Functional Pearl: The Decorator Pattern in Haskell

Nathan Collins Tim Sheard
Portland State University
nathan.collins@gmail.com sheard@cs.pdx.edu

Abstract

The Python programming language makes it easy to implement *decorators*: generic function transformations that extend existing functions with orthogonal features, such as logging, memoization, and synchronization. Decorators are modular and reusable: the user does not have to look inside the definition of a function to decorate it, and the same decorator can be applied to many functions. In this paper we develop Python-style decorators in Haskell generally, and give examples of logging and memoization which illustrate the simplicity and power of our approach.

Standard decorator implementations in Python depend essentially on Python's built-in support for arity-generic programming and imperative rebinding of top-level names. Such rebinding is not possible in Haskell, and Haskell has no built-in support for arity-generic programming. We emulate imperative rebinding using mutual recursion, and open recursion plus fixed points, and reduce the arity-generic programming to arity-generic currying and uncurrying. In developing the examples we meet and solve interesting auxiliary problems, including arity-generic function composition and first-class implication between Haskell constraints.

1. Decorators by Example in Python and Haskell

We begin by presenting Python and Haskell decorators by example, while glossing over a lot of details which will be provided in later sections. This section serves both to motivate the problem and give the intuition for our solution. The code described in this paper, and more elaborate examples not described here, are available on GitHub [3].

Our example decorators are call-tracing and memoization, and our example function to decorate is natural exponentiation b^p. We choose this example function because 1) it admits an obvious recursive implementation which makes redundant recursive calls, and 2) it's not a unary function.[1] The recursion makes call-tracing interesting and the redundant recursion makes memoization applicable. We care about higher arity because we want our decorators to be arity generic.

[1] For unary functions an obvious example is Fibonacci, which we consider later in Section 2.1.

Suppose we implement exponentiation in Haskell, using divide-and-conquer:

```
pow b p =
  if p <= 1
  then b * p
  else pow b (p `div` 2) * pow b (p - (p `div` 2))
```

And equivalently, in Python:

```
def pow(b, p):
  if p <= 1:
    return b * p
  else:
    return pow(b, p//2) * pow(b, p - (p//2))
```

Now, suppose we want to observe our function in order to debug it. One way to do this would be to print out call-trace information as the function runs. This could be accomplished by interleaving print statements with our code (using `Debug.Trace` in Haskell): ugly, but it works.

In Python, we can instead do something modular and reusable: we can write a generic call-tracing decorator:

```
LEVEL = 0
def trace(f):
  def traced(*args):
    global LEVEL
    prefix = "| " * LEVEL
    print prefix + ("%s%s" % (f.__name__ , args))
    LEVEL += 1
    r = f(*args)
    LEVEL -= 1
    print prefix + ("%s" % r)
    return r
  return traced
```

For those not familiar with Python, decorators in Python are explained in more detail in Appendix A. Their utility depends heavily on being arity-generic[2] and being able to trap recursive calls to the function being traced[3]. After adding the line

```
pow = trace(pow)
```

to the source program, we run `pow(2, 6)` and see

```
pow(2, 6)
| pow(2, 3)
| | pow(2, 1)
```

[2] In Python, `*args` as a formal parameter, in `def traced(*args)`, declares a variadic function, like `defun traced (&rest args)` in LISP; `*args` as an actual parameter, in `f(*args)`, applies a function to a sequence of arguments, like `(apply f args)` in LISP; `%` as a binary operator is format-string substitution.

[3] In Python, function names are just lexically scoped mutable variables, so trapping is simply a matter of redefinition.