Safe APIs from Ghosts of Departed Proofs

Matt Noonan

September 26, 2018

Kataskeue LLC & Input Output HK

Play along at home!

```
git clone https://github.com/matt-noonan/gdp-talk
cd gdp-talk && stack ghci
```

[Rico Mariani] admonished us to think about how we can build platforms that lead developers to write great, high performance code such that developers just fall into doing the "right thing".

That concept really resonated with me. It is the key point of good API design.

We should build APIs that steer and point developers in the right direction.

— *Brad Abrams*

How can we make good APIs?

Narrowing the focus

Specifically, how can we make APIs that are:

- Safe.

  Incorrect use of the API should result in an error at compile-time.

- Ergonomic.

  Correct use of an API should not place an undue burden on the user.

What is the goal?

In this talk, we will investigate how

- existentially-quantified type-level names for values,
- theorems as phantom types, and
- safe coercions

can combine to provide a safe and ergonomic strategy for designing APIs with complex requirements.

Unsafe idiom: explode

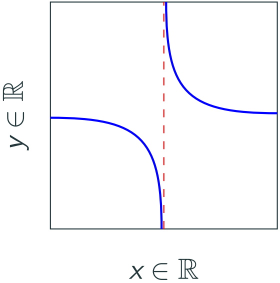
```
mnoonan@euclid:~/gdp-talk/tmp$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/mnoonan/.ghci
λ> head []
*** Exception: Prelude.head: empty list
λ>
```

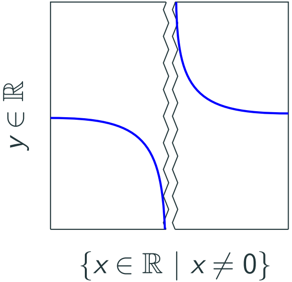
Unsafe idiom: anything goes

```
mnoonan@euclid:~/gdp-talk/tmp$ pygmentize head.cpp \
> && g++ head.cpp && echo "// Result:" && ./a.out
#include <iostream>
#include <vector>

int main() {
  std::vector<int> x{1};
  x.pop_back();
  std::cout << "Size is " << x.size() << ", "
            << "head is " << x[0] << std::endl;
  return 0;
}
// Result:
Size is 0, head is 1
mnoonan@euclid:~/gdp-talk/tmp$
```

Safe idiom: use refinement types to restrict domain

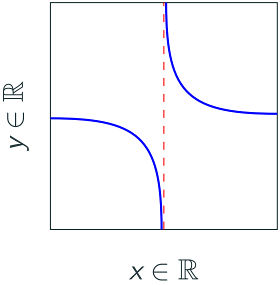
```haskell
data NonEmptyList a = NonEmptyList a [a]

headNE :: NonEmptyList a → a
headNE (NonEmptyList x xs) = x
```

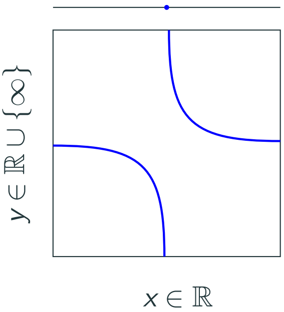




Safe idiom: use option types to expand range

```
headMay :: [a] → Maybe a
headMay = \case
    []     → Nothing
    (x:xs) → Just x
```





Safe idiom: say what you mean with dependent types

```
head : ∀ {A n} → Vec A (1 + n) → A

zip : ∀ {A B n} → Vec A n → Vec B n → Vec (A × B) n

take : ∀ {A} m {n} → Vec A (m + n) → Vec A m
```

Safe idiom: say what you mean with dependent types

```
head : ∀ {A n} → Vec A (1 + n) → A

zip : ∀ {A B n} → Vec A n → Vec B n → Vec (A × B) n

take : ∀ {A} m {n} → Vec A (m + n) → Vec A m
```

(...but what properties should be reflected in the type?)

A case study in API design

A finicky API for merging and sorting

```haskell
sortBy  :: (a -> a -> Ordering) -> [a]          -> [a]
mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]

-- BE CAREFUL! xs and ys must already be sorted by comp!
mergeBy comp xs ys = case (xs, ys) of
    (_, []) -> xs
    ([], _) -> ys
    ((x:xs'), (y:ys')) -> case comp x y of
        LT -> x     : mergeBy comp xs' ys
        GT ->     y : mergeBy comp xs  ys'
        EQ -> x : y : mergeBy comp xs' ys'
```

Can we make it safe with optional types?

```haskell
module FancySafeMerge where

mergeMay :: (a → a → Ordering) → [a] → [a] → Maybe [a]

mergeMay comp xs ys =
    if isSorted xs && isSorted ys
      then Just (mergeBy comp xs ys)
      else Nothing

  where
    isSorted (z : zs@(z' : _)) =  comp z z' /= GT
                                  && isSorted zs
    isSorted _ = True
```

Maybe more like this...


L. Boilly 1824.
I. lith. de Delpech.

Leading the user into sin and vice

```haskell
import FancySafeMerge

myMergeDown :: [Int] → [Int] → [Int]
myMergeDown xs ys =
    let comp = comparing Down
        xs' = sortBy comp xs
        ys' = sortBy comp ys

    in  fromJust (mergeMay comp xs' ys')
```

Why is the user frustrated?

Why is the user frustrated?

- The library API demands a precondition is met.

Why is the user frustrated?

- The library API demands a precondition is met.
- The library forces error checks by returning option types.

Why is the user frustrated?

- The library API demands a precondition is met.
- The library forces error checks by returning option types.
- The user correctly ensured that the precondition held.

Why is the user frustrated?

- The library API demands a precondition is met.
- The library forces error checks by returning option types.
- The user correctly ensured that the precondition held.

...so why are we making them check the `Nothing` case?!

Frustration in the wild

A rough `grep` of Hackage finds over 2000 cases where `lookup :: k → Map k v → Maybe v` is followed by `fromJust`.

`lookup` tries to be virtuous by ensuring that the user handles the "missing key" case.

But what recourse is there for the gallant user who already proved that the key is present?

Can we do better?

We want two-way communication between user and library!

The library author wants to tell the user "this function can only be used when condition X holds".

The library user wants to tell the library "I have ensured that X holds, so please let me use the function `Maybe`-free".

We want two-way communication between user and library!

The library author wants to tell the user "this function can only be used when condition X holds".

The library user wants to tell the library "I have ensured that X holds, so please let me use the function `Maybe`-free".

Problem: How can we reflect constraints on the function's input *values* into the functions's *type*?

Key idea #1: phantom type-level names for values

```haskell
module Named (name, type (~~)) where
                                -- ^ Constructor NOT exported!
import Data.Coerce

newtype a ~~ name = Named a

-- Forgetting names
instance The (a ~~ name) a where
    the = coerce :: (a ~~ name) → a

-- Introducing names
name :: a → (forall name. (a ~~ name) → t) → t
--   :: a →  exists name. (a ~~ name)
name x cont = cont (coerce x)
```

Key idea #1: phantom type-level names for values

```
module Named (name, type (~~)) where
                         -- ^ Constructor NOT exported!
import Data.Coerce

newtype a ~~ name = Named a

-- Forgetting names
instance The (a ~~ name) a where
    the = coerce :: (a ~~ name) → a

-- Introducing names
name :: a → (forall name. (a ~~ name) → t) → t
--   :: a →  exists name. (a ~~ name)
name x cont = cont (coerce x)
```

Key idea #1: phantom type-level names for values

```haskell
module Named (name, type (~~)) where
                          -- ^ Constructor NOT exported!
import Data.Coerce

newtype a ~~ name = Named a

-- Forgetting names
instance The (a ~~ name) a where
    the = coerce :: (a ~~ name) → a
```

```haskell
-- Introducing names
name :: a → (forall name. (a ~~ name) → t) → t
--   :: a →  exists name. (a ~~ name)
name x cont = cont (coerce x)
```

Key idea #2: predicates as **newtypes** + phantom types

```haskell
module GDP.Merge (SortedBy, mergeGDP, sortGDP) where
                  -- ^ constructor NOT exported!

-- A `SortedBy comp a` is an `a` that
-- has been sorted by `comp`.
newtype SortedBy comp a = SortedBy a

instance The (SortedBy comp a) a

-- How do we get a `SortedBy comp [a]`?
-- By sorting a list using a comparator named `comp`!
sortGDP :: ((a → a → Ordering) ~~ comp)
        → [a]
        → SortedBy comp [a]

sortGDP comp = coerce . sortBy (the comp)
```

Key idea #2: predicates as `newtypes` + phantom types

```haskell
module GDP.Merge (SortedBy, mergeGDP, sortGDP) where
                -- ^ constructor NOT exported!

-- A `SortedBy comp a` is an `a` that
-- has been sorted by `comp`.
newtype SortedBy comp a = SortedBy a
```

```haskell
instance The (SortedBy comp a) a
```

```haskell
-- How do we get a `SortedBy comp [a]`?
-- By sorting a list using a comparator named `comp`!
sortGDP :: ((a -> a -> Ordering) ~~ comp)
        -> [a]
        -> SortedBy comp [a]

sortGDP comp = coerce . sortBy (the comp)
```

Key idea #2: predicates as `newtypes` + phantom types

```haskell
module GDP.Merge (SortedBy, mergeGDP, sortGDP) where
                -- ^ constructor NOT exported!

-- A `SortedBy comp a` is an `a` that
-- has been sorted by `comp`.
newtype SortedBy comp a = SortedBy a

instance The (SortedBy comp a) a
```

```haskell
-- How do we get a `SortedBy comp [a]`?
-- By sorting a list using a comparator named `comp`!
sortGDP :: ((a -> a -> Ordering) ~~ comp)
        -> [a]
        -> SortedBy comp [a]

sortGDP comp = coerce . sortBy (the comp)
```

How to communicate facts and requirements to the user

```haskell
-- This type reads as:
-- "mergeGDP takes a comparator and two lists that
-- have been sorted by that same comparator.
--
-- It returns a new list that is also sorted by that
-- same comparator."

mergeGDP :: ((a → a → Ordering) ~~ comp)
         → SortedBy comp [a]
         → SortedBy comp [a]
         → SortedBy comp [a]

mergeGDP comp xs ys =
    coerce (mergeBy (the comp) (the xs) (the ys))
```

How to communicate due-dilligence back to the library

```haskell
import FancySafeMerge

myMergeDown :: [Int] → [Int] → [Int]
myMergeDown xs ys =
    let comp = comparing Down
        xs' = sortBy comp xs
        ys' = sortBy comp ys

    in  fromJust (mergeMay comp xs' ys')
```

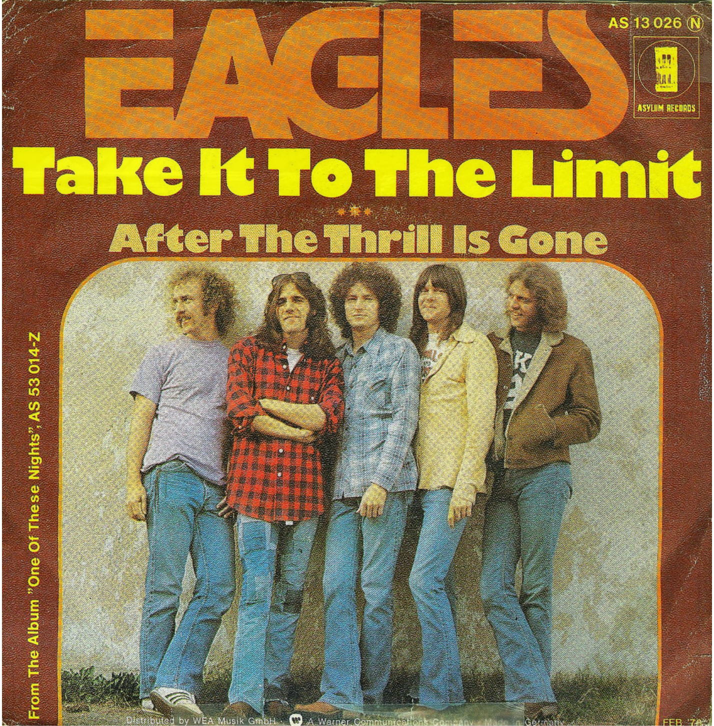
How to communicate due-dilligence back to the library

```haskell
import GDP.Merge

myMergeDown :: [Int] → [Int] → [Int]
myMergeDown xs ys =
  name (comparing Down) $ \comp →
    let xs' = sortGDP comp xs
        ys' = sortGDP comp ys

    in      the  (mergeGDP comp xs' ys')
```

Key idea #3: ghosts disappear on compilation

```
-- RHS size: {terms: 11, types: 6, coercions: 0, joins: 0/0}
Main.myMergeDown1 :: Int → Int → Ordering
Main.myMergeDown1
  = \ (x_a3eb :: Int) (y_a3ec :: Int) →
      case y_a3ec of { ghc-prim-0.5.2.0:GHC.Types.I# x#_a536 →
      case x_a3eb of { ghc-prim-0.5.2.0:GHC.Types.I# y#_a53b →
      ghc-prim-0.5.2.0:GHC.Classes.compareInt# x#_a536 y#_a53b
      }
      }

-- RHS size: {terms: 10, types: 7, coercions: 0, joins: 0/0}
myMergeDown :: [Int] → [Int] → [Int]
myMergeDown
  = \ (xs_a1zh :: [Int]) (ys_a1zi :: [Int]) →
      Data.List.Utils.mergeBy
        @ Int
        Main.myMergeDown1
        (base-4.11.1.0:Data.OldList.sortBy @ Int Main.myMergeDown1 xs_a1zh)
        (base-4.11.1.0:Data.OldList.sortBy @ Int Main.myMergeDown1 ys_a1zi)
```

Ghosts of Departed Proofs


EAGLES
AS 13 026 (N)
ASYLUM RECORDS
Take It To The Limit
After The Thrill Is Gone
From The Album "One Of These Nights", AS 53 014-Z
Distributed by WEA Musik GmbH · A Warner Communications Company · Made in Germany

Maybe-free lookup in maps

```
-- Ghosts of departed key sets
newtype JMap keys k v = JMap (Map k v)
newtype k ∈ keys      = Key k

-- Key search, avoiding boolean blindness
member :: k → JMap keys k v → Maybe (k ∈ keys)
member k m = if Map.member k (the m)
                then Just (coerce k)
                else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Maybe-free lookup in maps

```
-- Ghosts of departed key sets
newtype JMap keys k v = JMap (Map k v)
newtype k ∈ keys      = Key k

-- Key search, avoiding boolean blindness
member :: k → JMap keys k v → Maybe (k ∈ keys)
member k m = if Map.member k (the m)
                then Just (coerce k)
                else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Maybe-free lookup in maps

```
-- Ghosts of departed key sets
newtype JMap keys k v = JMap (Map k v)
newtype k ∈ keys      = Key k

-- Key search, avoiding boolean blindness
member :: k → JMap keys k v → Maybe (k ∈ keys)
member k m = if Map.member k (the m)
               then Just (coerce k)
               else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Maybe-free lookup in maps

```
-- Ghosts of departed key sets
newtype JMap keys k v = JMap (Map k v)
newtype k ∈ keys      = Key k

-- Key search, avoiding boolean blindness
member :: k → JMap keys k v → Maybe (k ∈ keys)
member k m = if Map.member k (the m)
                then Just (coerce k)
                else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Working with a more complex API

```
-- Ghostly predicates
data Keys m
data x ∈ s

-- Key search, avoiding boolean blindness
member :: (k ~~ key)
       → (Map k v ~~ m)
       → Maybe (k ~~ key ::: key ∈ Keys m)
member k m = if Map.member k (the m)
                then Just (coerce k)
                else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Three

Four

Five

Six

Seven

Examples

Hashed data structures

```haskell
newtype HashOf x = HashOf Defn

realHash :: Serializable a ⇒  a        →  Hash
hash     :: Serializable a ⇒ (a ~~ x) → (Hash ~~ HashOf x)

hash x = defn (realHash (serialize $ the x))

-- A type for objects along with their hash.
data ThingWithHash a = forall x. ThingWithHash
  { _thing :: a    ~~ x
  , _hash  :: Hash ~~ HashOf x }

-- Use it like this:
hashIt :: Serializable a ⇒ a → ThingWithHash a
hashIt x = name x $ \x' →
  ThingWithHash { _thing = x', _hash = hash x' }
```


SHARING
IS CARING
imgflip.com

The ST monad, with shared memory regions

```
-- Running an ST computation with shared regions
runSt2  ::
  STRef (forall mine yours. ST (mine ∩ yours) a) → a

inMine  :: ST mine  a → ST (mine ∩ yours) a
inYours :: ST yours a → ST (mine ∩ yours) a

-- Sharing an STRef we own
share  :: STRef mine a → ST mine (STRef (mine ∩ yours) a)

-- Using an STRef that was shared with us
use    :: STRef (mine ∩ yours) a → STRef mine a

-- Algebraic lemmas
symm   :: STRef (mine ∩ yours) a → STRef (yours ∩ mine) a
```

Maybe-free lookup in maps

```
-- Ghosts of departed key sets
newtype JMap keys k v = JMap (Map k v)
newtype k ∈ keys      = Key k

-- Key search, avoiding boolean blindness
member :: k → JMap keys k v → Maybe (k ∈ keys)
member k m = if Map.member k (the m)
               then Just (coerce k)
               else Nothing

-- Maybe-free lookup
lookup :: (k ∈ keys) → JMap keys k v → v
lookup k m = Map.lookup (the k) (the m)

-- A safe adjacency-list type for directed graphs
type Digraph k v = JMap keys k [k ∈ keys]
```

Two

Three

Summary

API design with Ghosts of Departed Proofs

- Use existentially-quantified names to discuss values at the type level.
- Avoid boolean blindness by returning proofs to the user.
- Avoid runtime overhead by putting proofs in phantom types.
- Give the user combinators for reasoning about proofs.

Try it out!

- The `gdp` library is on Hackage and Stackage
  `http://hackage.haskell.org/package/gdp`
- The paper's repo has smaller examples that are easy to play with in `ghci` (see the `gdp-demo` subdirectory)
  `https://github.com/matt-noonan/gdp-paper`
- `chessai` implemented the "ST-with-sharing" code in the `st2` library (also on Hackage)
  `https://github.com/chessai/st2`

Thank you for your time!