")) {
        setColorDepth(8);
        drawRect(screen, 1024, 768);
  } else if (mode.equals("medium")) {
        setColorDepth(16);
        drawRect(screen, 1600, 1200);
  }
}
```

Step 1 We anti-unify the two branches into a common function. Using reverse substitution on each branch + bacward reduction of a function application, we have:

```
public void displayGame(String mode) {
  if (mode.equals("small")) {
      ((x,w,h) -> {
        setColorDepth(x);
        drawRect(screen, w, h);)(8,1024,768)
  } else if(mode.equals("medium"))
      ((x,w,h) -> {
        setColorDepth(x);
        drawRect(screen, w, h);)(16,1600,1200)
  }
}
```

(Remaining steps can be done in different orders:)

Step 2

We hoist this common function out of the if-statement.

For syntactic convenience, we'll treat the code as if "small" and "medium" were guaranteed to be the only two modes.

```
public void displayGame(String mode) {
  ((x,w,h) -> {
        setColorDepth(x);
        drawRect(screen, w, h);
  })(mode.equals("small") ? (8,1024,768) : (16,1600,1200));
}
```

Step 3

These arguments are equivalent to a tuple or named tupled object. We create a Config type

```
public void displayGame(String mode) {
  ((cfg) -> {
        setColorDepth(cfg.getColorDepth());
        drawRect(..., cfg.getWidth(), cfg.getHeight());
  })(mode.equals("small") ? new Config(colorDepth=8, width=1024,
   ↪ height=768) : new Config(colorDepth=16, width=1600,
   ↪ height=1200));
}
```

Step 4

Lift the if statement out of function

```
public void displayGame(Config cfg) {
  ((cfg) -> {
        setColorDepth(cfg.colorDepth);
        drawRect(..., cfg.width, cfg.height);
  })(cfg)
}
```

This means that we replace all caller arguments with

```
(mode.equals("small") ? new Config(colorDepth=8, width=1024,
 ↪  height=768) : Config(colorDepth=16, width=1600, height=1200)))
```

All callers, either the mode is known and this conditional collapses into a single fixed Config, or we continue moving the conditional backwards up until the point where mode is chosen, resulting in having a structured Config object instead of string values.

Step 5: Inline

```
public void displayGame(Config cfg) {
        setColorDepth(cfg.getColorDepth());
        drawRect(..., cfg.getWidth(), cfg.getHeight());
}
```

Why doesn't defunctionalization help here?

Answer

Actually, the original example is already defunctionalized. The refunctionalized form would have separate displayGameSmall and displayGameMedium versions, and use dynamic dispatch to call the appropriate one. The code actually does the same thing in both branches, but with different values. So, we need to factor out a single common function, whereas defunctionalization/refunctionalization is always for multiple distinct operations, rather than the same operation with different values.