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. 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:

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

And equivalently, in Python:

```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:

```python
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 and being able to trap recursive calls to the function being traced. After adding the line

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

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

1 In Python, *args as a formal parameter, in def traced(*args), declares a variadic function, like `defun traced (krest 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.

2 In Python, function names are just lexically scoped mutable variables, so trapping is simply a matter of redefinition.
Noting the repeated sub computations, we see that memoization would be an improvement. So, we write a generic memoization decorator:

```python
def memoize(f):
    cache = dict()
    def memoized(*args):
        if args not in cache:
            cache[args] = f(*args)
        return cache[args]
    memoized.__name__ = f.__name__
    return memoized
```

and replace the line `pow = trace(pow)` with `pow = trace(memoize(pow))`

Running `pow(2, 6)`, we see

```
2
pow(2, 6)
|  8
|  pow(2, 3)
|  |  2
|  |  pow(2, 1)
|  |  |  2
|  |  |  pow(2, 1)
|  |  |  2
|  |  |  pow(2, 1)
|  |  2
|  8
|  pow(2, 3)
|  8
64
```

Arity-generic decorators are easy to write in Python, and are reusable. In Haskell, things do not appear to be so simple. But, it turns out that, in Haskell it’s also easy to write arity-generic decorators! Indeed, that’s what this paper is about.

An arity-generic decorator needs to solve two problems: intercept recursive calls and handle functions of any arity uniformly. In Python, arity genericity is easy to implement via the built-in `*args` feature, and a function name is simply a statically scoped mutable variable, so a simple assignment can be used to intercept recursive calls. In Haskell these problems need to be solved in another way.

Let’s start with arity-genericity. What Python’s `*args` feature does is allow us to treat functions of any arity uniformly as `unary` functions that instead take a single `tuple` as argument. In Haskell then, a good analogy is arity-generic currying and uncurrying:

```plaintext
curry f x1 ... xn = f (x1, ... , xn)
uncurryM f (x1, ... , xn) = f x1 ... xn
```

Here `curry` in Haskell corresponds to `def f(*args):` ... in Python, and `uncurryM` corresponds to `f(*args)` in Python. We’ll explain how to statically type and implement these functions later, but for now we just need to understand them operationally at an intuitive level.

With `curry` and `uncurryM` in hand we can write a well-typed¹ call-tracing decorator in Haskell quite similar to the Python decorator we saw earlier:

```haskell
trace levelRef name f = curry traced where
    traced args = do
        level <- readIORef levelRef
        let prefix = concat . replicate level $ "| "
        putStrLn $ prefix ++ name ++ show args
        modifyIORef levelRef (+1)
        r <- uncurryM f args
        modifyIORef levelRef (subtract 1)
        putStrLn $ prefix ++ show r
        return r
```

Similarly, we can write a well-typed memoization decorator:

```haskell
memoize cacheRef f = curry memoized where
    memoized args = do
        cache <- readIORef cacheRef
        case Map.lookup args cache of
            Just r -> return r
            Nothing -> do
                r <- uncurryM f args
                modifyIORef cacheRef (Map.insert args r)
                return r
```

These decorators are both monadic; we discuss non-monadic decorators in Section 3.

To apply these decorators to `pow` we make two changes: 1) we rewrite `pow` as a monadic function, because the decorators are monadic; 2) we rewrite `pow` as an open-recursive function, so that we can trap recursive calls. In general, (1) is obviously unnecessary if the function we want to decorate is already monadic, and for pure functions we can actually use `unsafePerformIO` to avoid making the function monadic, as we explain in Section 3.2. For (2), we can alternatively use mutual recursion, as we discuss in Section 2.1.

Before decoration, a monadic version of `pow` using (unnecessary) open recursion is

```haskell
openPowM pow b p = do
    if p <= 1
        then pure $ b * p
        else (*$) <$> pow b (p `div` 2) <*>
            pow b (p - (p `div` 2))
powM = fix openPowM
```

We can now decorate `powM` with both memoization and tracing with just a few lines:

```
powM :: Int -> Int -> IO Int
```

²The “M” in “uncurryM” stands for “monadic”.

³In fact, once `curry` and `uncurryM` are defined, GHC 7.6.3 can infer the types of these decorators. In practice, the type annotations make good documentation, but we aren’t ready to explain the types yet, so we postpone them until Section 2.3.2.

⁴Yes, `unsafePerformIO` is easily abused, but we think `Debug.Trace` is a good precedent here, at least in the call-tracing use case.
Running powM 2 6 we see:

```
7
powM(2,(6, ()))
|  powM(2,(3, ()))
|   |  powM(2,(1, ()))
|   |   2
|   |  powM(2,(2, ()))
|   |   |  powM(2,(1, ()))
|   |   |   2
|   |   |  4
|   |  8
|  0
64
```

Of course, it may not yet be obvious how to implement curry and uncurryM. So, it’s time to fill in the details.

## 2. Decorators in Haskell

To implement Python-style decorators in Haskell there are two problems we must solve:

- How to intercept recursive calls, which was solved by imperative rebinding of function names in Python.
- How to treat any number of arguments uniformly, which was solved using *args in Python.

We address each of these in turn.

### 2.1 Intercepting Recursive Calls

We know two ways to intercept recursive calls in Haskell: mutual recursion, and open recursion plus fixed points. Either approach can be used, and which one you use is mostly a matter of style. The mutual recursion scheme is often easier to explain to programmers who are not familiar with fixpoints, but open recursion is often easier to reuse. Both techniques work the same for monadic and non-monadic functions.

We start with mutual recursion. One approach uses a where clause to introduce mutually recursive functions, for example, `fib` and `fib'`:

```haskell
fib :: Int -> Int
fib = fib' where
  fib' n = if n <= 1
    then n
    else fib (n-1) + fib (n-2)
```

The `where` clause isn’t necessary, but it hides the inner function `fib'` from the rest of the program. Now, suppose we have a decorator `dec :: (Int -> Int) -> (Int -> Int)`.

```haskell
fib = fix (dec . openFib)
```

We generalize to n-ary functions in the next section.

### 2.2 Writing Effectful Typed Decorators

The two decorators we discussed in the intro, `memoize` and `trace`, both use effects. In Haskell this means using some kind of monad. Here we use IO to illustrate the techniques, but they easily generalize to other monads. Effectful decorators sometimes take inputs (other than the function being decorated) which are used to initialize these effects. We illustrate this first with a version of `memoize` that works only on unary functions:

```haskell
memoize :: Ord a => IORef (Map.Map a b) -> (a -> IO b) -> (a -> IO b)
memoize cacheRef f = memoized

memoized :: a -> IO b
memoized x = do
  cache <- readIORef cacheRef
  case Map.lookup x cache of
    Just r -> return r
    Nothing -> do
      r <- f x
      modifyIORef cacheRef (Map.insert x r)
  return r
```

We generalize to n-ary functions in the next section.

We illustrate the use of this decorator, by reformulating `fib` into its monadic counterpart `fibM`. We can define `fibM` as follows:

```haskell
fibM _ = if n <= 1
          then n
          else fib (n-1) + fib (n-2)
```

Writing recursive functions in this mutually recursive style may seem pointless, but it makes them amenable to decoration at a negligible cost – less than one line. It is also very easy to re-factor a function not written in this style using regexp-search and replace in your editor. Once done, this split into two mutually recursive functions does not need to be undone if one decides that decoration is no longer necessary. For example, after using `trace` to debug a function, we might want to disable tracing by removing the decorator.

Alternatively, we can use open recursion and fixed points. Given the open-recursive `openFib` defined by

```haskell
openFib :: (Int -> Int) -> (Int -> Int)
openFib fib n =
  if n <= 1
    then n
    else fib (n-1) + fib (n-2)
```

we can rewrite `fib` as a fixed point of `openFib`:

```haskell
fib = fix openFib
```

Next, we can decorate as follows:

```haskell
fib = fix (dec . openFib)
```

To see that this works, note that

```haskell
fix openFib = openFib (fix openFib)
```

is the defining equation for `fib` defined via `openFib`, and that

```haskell
fix (dec . openFib) = (dec . openFib) (fix (dec . openFib))
= dec (openFib (fix (dec . openFib)))
```

I.e., `fix (dec . openFib)` is `dec` applied to a version of `openFib` which calls `fix (dec . openFib)` on recursive calls, and so we see that `dec` intercepts all recursive calls.
fibM :: Int -> IO Int
fibM n = do
  cacheRef <- newIORef Map.empty
  let fib = memoize cacheRef fib'
  let fib' n = 
    if n <= 1
    then pure n
    else (+) <$> fib (n-1) <*> fib (n-2)
  return $ fib n

Or, by using open recursion plus fixed points:
openFibM :: (Int -> IO Int) -> (Int -> IO Int)
openFibM fib n = do
  if n <= 1
  then pure n
  else (+) <$> fib (n-1) <*> fib (n-2)

fibM :: Int -> IO Int
fibM = memoizeM openFib

2.3 Arity-Generic Decorators via Currying and Uncurrying

In this section we generalize decorators for unary functions to decorators for functions of any arity, by defining n-ary currying and uncurrying at the value and type levels.

2.3.1 Motivation

We’d like to generalize unary memoize from the last section to an n-ary version. We start with some hand waving:

memoize :: Ord (a1 , ... , an) => IORef (Map.Map (a1 , ... , an) b) -> (a1 -> ... -> an -> IO b) -> t = a1 -> ... -> an -> m b
memoizeM t = (a1 , ... , an) -> m b

Next, we introduce type families for the parts of t:

ArgsM t = (a1 , ... , an)
RetM t = b
MonadM t = m

Finally, using curry and uncurryM, and our type families, we can make our hand-wavy decorator look legit:

memoizeM :: forall t. (Ord (ArgsM t) , MonadM t ~ IO) => IORef (Map.Map (ArgsM t) (RetM t)) -> t -> t
memoizeM cacheRef f = curry memoized
memoized :: UncurriedM t
memoized args = do
  cache <- readIORef cacheRef
  case Map.lookup args cache of
    Just r -> return r
    Nothing -> do
      r <- uncurryM f args
      modifyIORef cacheRef (Map.insert args r)
      return r

It remains to eliminate the “...”s, which are now hidden in the definitions of curry, uncurryM, and the type families.

2.3.2 Implementing the “...”s

We now formalize the “...”s, producing code that actually type checks in GHC 7.6.3.

Because we want to treat all arities uniformly, and there is no relation in Haskell between the flat tuples of different arities, we instead used nested tuples. For a function of two arguments the previous definitions actually take the form:

curry k x1 ... xn = k (x1 , ... , xn)
uncurryM f (x1 , ... , xn) = f x1 ... xn

where curry k corresponds to def k(*args) in Python and uncurryM f args corresponds to f(*args) in Python.

To type curry and uncurryM, and code which uses them, we introduce some type families. Since tracing and memoization are side-effecting, we restrict our attention to monadic functions. For a monadic function type
t = a1 -> ... -> an -> m b
we define the type family UncurriedM by
UncurriedM t = (a1 , ... , an) -> m b

We can now type curry and uncurryM:
curry :: UncurriedM t -> t
uncurryM :: t -> UncurriedM t

Next, we introduce type families for the parts of t:

ArgeM t = (a1 , ... , an)
RetM t = b
MonadM t = m

Finally, using curry and uncurryM, and our type families, we can make our hand-wavy decorator look legit:

memoize :: forall t. (Ord (ArgsM t) , MonadM t ~ IO) => IORef (Map.Map (ArgsM t) (RetM t)) -> t -> t
memoize cacheRef f = curry memoized
memoized :: UncurriedM t
memoized args = do
  cache <- readIORef cacheRef
  case Map.lookup args cache of
    Just r -> return r
    Nothing -> do
      r <- uncurryM f args
      modifyIORef cacheRef (Map.insert args r)
      return r

It remains to eliminate the “...”s, which are now hidden in the definitions of curry, uncurryM, and the type families.

2.3.3 Implementing the “...”s

We now formalize the “...”s, producing code that actually type checks in GHC 7.6.3.

Because we want to treat all arities uniformly, and there is no relation in Haskell between the flat tuples of different arities, we instead used nested tuples. For a function of two arguments the previous definitions actually take the form:

curry k x1 x2 = k (x1 , (x2 , ()))
uncurryM f (x1 , (x2 , ())) = f x1 x2

We right-nest our tuples because arrow types are right associated.

For t = a1 -> a2 -> m b we actually have

UncurriedM t = (a1 , (a2 , ())) -> m b
ArgeM t = (a1 , (a2 , ()))
RetM t = b
MonadM t = m

The general versions are formalized in the class definitions below.

---

There are clever ways to implement memoization in a pure way, e.g. see http://hackage.haskell.org/package/memoize-0.6, but the simplest way is to mutate a cache.
Our definition of currying (as a Haskell type class) is relatively straightforward, except for a subtlety due to the potential mismatch between iterative tupling at the term and type level: arrow types are right associative, but iterated function application is left associative. If we iterative tupled the arguments in \( curry \ f \ x1 \ x2 \) using an accumulator, we’d get a left-nesting:

\[
(\lambda() \cdot x1), x2 :: (\lambda() \cdot a1), a2)
\]

So, instead, we treat the argument \( f \) to \( curry \ f \ x1 \ x2 \) as a continuation, allowing us to right-nest the argument tuple. The Curry type class formalizes this pattern for all \( n \):

\[
\text{class Curry \ (as :: *) \ (b :: *)) \ where} \\
\text{ type as ->* b :: *} \\
\text{ curry :: (as -> b) -> (as ->* b)}
\]

\[
\text{instance Curry as b => Curry (a , as) b where} \\
\text{ type (a , as) ->* b = a -> (as ->* b)} \\
\text{ curry f x = curry (\(\lambda\) xs -> f (x , xs))}
\]

\[
\text{instance Curry () b where} \\
\text{ type () ->* b = b} \\
\text{ curry f = f ()}
\]

Note that \( \rightarrow* \) is an infix type constructor. Mnemonically, “as \( \rightarrow* \) b” means “insert zero or more \((\ast\text{-}many)\) arrows between the types in the (right-nested) product as and range b”.

The implementation of uncurrying is simple in principle, but complicated in practice in order to avoid overlapping instances: we give one obvious recursive case followed by \( n \) carefully chosen base cases.\(^9\)

\[
\text{class Monad (MonadM t) => UncurryM (t :: *) where} \\
\text{ type MonadM (t m r) = t m} \\
\text{ type RetM (t m r) = r} \\
\text{ type ArgsM (t m r) = ()} \\
\text{ uncurryM :: t -> UncurriedM t}
\]

\[
\text{instance UncurryM b => UncurryM (a -> b) where} \\
\text{ type ArgM (a -> b) = (a , ArgM b)} \\
\text{ type RetM (a -> b) = RetM b} \\
\text{ type MonadM (a -> b) = MonadM b} \\
\text{ uncurryM f (x , xs) = uncurryM (f x) xs}
\]

\[
\text{instance (Monad m , Monad (t m)) => UncurryM (t m r) where} \\
\text{ type ArgM (t m r) = ()} \\
\text{ type RetM (t m r) = r} \\
\text{ type MonadM (t m r) = t m} \\
\text{ uncurryM f () = f}
\]

\[
\text{memoize :: forall t.} \\
\text{ (CurryUncurryM t , Ord (ArgM t)) =>} \\
\text{ IORef (Map.Map (ArgM t) (RetM t)) -> t -> t} \\
\text{ memoize f = curry memoized where} \\
\text{ memoized :: UncurriedM t} \\
\text{ memoized args = do} \\
\text{ cache <- readIORef cacheRef} \\
\text{ case Map.lookup args cache of} \\
\text{ Just r -> return r} \\
\text{ Nothing -> do} \\
\text{ r <- uncurryM f args} \\
\text{ modifyIORef cacheRef (Map.insert args r)} \\
\text{ return r}
\]

Indeed, this type-checks in GHC, and in fact GHC can infer the types. Similarly, the trace from the intro becomes:

\[
\text{trace :: forall t.} \\
\text{ (CurryUncurryM t , Show (ArgM t)) =>} \\
\text{ Show (RetM t)}
\]

The potential overlap we avoid, and the way we avoid it, are both subtle. Naively, we’d like to write a single base case:

\[
\text{instance Monad m => UncurryM (m b) where} \\
\text{ and the same \text{instance (Monad m , Monad (t m)) => UncurryM (t m r) where} ...}
\]

However, these two instance overlap, because \((m b)\) and \((a -> b)\) unify, with substitution \( m = (\rightarrow*) a \).\(^10\) So, instead, we factor the base case into I0 and transformers. Since most non-I0 monads in the standard libraries are defined as transformers applied to \( \text{Id} \), this factoring covers most types in practice! The tricky part of the factoring is

\[
\text{instance (Monad m , Monad (t m)) => UncurryM (t m r)}
\]

This forces \( t :: (* -> *) \rightarrow * \rightarrow * \) and \( (\rightarrow*) :: * \rightarrow * \rightarrow * \) to have incompatible kinds and so overlap is avoided.

Finally, we define a constraint class CurryUncurryM which is shorthand for “currying after uncurrying makes sense”. In long-hand: \( t \) supports uncurrying \((UncurryM t)\), \( t \) supports uncurrying after currying \((Curry (ArgM t) (MonadM t (RetM t)))\), and uncurrying followed by currying is the identity on types \((UncarM t ((->) t) = t)\):

\[
\text{type CurryUncurryM (t :: *) =} \\
\text{ (UncurryM t , Curry (ArgM t) (MonadM t (RetM t))) , (ArgM t ->* MonadM t (RetM t)) \rightarrow t)}
\]

Note that CurryUncurryM \( t \) holds for \text{all concrete monadic types} \( t = a1 -> ... -> an -> m b \) whose monads are \( \text{I0} \), or a transformer. So, this constraint imposes no constraints on the user, in practice. But, the constraint does help the Haskell type checker infer types when a decorator is used.

We now have everything we need to implement an arity-generic decorator with no hand-wavy “...’s:

\[
\text{memoize :: forall t.} \\
\text{ (CurryUncurryM t , Ord (ArgM t)) =>} \\
\text{ IORef (Map.Map (ArgM t) (RetM t)) -> t -> t} \\
\text{ memoize f = curry memoized where} \\
\text{ memoized :: UncurriedM t} \\
\text{ memoized args = do} \\
\text{ cache <- readIORef cacheRef} \\
\text{ case Map.lookup args cache of} \\
\text{ Just r -> return r} \\
\text{ Nothing -> do} \\
\text{ r <- uncurryM f args} \\
\text{ modifyIORef cacheRef (Map.insert args r)} \\
\text{ return r}
\]

\( ^9 \)GHC 7.8 has closed ordered type families, which allow us to write the type functions \( \text{ArgM, RetM} \), and \( \text{MonadM} \) in the naive way. However, without “closed ordered type classes”, we still have trouble implementing the term function \( \text{uncurryM} \) without overlap. We could define \( \text{uncurryM} \) in terms of the uncurry (no “M”) we introduce later, using a type function which computes the length of a nested tuple to instantiate the Proxy \( \text{Nat} \) parameter of uncurry, but we don’t consider that approach here.

\( ^{10} \)The type \((->*) a :: * \rightarrow * \) has a standard monad instance, so the \text{Monad m} precondition of the base case doesn’t disambiguate. But preconditions aren’t used to disambiguate overlapping instances: the open-world assumption means we have to assume an instance does exist if it’s kind correct.
3. Decorating Non-Monadic Functions

So far we've considered monadic decorators for monadic functions, but we might also want non-monadic decorators for non-monadic functions. Towards this end we describe a non-monadic analog of uncurryM. Also in this section, we show how to apply side-effecting decorators to pure functions using unsafePerformIO.\footnote{That Debug.Trace is in the GHC base libraries indicates that people want to trace pure functions.}

3.1 Non-Monadic Decorators for Non-Monadic Functions

The Curry class has nothing to do with monads, so we just need a non-monadic version of uncurryM, which we call Uncurry. There is one issue though: in uncurryM we used the right-most monad to identify the return type. For a curried pure function, however, the return type is not actually well defined! Indeed, the type $a_1 \to a_2 \to b$ could be intended as a higher-order function of one argument $a_1$ that returns a function of one argument $a_2$, or as a curried function of two arguments $a_1$ and $a_2$.

So, we require the user to say how many arguments there are:

```
class Uncurry (n :: Nat) (t :: *) where
    type Args n t :: *
    type Ret n t :: *
    uncurry :: Proxy n -> t -> Uncurried n t
```

```
instantiation Uncurried n t = Args n t -> Ret n t
```

```
instance Uncurry n b => Uncurry (Succ n) (a -> b) where
    type Args (Succ n) (a -> b) = (a -> Args a b)
    type Ret (Succ n) (a -> b) = Ret n b
    uncurry _ f (x, xs) =
        uncurry (Proxy::Proxy n) (f x) xs
```

Because of issues\footnote{In short, GHC 7.6.3 doesn’t reason about injectivity of successor for the GHC.TypeLits.Nat. There is a detailed discussion of the problem on Stack Overflow [2].} with GHC.TypeLits.Nat, we roll our own type-level [9] nats and use Template Haskell [8] to provide friendly literals:

```
data Nat = Zero | Succ Nat
```

3.2 Reusing Monadic Decorators with Non-Monadic Functions

Finally, we show how to reuse monadic decorators with pure functions, via unsafePerformIO. Practically speaking, this gives a much better version of Debug.Trace, and finally approaches the simplicity of the Python decorators for simple use cases.

Our approach is to 1) turn a pure function into a monadic function, by composing its result with return, 2) apply the monadic decorator, and then 3) use unsafePerformIO to escape from IO.

So, for example, for a two-argument function like pow from the intro:

```
pow , pow' :: Int -> Int -> Int
pow = \b p -> unsafePerformIO $ memoize cacheRef
    (\b p -> return $ pow' b p) b p
pow' b p =
    if b <= 1
        then b * p
        else pow b (p 'div' 2) * pow b (p - (p 'div' 2))
```

```
defining the n-ary composition by
    compose :: (Uncurry n t , Curry (Args n t) a) =>
        Proxy n -> (Ret n t -> a) -> t -> Args n t ->* a
    compose p g f = curry (g . uncurry p f)
```

so that e.g.

```
    compose $(proxyNat 2) return pow' b p =
        return $ pow' b p
```

We can capture the whole pattern abstractly:

```
type InjectIO n t = Args n t ->* IO (Ret n t)
```

```
{-# NOINLINE unsafePurify #-}
unsafePurify :: forall n t.
    UnsafePurifiable n t =>
        Proxy n -> IO (InjectIO n t) -> t
unsafePurify t makeDecorator = unsafePerformIO $ do
    decorate <- makeDecorator
    return $ compose p unsafePerformIO'
```

where we've locally specialized the types of unsafePerformIO and return to help GHC with inference, and made return strict\(^{13}\) to enforce correct sequencing of the unsafe IO.

The UnsafePurifiable is a constraint synonym capturing when it makes sense to compose with return and then with unsafePerformIO. Consistent with the theme, this is always satisfied in practice for concrete types \(n\) when \(n\) is sensible. The details:

```haskell
type UnsafePurifiable n t =
    (CurryUncurry n t
     , UncurryCurry n (Args n t) (IO (Ret n t))
     , UncurryMCurry n (IO (Ret n t))
     )

type UncurryCurry (n :: Nat) (as :: *) (r :: *) =
    (Curry as r
     , Uncurry n (as ->* r)
     , Args n (as ->* r) -> as
     , Ret n (as ->* r) -> r
     )

type UncurryMCurry (as :: *) (m :: * -> *) (r :: *) =
    (Curry as (m r)
     , UncurryM (as ->* m r)
     , ArgsM (as ->* m r) -> as
     , RetM (as ->* m r) -> r
     , MonadM (as ->* m r) -> m
     )
```

Where in turn, UncurryCurry and UncurryCurryM capture what it means for uncurrying after currying to make sense.

The unsafePurify takes a computation makeDecorator that makes a decorator, and not a decorator directly, so that state can be allocated. E.g., we can make a memoizer for pure functions, which allocates its own cache, with

```haskell
unsafeMemoize ::
    (UnsafePurifiable n t
     , Ord (Args n t)
     ) => Proxy n -> t -> t
unsafeMemoize p =
    unsafePurify p (memoize <$> newIORef Map.empty)
```

For unsafeTrace, it's actually more useful to allocate the state outside, so that it can be shared between multiple traced functions. So, we end up with a trivial makeDecorator:

```haskell
{-# NOINLINE levelRef #-}
levelRef :: IORef Int
levelRef = unsafePerformIO $ newIORef 0
unsafeTrace ::
    (UnsafePurifiable n t
     , Show (Args n t)
     ) => Proxy n -> t -> t
unsafeTrace n name =
    unsafePurify n (return $ trace levelRef name)
```

Finally, we can write a decorated non-monadic pow function:

```haskell
pow' :: Int -> Int -> Int
pow' = unsafeTrace n "pow" . unsafeMemoize n $ pow',
    where
      pow' b p =
        if p <= 1
          then b * p
          else pow' b (p 'div' 2) * pow' b (p - (p 'div' 2))
```

Whew! Evaluating \(pow 2 \ 6\) we see

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

Woo!

4. The Big Picture

In his paper we've limited ourselves to simple decorators. The definitions of currying and uncurrying are relatively complicated, but this is crucial to making our decorators widely applicable. However, we don't want to leave you with the impression that the decorators themselves need be simple. A few thoughts on some of the things we have already done, and things we'd like to do:

- Our original motivating example was a call-tracer that does not print out the arguments and results, but rather, builds up all the arguments, results, and recursive calls into a tree of heterogeneous (existentially quantified) data. This tree-of-data approach allows for arbitrary post processing, including simple printf-style tracing as we've shown here, but also more interesting post-processing:
  - Debugging: walk the tree interactively and inspect the data at each node.
  - Fancy formatting: produce a Graphviz graph of the call-trace, or a LaTeX proof tree.

We implemented a LaTeX proof tree backend, and instantiated it for a trivial type checker [1].

- An interesting problem which came up in implementing the tree-of-data logger was how to reuse heterogeneous data at multiple classes. If a data tree stores existentially quantified data representing function arguments and return values, how do we use the same tree to build several different traversals? We solved this problem by implementing implication between Haskell constraints, in a way that allows us to safely cast the data tree to different types for each of the different traversals. We summarize the main ideas in Appendix B.

- In this paper we've given a Haskell memoization decorator which is very close to the Python version, but in Haskell a more general decorator is probably preferable. We don't really want to restrict decorators to the IO monad. See Appendix C for a fancier version where the monad is more abstract, and we use a single cache of caches, so that we don't have to create and initialize a new cache for every function we decorate.

- A decorator we'd like to try, but have only sketched on paper and not implemented yet, is a decorator for automatic hash consing. The strategy we have in mind uses two-level types, the algebraic data type analog of open-recursion, to get inside the

\(^{13}\)This is really important!
recursive knot in the data. Two-level types have traditionally been painful in Haskell, “infecting” your whole program. However, using pattern synonyms in GHC 7.8, we hope to be able to implement a hash-consing decorator via two-level types in a way that only appears to affect code locally.

- There are many variations on memoizing pure functions. Here we gave an unsafePerformIO-based hack, but more principled side-effect based approaches include compiler support [6]. Alternatively, in Haskell as it exists today, one can use lazy evaluation and a possibly infinite data structure to implement a map which spans the complete domain of the memoized function, but only lazily computes the data structure as calls are made [5]. Conol Elliot describes some examples of this on his blog [4].

Finally, our combination of simple decorators and complicated primitives in this paper brings to mind a comment in Cabal’s Distribution.Simple module [7]:

This module isn’t called “Simple” because it’s simple. Far from it. It’s called “Simple” because it does complicated things to simple software.

Acknowledgments

This work was supported by NSF grant 0910500.

References


A. Appendix: Decorators in Python

In this section we derive the Python memoization decorator memoize, identifying the general concepts along the way. If you already understand the Python memoize, you can safely skip this section.

Recall the world’s most popular straw-man recursive function, Fibonacci. In Python:

```python
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

As everyone knows, this natural definition has exponential run time. We can make the run time linear via dynamic programming (with $O(1)$ additional space) or memoization (with $O(n)$ additional space). Here dynamic programming can beat memoization in space overhead by depending on details of the definition of the fib. Namely, the recursive calls in fib(n) are to fib(n-1) and fib(n-2), so dynamic programming can get by with two extra variables storing those values. In code:

```python
def dp_fib(n):
    f0, f1 = 0, 1
    for _ in range(n):
        f0, f1 = f1, f1 + f0
    return f0
```

On the other hand, memoization does not depend on the definition of fib, and just naively caches all previously computed results. In code:

```python
def memo_fib_cache = dict()
def memo_fib(n):
    if n not in memo_fib_cache:
        if n <= 1:
            r = n
        else:
            r = memo_fib(n-1) + memo_fib(n-2)
        memo_fib_cache[n] = r
    return memo_fib_cache[n]
```

where dict() creates an empty dictionary / hash table.

The memoization transformation didn’t depend on the definition of fib: for an arbitrary unary function f, the memoized version would be:

```python
def memo_f_cache = dict()
def memo_f(n):
    if n not in memo_f_cache:
        # Compute original definition of 'f'
        # with recursive calls replaced by 'memo_f'
        r = memo_f(n-1) + memo_f(n-2)
        memo_f_cache[n] = r
    return memo_f_cache[n]
```

Now, there is one obvious problem with capturing this transformation formally in code: how to implement the replacement of recursive calls? But, it turns out that there is a simple way to do this: a Python function name is just a mutable variable, and so is evaluated each time the function is called. So, for a recursive function f and a transformation on functions t, we can write f = t(f): the call-by-value argument f in t(f) evaluates to the current definition of f, and then f is made to refer to whatever t(f) returns. If t(f) captures the original value of f in a closure, then it can call the original f. However, recursive occurrences of f in the original definition of f are evaluated on each call, and so resolve to t(f)!

So then, we can define a memoization decorator for unary functions:

```python
def memoize(f):
    cache = dict()
    def memoized(n):
        if args not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Now, if we add f = memoize(f), then the original function f is captured in a closure memoized, along with a fresh cache. A call f now resolves to memoized(n), and when n is not in the cache, the captured original f is called to compute the requested result. If the original f makes any recursive calls, they resolve memoized! The point is that the treatment of function names as mutable variables gives an easy way to get inside the recursive knot and intercept recursive calls.
The final step is to generalize the memoization decorator from unary functions to functions of all arities, and give the memoized function the same name as the original function:

def memoize(f):
    cache = dict()
    def memoized(*args):
        if args not in cache:
            cache[args] = f(*args)
        return cache[args]
    return memoized

The *args syntax in a definition (formal parameter) means to collect all the arguments into a tuple; this is sometimes called a “variadic” function, and corresponds to the &rest args syntax in LISP.

The *args syntax in an expression (actual parameter) means to apply a function to a tuple of arguments; this corresponds to the apply function in LISP. These tupling and untupling transformations are analogous to currying and uncurrying, and motivate our use of those primitives in our Haskell implementation (Section 2.3).

B. Appendix: Constraint Implication

The original motivation for this work was a generic logger, which is too complicated to describe in this paper. However, in developing the generic logger we came across and solved the problem Haskell-constraint implication, and we expect our solution is generally useful when programming with heterogeneous data in Haskell. In this section we describe constraint implication and apply it to casting a simple heterogeneous container type H, which we make use of in several places in our implementation [3].

The heterogeneous wrapper type we use here is called $H$:

data $H$ $(c :: * \rightarrow \text{Constraint})$ where
$H : c a \Rightarrow a \rightarrow H c$

That is, an $H c$ value is a wrapped value of existentially quantified type which is known to satisfy the constraint $c$. To use an $H c$ value, we provide the higher-rank function $unH$:

$unH :: (\forall a. c a \Rightarrow a \rightarrow b) \rightarrow H c \rightarrow b$

$unH \ f \ (H \ x) = f \ x$

Note that this is the obvious “eliminator” for $H$, if we pretend that $\Rightarrow$ is a regular arrow.

In our actual use case, the generic logger, we have a recursive tree of existentially quantified data, with a uniform constraint over its contents. We have several type-classes which correspond to post-processing the tree in different ways, and so we want to constrain a given tree at several classes. However, each class requires itself to be the only constraint on the data tree, because of the recursion, and so we need a way to cast a tree constrained by multiple classes to trees constrained by each single class.

To capture multiple constraints as a single constraint, we define conjunction of constraints ($\& \&$):

$\text{infixr} \ : \& \&$

$\text{class} \ (c1 \ t, c2 \ t) \Rightarrow (c1 : \& \& : c2) \ t$

$\text{instance} \ (c1 \ t, c2 \ t) \Rightarrow (c1 : \& \& : c2) \ t$

Our goal is now to define a notion of constraint implication, $\text{Implies}$, such that e.g. $\text{Implies} \ (\text{Show} : \& \& : \text{Eq})$ is inhabited, and for which we can write a function for casting trees by implications. In this simplified presentation, the casting corresponds to $\text{coerceH}$:

$\text{coerceH} :: \forall c1 \ c2. \ \text{Implies} \ c1 \ c2 \Rightarrow H \ c1 \rightarrow H \ c2$

Towards these ends, we reify class constraints:

data $\text{Reify} \ c \ a \ where$
$\text{Reify} :: c \ a \Rightarrow \text{Reify} \ c \ a$

We then define $\text{Implies}$ by

type $\text{Implies} \ c1 \ c2 = \forall a. \ \text{Reify} \ c1 \ a \Rightarrow \text{Reify} \ c2 \ a$

Note that all concrete instances of $\text{Implies} \ c1 \ c2$ are simply

\[\text{\textbackslash case Reify} \Rightarrow \text{Reify}\]

Not bad!

For this definition of $\text{Implies}$, we can define $\text{coerceH}$ by

$\text{coerceH} :: \forall c1 \ c2. \ \text{Implies} \ c1 \ c2 \Rightarrow H \ c1 \rightarrow H \ c2$

$\text{coerceH} \ \text{impl} \ (H \ (x :: a)) =$

$\text{case impl} \ (\text{Reify} :: \text{Reify} \ c1 \ a) \ of$

$\text{Reify} \Rightarrow H \ x$

In the case of heterogeneous trees, called LogTree in our implementation, the definition of $\text{coerceLogTree}$ is similar to $\text{coerceH}$ in principle, but also includes mapping itself over the recursive subtrees.

In the next section we consider a memoizer which makes use of $H$, but not constraint implication.

C. Appendix: A More General Memoizer

In the intro we gave a memoization decorator specialized to $\text{IO}$, which received an I0Ref to a cache as one of its arguments. In practice, it’s more useful to support any monad which models mutable state, e.g. MonadIO, ST, and State. In this section we describe such a more general memoization decorator which can be instantiated at any mutable-state monad, and which shares a single cache across all memoized functions. In particular, this allows the user to allocate a single cache once, which is useful when the ambient monad is State.

The user supplies “lookup” and “insert” functions which manipulate a cache of caches: existentially quantified Typeable types keyed by strings. The existential quantification is provided by the type $H$, which we introduced above. The decorator allocates a Data.Map.Map under a user-specified string – in practice the module-qualified name of the memoized function, but any unique string will do – via the user-specified insert function. Because the Map must be Typeable, and is used to store cached results of the memoized function keyed by argument tuples for the memoized function, there are Typeable constraints on the domain and range of the memoized function:

$\text{castMemoize} :: \forall t. \ (\text{CurryUncurryM} \ t$

$\text{, Ord} \ (\text{ArgM} \ t)$

$\text{, Typeable} \ (\text{ArgM} \ t)$

$\text{, Typeable} \ (\text{RetM} \ t)$

$\text{, Functor} \ (\text{MonadM} \ t)) \Rightarrow$

$(\text{String} \rightarrow \text{MonadM} \ t \ (\text{Maybe} \ (\text{H} \ \text{Typeable})))$ $\rightarrow$

$(\text{String} \rightarrow \text{H} \ \text{Typeable} \rightarrow \text{MonadM} \ t \ ())$ $\rightarrow$

String $\rightarrow$

t $\rightarrow$ t

$\text{castMemoize} \ \text{lookup} \ \text{insert} \ \text{tag} \ f = \text{curry} \ \text{memoized where}$

$\text{memoized :: UncurriedM} \ t$

$\text{memoized} \ \text{args} \ = \ do$

$\text{cache} \leftarrow \text{getCache}$

$\text{case Map.lookup args cache of}$

Just ret $\rightarrow$ return ret

Nothing $\rightarrow$ do

ret $\leftarrow$ uncurryM f args

cache $\leftarrow$ getCache

insert tag $\$ (Map.insert args ret cache)$

return ret

short description of paper 9 2014/8/27
getCache :: MonadM t (Map.Map (ArgsM t) (RetM t))
getCache =
    maybe Map.empty (unH castCache) <$> lookup tag

castCache :: Typeable a => a -> Map.Map (ArgsM t) (RetM t)
castCache d = case cast d of
    Just cache -> cache
    Nothing -> error "castMemoize: Inconsistent cache!"

If we are in a MonadIO monad, then assuming

cacheRef :: IORef (Map.Map String (Maybe (H Typeable)))
we can instantiate castMemoize with

lookup args = do
    cache <- liftIO $ readIORef cacheRef
    return $ Map.lookup args cache

insert args r =
    liftIO $ modifyIORef cacheRef (Map.insert args r)

and similar for if we are in ST.

If we are in a MonadState (Map.Map String (Maybe (H Typeable)))
monad, then we can instantiate castMemoize with

lookup args = do
    cache <- get
    return $ Map.lookup args cache

insert args r =
    modify (Map.insert args r)