GTD in Haskell

`for`-loops

Susan Potter

2019-10-11

Revisiting `for`-loops

Refresher

`for`-loops have been a staple of imperative languages, starting with `FORTRAN`'s `DO` loop in 1957.

Syntax

```
        DO label counter = first, last, step
          statements
label     statement
```

Example

```
DO 9, COUNTER = 1, 5, 1
  WRITE (6,8) COUNTER
  FORMAT( I2 )
CONTINUE
```

History

- 1957: FORTRAN provided the `DO` loop
- 1960: COBOL provided `PERFORM` verb with many options akin to for-loop
- 1964: BASIC gives us FOR-NEXT loops and PL/I offers first `break` in `LEAVE`
- 1968: Algol 68 gives us the first universal `for` loop as we know it toay.

...

Variations

Mainstream languages today offer various types of for-loops:

- numeric/traditional `for`-loops (most common, low-level)
- iterator-based `for`-loops (most common, high-level)
- vectorized `for`-loops (niche, e.g. FORTRAN 95)
- compound `for`-loops (niche, e.g. ALGOL 68)

Low-level for vs higher-level ”foreach” (Java)

```java
Pet pets[] = {
  new Pet("Garfield"),
  new Pet("Odie"),
  new Pet("Nermal")
};
```

```java
// low-level (numeric) for-loop
for (int i = 0; i < pets.length; i++) {
  System.out.println(pets[i]);
}
```

```java
// slightly higher-level "foreach"-style loop
for (int pet : pets) {
  System.out.println(pet);
}
```

for examples (Ruby)

```
pets = [
  Pet.new("Garfield"),
  Pet.new("Odie"),
  Pet.new("Nermal")
]

// language construct
for pet in pets
  puts pet
end

// non-language-based variations
pets.size.times { |idx| puts pets[idx] }
pets.each_index { |idx| puts pets[idx] }
pets.each { |pet| puts pet }
```

Basic intuition

iteration (*noun*): the action or a process of iterating or repeating: such as

- a procedure in which repetition of a sequence of operations yields results successively closer to a desired result
- the repetition of a sequence of computer instructions a specified number of times or until a condition is met — compare to recursion

Related:

- recursion
- `while` loops

Levels of abstraction

Application developers

Build software for specialized domains not generalized computing, e.g.:

- publishing
- investment management
- cloud infrastructure management
- generating computer music
- financial planning
- dashboarding data
- pretty much every piece of software is ”domain-specific”

Focus on the *what* not the *how*

When building programs with `for`, `while`, `do` loops, we are focusing on the *how*.

We need to remove:

- mutable state
- loops

But how?

Higher-level primitives

- `filter`
- `map`
- `reduce`
- apply effects
- composition of the above

`filter` in Ruby

```ruby
pets = [
  { name: "garfield", type: :cat },
  { name: "odie", type: :dog },
  { name: "nermal", type: :cat }
]
cats = pets.select { |p| p[:type] == :cat }
# cats contains garfield and nermal

dogs = pets.select { |p| p[:type] == :dog }
# dogs contains odie
```

map in Ruby

```ruby
pet_names = pets.collect do |p|
  p[:name]
end
# => ["garfield", "odie", "nermal"]

pet_types = pets.collect do |p|
  p[:type]
end
```

In many mainstream languages like Ruby these data transformations are available on basic data structures like `Array`-s, `Hash`-es, etc.

`reduce` in Ruby

```ruby
cat_names_string = cat_names.join(', ')
# => "garfield, nermal"
```

In Ruby (and most mainstream languages) these *reduction* methods are monomorphic. This means we can't abstract in the general case!

apply [side] effects in Ruby

```ruby
# typically found inline in controller or
  view code in Rails
pets.each do |p|
  stream_via_websockets_to_client p.to_json
end
```

In Ruby we have side effects not *explicit* effects. This means we compromise on:

- testability
- composition

`stream_via_websockets_to_client` isn't abstracted therefore can't test without mocking which isn't high assurance testing technique.

Composition of `filter`, `map`, `reduce`, `apply`, part 1

```ruby
cats = pets.select { |p| p[:type] == :cat }
cat_names = cats.collect { |c| c[:name] }
# => ["garfield", "nermal"]

puts cat_names.join(', ')
# print to stdout "garfield, nermal"
```

In mainstream languages we must:

- use `#select`, `#collect`, `#join`, etc. - the *pure* versions - not `#select!`, `#collect!`, or manual mutation.
- not include side effects in the given blocks
- not use `#each`
- no language support to ensure expressions are used not statements with side effects

State of the `union(filter, map, reduce, apply)` in mainstream languages

- No general abstraction (specific to data structures like `Array`, `Hash`, etc.)
- Still hard to test [side] effects in isolation
- Some composition possible, but not general case
- Lack of reuse since `Enumerable` interface is implemented per data type wholesale

Can we do better?

Assume we have the following Haskell defined

```haskell
import Relude

data Pet
  = MkPet { petName :: Text
          , petAnimal :: Animal }
          deriving (Show)
data Animal = Cat | Dog deriving (Show)
mkCat name = MkPet name Cat
mkDog name = MkPet name Dog

pets =
  [ mkCat "garfield"
  , mkDog "odie"
  , mkCat "nermal" ]
```

`filter` in Haskell

```haskell
isCat (MkPet _ Cat) = True
isCat _             = False

isDog (MkPet _ Dog) = True
isDog _             = False

cats = filter isCat pets
dogs = filter isDog pets
```

map in Haskell

```haskell
-- Using (<&>) which is flipped fmap aka (<$>)
-- >>> :t (<&>)
--(<&>) :: Functor f => f a -> (a -> b) -> f b
-- >>> catNames
-- ["garfield","nermal"]
catNames = cats <&> petName

-- >>> :t (<$>) -- alias for fmap
--(<$>) :: Functor f => (a -> b) -> f a -> f b
-- >>> dogNames
-- ["odie"]
dogNames = petName <$> dogs
```

reduce in Haskell

```haskell
import Data.List.NonEmpty (NonEmpty)
import Data.Semigroup (Semigroup (sconcat))

-- >>> :i NonEmpty
-- data NonEmpty a = a :| [a]
--
-- >>> :i Semigroup
-- class Semigroup a where
--  (<>) :: a -> a -> a
--  sconcat :: NonEmpty a -> a
--  stimes :: Integral b => b -> a -> a
--  {-# MINIMAL (<>) #-}
reduceS :: Semigroup a => NonEmpty a -> a
reduceS = sconcat

```

`reduce` in Haskell (usage)

```haskell
namesString = mconcat . intersperse ", "

-- >>> catNamesString
-- "garfield, nermal"
catNamesString = namesString catNames

-- >>> dogNamesString
-- "odie"
dogNamesString = namesString dogNames
```

ap-ply in Haskell

```haskell
import Data.Traversable (traverse)
import System.IO (FilePath, IOMode (..), hPutStrLn, withFile)

writePet :: Handle -> Pet -> IO Pet
writePet h (MkPet name animal) = hPutStrLn h $ textRep
  where textRep = show animal <> " { " <> show name <> "}"

writePets :: FilePath -> [Pet] -> IO ()
writePets path = withFile path WriteMode $ void $ traverse (writePet h)
```

Composition in Haskell

- With Haskell's equational reasoning we get substitution for free
- Use general abstract notion of `Functor` to transform underlying data
- Implementing `fmap` for your datatype in the `Functor` instance will net you many `Functor` based functions for free
- Similarly for `Monoid`, `Traversable`, `Applicative`, etc.
- Use general abstract notion of `Monoid` to reduce a list of values to one (e.g. sum, concatenate)
- Compose effectful actions in the same effect context (in this case IO, below):

```haskell
main = do
  writePets "/tmp/cats.txt" cats
  writePets "/tmp/dogs.txt" dogs
```

Results (Haskell)

```
$ cat /tmp/cats.txt
Cat{ "garfield"}
Cat{ "nermal"}
```

```
$ cat /tmp/dogs.txt
Dog{ "odie"}
```

Encoding **for** in Haskell using more generalized but expressive types:

```haskell
-- In Data.Traversable
for
  -- constraints
  :: (Traversable t, Applicative f)
  -- "traversable" collection of elements
  => t a
  -- given an element of the collection
  ↪  produce a b in context f
  -> (a -> f b)
  -- produce in the effect context f the
  ↪  resultant traversable of b's
  -> f (t b)
```

The monadic form:

```
-- In Control.Monad
forM
  -- constraints
  :: (Traversable t, Monad m)
  -- "traversable" collection of a's
  => t a
  -- monadic function to perform for each
   ↪  element
  -> (a -> m b)
  -- produce new "traversable" in monadic
   ↪  effect context m
  -> m (t b)
```

Most specific forms of `for`

```haskell
-- In import Data.Foldable
foldMap
  -- constraints
  :: (Foldable t, Monoid m)
  -- function to convert a collection element to monoidal value
  => (a -> m)
  -- collection of "foldable" of a elements
  -> t a
  -- produce reduced monoidal result
  -> m

scorePet (MkPet _ Cat) = Sum 1
scorePet (MkPet _ Dog) = Sum 2
example = foldMap scorePet pets
```