Functional Pearl: Ghosts of Departed Proofs

Matt Noonan
Kataskeue LLC
Ithaca, NY, USA
mnoonan@kataskeue.com

Abstract

We present a simple technique that allows library authors to control how APIs are used.

CCS Concepts • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Keywords Wireless sensor networks, media access control, multi-channel, radio interference, time synchronization

ACM Reference Format:
Matt Noonan. 2018. Functional Pearl: Ghosts of Departed Proofs. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, Article 39, 4 pages. https://doi.org/0000001.0000001

1 Introduction

1.1 Encoding with Universals

It is a theorem of both classical and constructive logics that

$$\forall t.\ (\forall s. \varphi(s) \Rightarrow t) \Rightarrow t \equiv \exists c.\ \varphi(c)$$

2 Warmup: Not quite dependent types

```
norm2 :: [Double] → Double
norm2 xs = sizing xs (\v → v `dot` v)
```

```
sizing xs $ \xs' →
  case align xs' ys of
    Just ys' → (xs' `dot` ys') / (xs' `dot`
   xs')
    Nothing  → 17
```


Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/0000001.0000001

```haskell
{-# LANGUAGE RankNTypes #-}
module Sized
  (Size, the, sZipWith, sizing, align) where

newtype Size n a = Size a

the :: Size n a → a
the (Size x) = x

sZipWith :: (a → b → c)
         → Size n [a]
         → Size n [b]
         → Size n [c]
sZipWith f xs ys =
  Size (zipWith f (the xs) (the ys))

sizing :: [a] → (forall n. Size n [a] → t) → t
sizing xs k = k (Size xs)

align :: Size n [a] → [b] → Maybe (Size n [b])
align xs ys = if length (the xs) == length ys
              then Just (Size ys)
              else Nothing
```

Figure 1. A small module defining a type for lists with a known length.

```haskell
import Sized

dot :: Size n [Double] → Size n [Double] → Double
dot xs ys = sum (the $ sZipWith (*) xs ys)

main :: IO ()
main = do
  xs ← readLn
  ys ← readLn
  sizing xs $ \xs' → do
    case align xs' ys of
      Nothing  → putStrLn "Size mismatch!"
      Just ys' → print (dot xs' ys')
```

Figure 2. A user-defined dot product function that can only be used on same-sized lists, and a usage example.

```haskell
class The d a | d → a where
    the :: d → a
    default the :: Coercible d a ⇒ d → a
    the = coerce

instance The (Size n a) a
```

Figure 3. The `The` typeclass, for dropping ghosts from a type. The default instance should always be used, so new instances can be created with an empty `instance` declaration.

Despite • appearences, the phantom type parameter *n* does not really represent the vector's length *per se*. Instead, we propose to think of `Size n` as a predicate, and values of type `Size n [a]` should be thought of as "lists of type `[a]`, equipped with a proof that they satisfy `Size n`". Critically, this proof has no run-time impact: it is trapped in the phantom type parameter.

This approach gives us a straightforward way to interpret the type signatures from example ***:

```haskell
-- You can take the dot product of two lists
  , if you have proven
-- that they have the same Size n.
dot :: Size n [Double] → Size n [Double] →
   Double

-- When you map a function over a list of
  Size n, the
-- result will also have Size n.
smap :: (a → b) → Size n [a] → Size n [b]

-- For any list, there is some n such that
  Size n is true.
sizing :: [a] → (∀ n. Size n [a] → t) → t

-- Given a list of Size n, we may be able to
   prove that
-- another list also has Size n.
align :: Size n [a] → [b] → Maybe (Size n
  [b])
```

As we attach increasingly sophisticated information into the phantom types, it becomes useful to have a uniform method for *forgetting* all of the ornamentation, revealing the normal value underneath.

3 Case Study #1: Sorted lists

Clients of the library are somewhat more restricted, in the sense that they cannot create a value of type `OrderedBy comp t` without going through the library's public API.

```haskell
module Sorted
  (Named, SortedBy, sortBy, mergeBy) where

import The
import Named

import qualified Data.List as L
import qualified Data.List.Utils as U

newtype SortedBy o a = SortedBy a
instance The (SortedBy o a) a

sortBy :: Named comp (a → a → Ordering)
       → [a]
       → SortedBy comp [a]
sortBy comp xs = coerce (L.sortBy (the comp) xs)

mergeBy :: Named comp (a → a → Ordering)
        → SortedBy comp [a]
        → SortedBy comp [a]
        → SortedBy comp [a]
mergeBy comp xs ys =
  coerce (U.mergeBy (the comp) (the xs) (the ys))
```

Figure 4. A module for working with lists that have been sorted by an arbitrary comparator.

```haskell
module Named (Named, name) where

import The

newtype Named name a = Named a
instance The (Named name a) a

name :: a → (forall name. Named name a → t) → t
name x k = k (coerce x)
```

Figure 5. A module for attaching ghostly names to values.

```haskell
import Sorted
import Named
main = do
  xs ← readLn :: IO Int
  ys ← readLn
  name (>) $ \gt → do
    let xs' = sortBy gt xs
        ys' = sortBy gt ys
    print (the xs', the ys', the (mergeBy gt xs' ys'))
```

Figure 6. Usage example

```
minimum_O1 :: SortedBy comp [a] → Maybe a
minimum_O1 xs = case (the xs) of
    []     → Nothing
    (x:_) → Just x
```

3.1 Conjuring a name

Finally, for the user to be able to *use* this library, there must be a way for them to create `Named` values from normal values. The library must export a function similar to this:

```
name :: a → (∀ name. Named name a → t) → t
name x k = k (coerce x)
```

This function is quite similar to `sizing` from the previous section, and the rank-2 type gives it a bit of an ominous feel. You might wonder: why not just have a function with a simple type like this?

```
any_name :: a → Named name a
any_name = coerce
```

The crux of the issue is all about *who gets to choose* what `name` will be. In the signature of `any_name`, the *caller* gets to select the types `a` and `name`. In particular, they can attach any name they would like!

If that still does not sound so bad, consider this code:

```
up, down :: Named () (Int → Int → Ordering)
up   = any_name (<)
down = any_name (>)

list1 = sortBy up   [1,2,3]
list2 = sortBy down [1,2,3]

merged = the (mergeBy up list1 list2) :: [Int]
-- [1,2,3,3,2,1]
```

Now compare to the analogous program, using `name` instead of `any_name`:

```
name (<) $ \up →
  name (>) $ \down →
    let list1 = sortBy up   [1,2,3]
        list2 = sortBy down [1,2,3]
    in the (mergeBy up list1 list2)
```

resulting in a compile-time error:

```
  • Couldn't match type "name1" with "name"
      ...
    Expected type: SortedBy name [Integer]
      Actual type: SortedBy name1 [Integer]
```

A general rule of thumb for library authors is: *a ghost should not appear in the return type, unless it also appears in an argument's type.* This simple rule ensures that the user of the library will not be allowed to materialize ghosts out of thin air.

4 Case Study #2: `Maybe`-free lookup in containers

4.1 Application: a type for directed graphs

```
data Neighbors phi = Neighbors
    { outEdges :: [JKey φ Vertex]
    , inEdges   :: [JKey φ Vertex] }

type Digraph φ = JMap φ Vertex (Neighbors φ)
```

```
addEdge   :: Vertex φ → Vertex φ → (forall φ'. Digraph φ' → t) → 
```

```
check  :: Int → Digraph φ → Either (FreshVertex φ) (Vertex φ)
fresh  :: Digraph φ → FreshVertex φ
addVertex :: FreshVertex φ → Digraph φ →
           (forall φ'. (Vertex φ', Vertex φ → Vertex φ', Digraph φ')
```

4.2 Faster lookup

Although `justified-containers` defines a simple `newtype` wrapper for the key-plus-phantom-proof type, more interesting information about the location of the key within the corresponding data structure can sometimes be attached.

For example, imagine a simple binary search tree backed by a vector of key-value pairs. As in the previous example, we will give the `BST` type a phantom parameter that represents the set of valid keys present in the tree. But instead of wrapping the key type directly, we will use an index-plus-phantom-proof representation for keys.

```
newtype BST φ k v = BST (Vector (k,v))
newtype Index φ    = Index Int

toBST :: Ord k ⇒ Vector (k,v) → BST φ k v

find   :: Ord k ⇒ k → BST φ k v → Maybe (
    Index φ)
access :: Index φ → BST φ k v → (k,v)
```

5 Case Study #3: Encoding arbitrary properties

```
nonzero_length_implies_cons
  :: (Length xs == Succ n)
```

```
newtype JMap φ k v = JMap (Map k v)
    deriving Functor

newtype JKey φ k = Element k

instance The (JMap φ k v) (Map k v)
instance The (JKey φ k)  k

member ::  k → JMap φ k v → Maybe (JKey φ k)

lookup   :: JKey φ k → JMap φ k v → v

reinsert
  :: JKey φ k → v → JMap φ k v → JMap φ k v

withMap
  :: Map k v  → (forall φ. JMap φ k v → t) → t
```

```
test_table = Map.fromList [ (1, "Hello")
                          , (2, "world!") ]
```

```
withMap test_table $ \table →
  case member 1 table of

    Nothing  → putStrLn "Missing key!"

    Just key → do
      putStrLn ("Found key: " ++ show (the key))
      putStrLn ("Value in map 1: " ++
                lookup key table)

      let table'  = reinsert key "Howdy" table
          table'' = fmap (map upper) table
      putStrLn ("Value in map 2: " ++
                lookup key table')
      putStrLn ("Value in map 3: " ++
                lookup key table'')
{- Output:
Found key: 1
Value in map 1: Hello
Value in map 2: Howdy
Value in map 3: HELLO
-}
```

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\mathtt{IsNil}(x) \land |x| = 0}\,(\mathrm{p})}{|x| = 0}
}{0 = |x|}
\quad
\dfrac{}{|x| = 1 + n}\,(\mathrm{eq})
}{0 = 1 + n}
\quad
\dfrac{\dfrac{}{\forall m.\ \neg(0 = 1 + m)}}{\neg(0 = 1 + n)}
}{\bot}
}{\mathtt{IsCons}(x)}
}{\mathtt{IsNil}(x) \land |x| = 0 \longrightarrow \mathtt{IsCons}(x)}\,(\mathrm{p})
\quad
\dfrac{
\dfrac{\dfrac{}{\mathtt{IsCons}(x) \land |x| = 1 + |\mathtt{Tail}(x)|}\,(\mathrm{q})}{\mathtt{IsCons}(x)}
}{\mathtt{IsCons}(x) \land |x| = 1 + |\mathtt{Tail}(x)| \longrightarrow \mathtt{IsCons}(x)}\,(\mathrm{q})
\quad
\dfrac{}{(\mathtt{IsNil}(x) \land |x| = 0) \lor (\mathtt{IsCons}(x) \land |x| = 1 + |\mathtt{Tail}(x)|)}
}{\mathtt{IsCons}(x)}
}{|x| = 1 + n \longrightarrow \mathtt{IsCons(x)}}\,(\mathrm{eq})
$$

Figure 7. A proof that lists with nonzero length satisfy the `IsCons` predicate, in natural deduction style. Compare with the same proof using the `Proof` monad in listing ****; the steps after the `(|/)` operator correspond to the leftmost deductions in this proof tree. Note a slight difference: the listing proves $|x| = 1 + n \vdash \mathtt{IsCons}(x)$, while the derivation in this figure proves $\vdash |x| = 1 + n \longrightarrow \mathtt{IsCons}(x)$.

```
  → Proof (IsCons xs)

nonzero_length_implies_cons eq =
  do  toSpec length
   |$ or_elimR and_elimL
   |/ and_elimR
   |. symmetric
   |. transitive' eq
   |. (contradicts' $$ zero_not_succ)
   |. absurd
```

Received June 2018