Ghosts of Departed Proofs (Functional Pearl)
Matt Noonan
Kataskeue LLC, Input Output HK
Ithaca, NY, USA
mnoonan@kataskeue.com

Abstract
Library authors often are faced with a design choice: should a function with preconditions be implemented as a partial function, or by returning a failure condition on incorrect use? Neither option is ideal. Partial functions lead to frustrating run-time errors. Failure conditions must be checked at the use-site, placing an unfair tax on the users who have ensured that the function’s preconditions were correctly met.

In this paper, we introduce an API design concept called “ghosts of departed proofs” based on the following observation: sophisticated preconditions can be encoded in Haskell’s type system with no run-time overhead, by using proofs that inhabit phantom type parameters attached to newtype wrappers. The user expresses correctness arguments by constructing proofs to inhabit these phantom types. Critically, this technique allows the library user to decide when and how to validate that the API’s preconditions are met.

The “ghosts of departed proofs” approach to API design can achieve many of the benefits of dependent types and refinement types, yet only requires some minor and well-understood extensions to Haskell 2010. We demonstrate the utility of this approach through a series of case studies, showing how to enforce novel invariants for lists, maps, graphs, shared memory regions, and more.

CCS Concepts • Software and its engineering → Formal software verification; • Theory of computation → Logic and verification;

Keywords API design, software engineering, formal methods, higher-rank types

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Haskell ’18, September 27–28, 2018, St. Louis, MO, USA © 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-5835-4/18/09... $15.00
https://doi.org/10.1145/3242744.3242755

1 Introduction

[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 [1]

What is the purpose of a powerful type system? One practical perspective is that a type system provides a mechanism for enforcing program invariants at compile time. The desire to encode increasingly sophisticated program invariants has led to a vast expanse of research on more complex type systems, including dependent types [2, 3], refinement types [6], linear types [22], and many more. But despite this menagerie of powerful type systems, workaday Haskell programmers have already been able to encode surprisingly sophisticated invariants using nothing more than a few well-understood extensions to the Damas-Hindley-Milner type system.

An early success story is the ST monad, which allows pure computations to make use of local, mutable state. A phantom type parameter and a clever use of rank-2 types in the ST monad’s API gives a compile-time guarantee that the local mutable state is invisible from the outside, and hence the resulting computation really is pure. As we will see, this trick is just the tip of a rather large iceberg.

In this paper, we will take the perspective of a library author, writing in Haskell 2010 (plus a few battle-tested language extensions). As a library author, our goal will be to design safe APIs that are also ergonomic for the end user. “Safe” means that we want to prevent the user from causing a run-time error. “Ergonomic” means that the correct use of our API must not place an undue burden on the user.

1.1 Common Idioms for Handling Pre-conditions

No matter the language, a programmer often has to write functions that place constraints on their input. For example, the venerable head function will extract the first element of a list, but asks its users to only give it a non-empty list to operate on. Now put yourself in the shoes of head’s author: how can you ensure that head will be used properly? Let us recount a variety of strategies used in the wild.
Figure 1. Idioms for implementing the head function, along with usage examples. The gdpHead function can only be invoked by presenting a proof that the list is non-empty, combining the simplicity of the first example with the safety of the second. rev_cons is a proof combinator exported by the library to help the user prove that the reverse of a non-empty list is also non-empty. See section 5 for details.

Run-time failure on bad inputs. The simplest approach is to have a function just fail on malformed inputs. The failure mode can be an immediate run-time error (as in head from Figure 1), an exception, or undefined behavior (as in C++’s std::vector<T>::front()).

Returning a dummy value. To avoid run-time errors, some APIs may have a “dummy value” for indicating the result of a failed operation. For example, Common Lisp’s car and goolang’s Front() both return nil when passed an empty list. The caller must explicitly check for this dummy value. Other contortions may be needed if the container is also allowed to hold nil, to disambiguate between “the input list is empty” and “nil is the first element of this list”.

Returning a value with an option type. A related strategy for languages with stricter typing discipline is to use an “option type,” such as Haskell’s Maybe or Scala’s Option. A value of type Maybe T cannot be used where a value of type T was expected, so the user must explicitly pattern match on the optional value to extract the result and handle the error case. This approach may lead to frustration when the user believes that the error case is not possible, as when headMay is applied to the reverse of a non-empty list in Figure 1.

Modifying input types to exclude bad inputs. Finally, the API designer may select more restrictive types for the inputs in order to make the function total. For example, some Haskell libraries make use of the NonEmpty type for lists that contain at least one element. The head function then becomes total. The user can prove that their list is non-empty by making use of the smart constructor nonEmpty :: [a] -> Maybe a. The drawbacks include duplication (do we re-implement length for NonEmpty?) and awkwardness when encoding preconditions that relate several inputs (e.g. requiring two lists to have the same length).

1.2 Leading the User into Temptation
The “return-anoptionalvalue” idiom is well-known and popular in the functional programming world. The author of a library function that returns Maybe a can certainly sleep well at night, content in the knowledge that their function will never cause a run-time error.

But what about the users of that library? Has the library author helped the user stay on a virtuous path, or have they led the user into temptation?

In fact, the author of the library has merely pushed extra responsibility onto the user. Every time the user applies a function that uses the optional-return idiom, they are obliged to test the return value and handle the error case. Even worse, the user is still asked to handle the error case when they have correctly ensured that the function’s preconditions have been met! The library author sleeps well, while even the most vigilant users are forced to toil against those impossible error cases.

No wonder so many well-meaning users reach for unsafe functions like fromJust! They have already proved (to their
own satisfaction) that the function is being used properly, so they rightly feel justified in ignoring the error case entirely. But now we see how the user has been led into a pit of despair: they have ended up with a program that is exactly as fragile as one where the library author had used the run-time-failure idiom! Even if the user has mentally constructed a proof that this specific use of \texttt{fromJust} is safe now, who can say what will happen as the software changes over time? Without tooling to ensure that the user’s proof remains valid, the software is left in a brittle state.

For example, a recent snapshot of hackage reveals over 2000 instances where the partial function \texttt{fromJust} is applied to the result of \texttt{Data.Map’s lookup}. Any one of these instances may be a vignette of a programmer falling into a pit of despair: they had a mental proof that a certain key must be present in the map, but possessed no mechanism for communicating that proof to the \texttt{lookup} function. In frustration, they made the pragmatic—but unsafe—decision to introduce partiality.

1.3 Who Is to Blame?

It would be easy to lay the blame at the foot of the the user. After all, they were the ones who brought in partial functions! But this perspective misses the point: when we return a \texttt{Maybe}, even a perfect user who has done their due diligence will be forced to handle an error case—exactly the error case that they were so careful to avoid! The real problem is that the conversion from a partial function to a \texttt{Maybe}-returning function is a bit of a cheat on the part of the library author. Instead of adding \texttt{Nothing} to a function’s codomain, why not simply restrict the function’s domain to the set of valid inputs? The user would still be responsible for ensuring that the inputs are valid but, having done so, they would not be asked to introduce a spurious error handler.

1.4 An Alternative: Ghosts of Departed Proofs

In the following sections, we will elaborate a design concept for creating libraries that supports a dialogue between library and user: the library can require that certain conditions are met, and the user can explain how they have met those obligations. The key features of this approach are as follows:

**Properties and proofs are represented in code.** Proofs are concrete entities within the host language, and can be analyzed or audited independently. In the tradition of the Curry-Howard correspondence, propositions are represented by types, and the proof of a proposition will be a value of that type.

**Proofs carried by phantom type parameters.** To ensure that proof-carrying code does not have a run-time cost, proofs will only be used to inhabit types that appear as phantom type variables attached to newtype wrappers. The newtype wrapper is erased during compilation, leaving no run-time cost and no evidence of these proofs in the final executable. The phantom type parameter is only used as a mechanism for transmitting the “ghost of a departed proof” to the library API. The name “ghost proof” is meant to suggest the related concept of ghost variables in software verification [10], and to emphasize the idea that the proof is non-corporeal: no artifacts related to the proof should ever be discernible from the compiler’s output.

**Library-controlled APIs to create proofs.** Library authors should retain control over how domain-relevant proofs can be created. That is, the library author should be the only one able to introduce new axioms about the behavior of their API. This may mean exporting functions that create values with known properties, or that classify a value into mutually disjoint refinements, or that introduce existentially-quantified properties (name in Figure 2, \texttt{runSt} in Figure 5, or \texttt{withinMap} in Figure 8).

**Combinators for manipulating ghost proofs.** Libraries may export a selection of combinators so that the user can mix and match the evidence at hand to produce a satisfactory proof of a safety property. The goal is to enrich the vocabulary of the user, so that they can productively communicate their proofs to the library.

1.5 The Structure of This Paper

In this paper, we will use a series of case studies to show how library authors can use ghosts of departed proofs (GDP) to create APIs that are both safe and ergonomic: the user cannot cause a run-time error when using the API, and incorrect uses of the API will become compile-time errors. But the APIs must be straightforward enough that the user is not tempted to subvert the library’s safety guarantees by using unsafe functions. Crucially, we want the user to be able to communicate their informal proofs to the library. If the user believes that a precondition has been met, they should be able to explain why to the library!

The GDP design concept is relatively simple to implement. Each case study includes example library code, along with usage demonstrations. The examples in this paper are self-contained, and are bundled together in a project suitable for further experimentation [13]. The proof combinators and other machinery from Section 5 are available as the gdp library on Hackage [14].

1.6 A Very Short Tutorial on Safe Coercions

Several of the examples in this paper rely on a basic understanding of safe coercions, a relatively recent addition to GHC Haskell [4]. The details of safe coercions are a bit technical, but for the purposes of this paper it suffices to know the following operational facts:

- The types \( \text{T} \) and newtype \( \text{N = N T} \) have the same run-time representation.
- \text{coerce : Coercible a b => a -> b} can be used as a zero-cost safe cast from \( a \) to \( b \), whenever the \text{Coercible a b} constraint is satisfied.
We will make repeated use of this last property to help enforce encapsulation. Suppose a library author creates a module that defines N as a newtype of T, but does not export the constructor. Then the library author can use coerce to freely cast between T and N, but users of that library only see N as an opaque type, and are not able to coerce it to T.

2 Case Study #1: Sorted Lists

It is almost inevitable that a programmer will, at some point, be asked to work with lists that have been sorted in one way or another. To ensure correctness, the programmer may need to carefully manage various invariants, such as “all of these lists must have been sorted by the same comparator”. For a concrete example, consider these sortBy and mergeBy functions:

```haskell
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeBy comp xs ys = go xs ys
where
  go [] ys' = ys'
  go xs' [] = xs'
  go (x:xs') (y:ys') = case comp x y of
    GT -> y : go (x:xs') ys'
    _   -> x : go xs' (y:ys')
```

This efficient $O(n + m)$ implementation of mergeBy is easy to write, but it comes with a hidden cost to the end user. Anybody who uses mergeBy must ensure that the two input lists have been sorted by the same comparator. If the user accidentally fails to sort the two inputs, or does not sort them in the same way, mergeBy will quietly produce nonsense and introduce a subtle bug.

It would be possible to implement a version of mergeBy that carefully inspected the inputs xs and ys as it proceeded, and only produced a result if the inputs met the sorting requirement. But this would impose a runtime cost on every use of mergeBy, increase the complexity of its implementation, and change the result type to Maybe [a]. And then what? Most users of mergeBy would argue to themselves “This is absurd! I already know that I sorted the input lists properly. This function will never result in Nothing.” It would be hard to blame the user when they reach for an unsafe function like fromJust.

Clearly, everybody loses out in the above scenario. The library author is inconvenienced by the increased implementation complexity. The user is inconvenienced by the decreased performance and the need to pattern match on the result, even when they already know the outcome of that match. No wonder that the status quo is to prominently display a stern warning in the documentation, admonishing any user who tries to mergeBy what they didn’t sortBy.

But what if the user really does have proof that the input lists have been sorted properly? Can we devise a mechanism that allows the user to communicate this proof to mergeBy?

2.1 Conjuring a Name

The first challenge is how to express the idea of two comparators being “the same”. In a language that supports equality tests on functions, you could imagine a solution where the sortBy function returns both the sorted list and a reference to the comparator that was used; mergeBy could then check that the comparators matched. But this has a run-time cost for carrying around the comparator references, and it still would require mergeBy to return Nothing if it was given bogus arguments.

A different solution, in line with the GDP concept, is to introduce a newtype wrapper equipped with a phantom type parameter name. In code, we will write this wrapper as a $\sim n$, to be read as “values of type a with name n”. To ensure that there is no run-time penalty for using names, a $\sim n$ is implemented as a newtype around a, with a phantom type parameter n. A simple module for named values can be found in Figure 2; the key feature is the exported name function that expresses the concept “any value can be given a name”.

To emulate an existentially-quantified type in Haskell, we will have to jump through a small hoop with name. Instead of directly returning a value with a name attached, name says to the user “tell me what you wanted to do with that named value, and I’ll do it for you”. This slight-of-hand is responsible for the rank-2 signature of name. The user must hand name a computation that is entirely agnostic about the name that will be chosen. More on this point in section 2.4.

Once we have introduced names, it becomes handy to have a uniform way of stripping names and other phantom
module Sorted (Named, SortedBy, sortBy, mergeBy) where

import The
import Named

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

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

sortBy :: ((a -> a -> Ordering) ~~ comp) -> [a] -> SortedBy comp [a]
sortBy comp xs = coerce (L.sortBy (the comp) xs)

mergeBy :: ((a -> a -> Ordering) ~~ comp) -> SortedBy comp [a] -> SortedBy comp [a] -> SortedBy comp [a]
mergeBy comp xs ys =
  coerce (U.mergeBy (the comp) (the xs) (the ys))

Figure 3. A module for working with lists that have been sorted by an arbitrary comparator. The refinement SortedBy comp is used to denote values that have been sorted by the comparator named comp.

data from a value. We do this with a simple two-parameter typeclass, like so:

class The d a | d -> a where
  the :: d -> a
  default the :: Coercible d a => d -> a
  the = coerce

By using this default signature for the most instances of The can be declared with an empty body:

instance The (a ~~ name) a

The default method’s use of a safe coercion helps ensure that forgetting a value’s name incurs no run-time cost.

2.2 Implementing a Safe API for Sorting and Merging

Now that we know how to attach ghostly names to values, we can tackle the design of a safe and ergonomic interface to mergeBy. In Figure 3, we begin by defining a newtype wrapper SortedBy comp that represents the predicate "x has been sorted by the comparator named comp". The wrapper’s meaning is imbued by the type of sortBy, which takes a named comparator and a list, and produces a list that has been SortedBy comp. Note that by not exporting SortedBy’s constructor, we have ensured that the only way to obtain a value of type SortedBy comp [a] is through the sortBy or mergeBy functions. The user is not allowed to assert that a list is SortedBy comp by fiat.

The implementation is straightforward enough: we use the to coerce away the name of the comparator, apply the simpler version of sortBy from Data.List, and then introduce the SortedBy comp predicate by coercing the result. Since the coercions have no run-time effect, the code generated by the compiler for our GDP-style sortBy is simply a call to Data.List.sortBy!

Similarly, the generated code for our mergeBy will just call the "normal" mergeBy. But notice the argument types of the GDP-style mergeBy in Figure 3. The user must hand mergeBy a named comparator, plus two lists that have been sorted by that very same comparator. No stern warnings in the documentation are required: if the user tries to mergeBy what they didn’t sortBy, the program will simply fail to compile!

We have successfully developed a safe API for sortBy and mergeBy, but how ergonomic is it? A usage example appears in Figure 4. The program is almost identical to one that uses the standard versions of sortBy and mergeBy, except for the line where we attach a ghostly name to comparing Down. We are asking very little more from the user, yet end up with an API that cannot be used incorrectly.

2.3 Applications to User Code

Although the library author retains control over the introduction of ghost proofs, the user is still able to leverage these proofs for their own purposes, beyond the library author’s original design. For example, the user can write a simple function that extracts the minimal element of a list with respect to a given comparator:

minimum_O1 :: SortedBy comp [a] -> Maybe a
minimum_O1 xs = case the xs of
  [] -> Nothing
  (x_:_) -> Just x

Thanks to the meaning given to SortedBy comp by the Sorted API, this user-defined function offers a strong guarantee that it can only be called on a sorted list. Despite being user-defined, this function cannot be used incorrectly. Did
you forget to sort the list before calling `minimum_0`? Then your program will not compile.

### 2.4 Aside: On the Danger of Naming a Ghost

Let us return for a moment to the somewhat unusual type of name in Figure 2. Is all of this business about rank-2 types really necessary, or is it merely ivory tower bloat? You may well wonder, why not just have a function with a simple type like this:

```haskell
any_name :: a -> (a ~ name)
any_name = coerce
```

At its core, the question is really 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:

```haskell
data Simon

up, down :: (Int -> Int -> Ordering) ~ Simon
up = any_name compare
down = any_name (comparing Down)

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]
```

The user has decided to name two different functions `Simon`, subverting the guarantees offered by the API of the `Sorted` module. It is dangerous to name a ghost!

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

```haskell
name compare $ \up ->
  name (comparing Down) $ \down ->
  let list1 = sortBy up [1,2,3 :: Int]
  list2 = sortBy down [1,2,3]
  in the (mergeBy up list1 list2)
```

Attempting to compile this program results in a type error:

```
Could't match type "name1" with "name"
...
Expected type: SortedBy name [Int]
Actual type: SortedBy name1 [Int]
```

What is the critical difference between these two examples? In the first, a user is allowed to create a named value by fiat. In the second, the user is only allowed to consume a named value, by providing a polymorphic function that can work with any named value. The library’s API provides a helper function—in this case, `name`—for applying the consumer to a normal, unnamed value. In practice, it is as if the library has a secret supply of names, and selects one to use in a manner that is not predictable (or even inspectable!) to the user.

### 3 Case Study #2: Sharing State Threads

The trick for using rank-2 types to conjure names outside of the user’s control was inspired by the ST monad and its rank-2 `runST :: (forall s. ST s a) -> a` function [9]. In this brief case-study, we elaborate the connection between the ST monad and GDP-style names. The new perspective suggests novel extensions to the ST API. In Figure 5 we recall the basic ST API [9], writing `ST` to disambiguate our version from the existing type in Control.Monad.ST.

In their safety analysis of the ST monad, Timany et al. proposed to think of the `s` parameter as representing a name attached to a region of the heap [18]. We can think of `ST s` as acting like some kind of informal State monad over named regions, like in this Haskell-ish pseudocode:

```haskell
data Region = Region

type St s a = State (Region ~ s) a

runSt :: (forall s. St s a) -> a

runSt action = name Region (evalState action)
```

The notion of treating the ST monad’s phantom type as a region name immediately leads to ideas for other primitives. Once we can name regions, why not go on to invent more detailed names to describe the minute contours of those regions? For example, let us see what happens if we add a binary type constructor $\cap$ so that $s \cap s'$ names the region at the intersection of `s` and `s'`. We are quickly led to an API similar to Figure 6 that supports a new capability: individual sub-computations, at their discretion, may decide to share mutable reference cells with other sub-computations.

```haskell
runSt :: (forall s. St s a) -> a

newRef :: a -> St s (a ∈ s)
readRef :: (a ∈ s) -> St s a
writeRef :: (a ∈ s) -> a -> St s ()
```

Figure 5. The standard “state thread” API. We write $a ∈ s$ to denote a reference cell of type `$a$ in the region denoted by `$s$`. In Control.Monad.ST, we would write `a ∈ s` as `STRef s a`.

```haskell
runSt2 :: (forall s s'. St (s ∩ s') a) -> a

liftL :: St s a -> St (s ∩ s') a
liftR :: St s' a -> St (s ∩ s') a

share :: (a ∈ s) -> St s (a ∈ (s ∩ s'))

use :: (a ∈ (s ∩ s')) -> (a ∈ s)
symm :: (a ∈ (s ∩ s')) -> (a ∈ (s' ∩ s))
```

Figure 6. Extending the state thread API with shared references.
Within that computation, the user can run sub-computations
ward to write "missing symbol" handlers into the expression
where one sub-computation creates two cells: one private,
will fail to compile.

vindicate some of those 2000 moments, forever enshrined
value maps that rely on certain keys being present at critical
sions may maintain a symbol table, subject to the invariant
moments. For example, an evaluator for well-scoped expres-
s. Although the
reference is in scope during the second sub-computation,
reference here, the program will not compile.

Figure 7. An ST-style pure computation using local mutable
references. Although the secret reference is in scope during
the calculation in the "right" region, any attempted access
will fail to compile.

In effect, runSt2 lets the user run a computation that
makes use of two partially-overlapping memory regions.
Within that computation, the user can run sub-computations
bound to one or the other memory region. Furthermore, a
sub-computation can move any variable that it owns into the
common overlap via share. An example is shown in Figure 7,
where one sub-computation creates two cells: one private,
and the other shared. A second sub-computation has uncon-
strained access to the shared cell. Yet even though the private
reference is also in scope during the second sub-computation,
any attempts to access it there will fail to compile.

4 Case Study #3: Key-value Lookups

It is not uncommon to find algorithms based around key-
value maps that rely on certain keys being present at critical
moments. For example, an evaluator for well-scoped expres-
sions may maintain a symbol table, subject to the invariant
that any symbol found at an expression node should have a
 corresponding entry in the symbol table. It may be awk-
ward to write "missing symbol" handlers into the expression
evaluator—doubly so if the API was designed so that missing
symbols are supposed to be impossible.

In this case study, we will see how to use the GDP concept
to build an API where the user can express the thought "this
key must be present in that map." In the process, we will
vindicate some of those 2000 moments, forever enshrined
on Hackage, where a programmer fell into the pit of despair
and followed a map lookup by fromJust.

4.1 Designing for the User’s State of Knowledge

It is instructive to compare the two lookup types k -> Map
k v v -> Maybe v and Key ks k -> JMap ks k v v -> v. We
do not intend to claim that one of these is better than the
other. Instead, the claim is much simpler: these two functions
reflect different expectations about the user’s knowledge.

If the user legitimately does not know whether or not a
key is present, then the Maybe-returning lookup is entirely
appropriate. The user’s incomplete knowledge about the
result of the operation is exactly reflected in the return type,
so they will not feel inconvenienced by the need to handle
both the Just v (key present) and Nothing (key absent)
cases.

On the other hand, if the user already believes the key
should be present based on some external evidence, then
they will be happier writing a program that does not need
to handle the impossible missing-key state. But to ensure
safety, they must communicate that evidence to the library
somehow; here, via the Key ks predicate.

4.2 Application: Well-formed Adjacency Lists

The power of this method becomes more apparent when
considering maps where the values are expected to reference
the keys in some way. Consider this simple adjacency rep-
resentation for directed graphs that maps each vertex to its
list of immediate neighbors:

type Digraph v = Map v [v]

Figure 8 gives a small, GDP-style API based on the author’s
justified-containers package2. The key features are:

- A predicate Key ks, meaning "has a key set named ks." A value of type Key ks k is a value of type
k, with a ghost proof that it is present in the key set
named ks.
- A predicate JMap ks, meaning "has a key set named
ks." A value of type JMap ks k v is a Map k v, with
a key set named ks.
- The rank-2 withMap function, analogous to name, that
attaches a ghostly key set to a map. This function
encodes the notion that any map has some set of keys,
perhaps not known to us at compile time.
- The member function. This function checks if a key is
present in a map with key set ks and, if so, produces a
ghost proof of that fact using Key ks.
- Finally, the function lookup. This function is total be-
cause the key carries a ghost proof that it is present
in the map. As a result, lookup can safely return a v
instead of a Maybe v, with no fear of run-time failure.

Note that proving a key can be found in a certain map
does not mean it can only be found in that map! In Figure 9,
we see that some evidence can be re-used: we can find the
same key in a whole variety of maps.
Figure 8. A fragment of the API from justified-containers. The GDP-style predicates `Key ks k` and `JMap ks k v` are used to represent "a value of type k belonging to the set ks" and "a map with key set ks", respectively.

Well-formed Digraphs should satisfy the property that every vertex referenced in any neighbor list is also a valid key in the adjacency map.

Traditionally, graph APIs that use adjacency representations require well-formed graphs, but make it the user's responsibility to ensure well-formedness. For example, the `Data.Graph.Atyp` API from containers has a constructor that will silently discard edges whose targets do not appear in the node list.

Our GDP-style API for maps gives us a vocabulary for translating the notion "a well-formed adjacency list" into a program invariant that can be checked by the compiler. We simply write what we mean: a well-formed adjacency map should map each vertex to a list of vertices that are keys of that same map. In other words:

```haskell
type Digraph vs v = JMap vs v [Key vs v]
```

With the help of this type, a user can now enforce the invariant "this adjacency map must be well-formed" at compile time. A similar strategy can be used to eliminate a whole class of bugs when using symbol tables, evaluation contexts, database models, and or any other data structure based around a recursive key-value store.

### 4.3 Changing the Key Set

But what about maps that are related, yet do not have exactly the same key sets? As a concrete example, consider the insert function. Although insert will usually modify the key set of a map, we still know quite a lot about the keys in the updated map. Imagine you were a user, in possession of a key and a proof that it is present in the original map.

It would be quite frustrating if we were unable to use that same key freely in the expanded map! The library author, anticipating this need, should provide a proof combinator that converts a proof of "`k` is a valid key of `m`" into a proof of "`k` is a valid key of `insert` `k` `v` `m`".

To support this use-case, justified-containers provides the rank-2 function `inserting`:

```haskell
inserting :: Ord k => k -> v -> JMap ks k v -> (forall ks'. JMap ks' k v -> Key ks k -> Key ks' k -> t) -> t
```

Since insertion results in a map with a new key set, we must introduce the ghost of this new key set inside another `forall`. But what are the other parameters being passed to the continuation? They form a collection of evidence and proof combinators that the user may need to formulate a safety argument. Concretely, the continuation has access to:

1. The updated map, of type `JMap ks' k v`.
2. A function that represents the inclusion of `ks` into `ks'`.
3. Evidence that the inserted key is present in the new key set.

The library author must perform a balancing act here. They should give the user an ample supply of evidence and proof combinators to support the user’s arguments, but just what and how much? For example, the user may well want to argue that a every key `other` than the new one is also present in the original map, but the API provides no straightforward way to do this. It is also somewhat awkward to introduce yet another rank-2 function to the API.
-- Type exported, constructor hidden (but see `axiom`)

\[
data \text{ Proof } p \equiv \text{ QED}
\]

-- Attaching proofs to values

\[
\text{newtype } a :::: p = \text{ SuchThat } a
\]

\[
(\ldots) :: a \rightarrow \text{ Proof } p \rightarrow (a :::: p)
\]

x ... proof = coerce x

-- Logical constants. We can use empty data declarations,
because these types are only used as phantoms.

\[
data \text{ TRUE}
\]

\[
data \text{ FALSE}
\]

\[
data p \&\& q
\]

\[
data p \mid\mid q
\]

\[
data p \rightarrow q
\]

\[
data \text{ Not } p
\]

\[
data p =\equiv q
\]

-- Natural deduction rules (implementations all
--- ignore parameters and return `QED`)

\[
\text{andIntro} :: \text{ Proof } p \rightarrow \text{ Proof } q \rightarrow \text{ Proof } (p \&\& q)
\]

\[
\text{andElElm} :: \text{ Proof } (p \&\& q) \rightarrow \text{ Proof } p
\]

\[
\text{orIntro} :: \text{ Proof } p \rightarrow \text{ Proof } (p \mid\mid q)
\]

\[
\text{implIntro} :: \text{ Proof } p \rightarrow \text{ Proof } q \rightarrow \text{ Proof } (p \rightarrow q)
\]

\[
\text{implElElm} :: \text{ Proof } (p \rightarrow q) \rightarrow \text{ Proof } p \rightarrow \text{ Proof } q
\]

\[
\text{notIntro} :: \text{ Proof } p \rightarrow \text{ Proof } \text{ false } \rightarrow \text{ Proof } (\text{ Not } p)
\]

\[
\text{contradicts} :: \text{ Proof } p \rightarrow \text{ Proof } (\text{ Not } p) \rightarrow \text{ Proof } \text{ false }
\]

\[
\text{absurd} :: \text{ Proof } \text{ false } \rightarrow \text{ Proof } p
\]

\[
\text{refl} :: \text{ Proof } (x =\equiv x)
\]

--- and many more

-- Exported function that allows library authors to
--- assert arbitrary axioms about their API.

\[
\text{axiom} :: \text{ Proof } p
\]

\[
\text{axiom} = \text{ QED}
\]

Figure 10. Basic constants and functions for building up the “proofs” in “ghosts of departed proofs”.

In the final case study, we will investigate how the library author can separate API functions from the lemmas about those functions, and in the process remove the need for some of these additional rank-2 functions.

5 Case Study #4: Arbitrary Invariants

In the previous case studies, we saw how introducing names and predicates can help us develop safe APIs that allow the user to express correctness proofs. However, there are a few aspects that remained awkward.

First, we have several ways that a name-like entity could be introduced: either via the name operator itself, or through other library-defined rank-2 functions like runSt2, withMap, or inserting. It would be nice if the same mechanism could be used for all of these cases.

Second, we made extensive use of ghostly proofs carried by phantom type parameters. But these phantom types needed something to attach to, so we introduced various domain-specific newtype wrappers (SortedBy, ε, JMap). Each library exported its own idiosyncratic proof combinators for working with its newtype wrappers. It would be better to have a uniform mechanism for expressing, carrying, and manipulating these proofs.

In this case study, we will consider what kind of APIs we could write if we separated type-level names from the constraints we want to place on the named values. For example, let us return to the head function. We want to ensure that the user only calls head on a list xs with outer constructor (::) (“cons”). To express this condition, we introduce one more newtype wrapper, written :::: and pronounced “such that”. Altogether, the phrase (a :::: n :::: p) should be read “a value of type a, named n, such that condition p holds.”

We can now write, very explicitly, the requirement that our library places on the user of head: the parameter, called xs, must have outermost constructor (:). So we simply introduce a predicate IsCons using an empty datatype, and write down the definition of a GDP-style head:

\[
data \text{ IsNil} xs
\]

\[
data \text{ IsCons} xs
\]

\[
\text{head} :: ([a] :::: xs :::: \text{ IsCons} xs) \rightarrow a
\]

\[
\text{head} \text{ xs} = \text{ Prelude.head} (\text{the} \text{ xs})
\]

The (::::) type is similar to the Refined type from the refinement library [20], but it gains extra power when used together with names: names are the mechanism that allows us to take predicates about specific values and encode them at the type level. The type ([a] :::: xs :::: IsCons xs) becomes a statement about the particular list being passed to head. The library user is now free to come up with a proof of IsCons xs in whatever way they please.

5.1 Logical Combinators for Ghostly Proofs

We now have a mechanism for encoding arbitrary properties as phantom types. But how will the user create ghostly proofs to inhabit those phantom types? We can begin with a very simple Proof type, sporting a single phantom type parameter and exactly one non-bottom value:

\[
data \text{ Proof } p \equiv \text{ QED}
\]

From this humble beginning, we can encode all of the rules of natural deduction as functions that produce terms of type Proof p. Figure 10 gives a small taste of the basic syntax and encoded deduction rules.

Once we have constructed a proof of type Proof p, we can use the (\ldots) combinator to attach that proof to a value of type a, producing a value of type (a :::: p). Note that p will often be a proof about the wrapped value, but that is not required! Any value can carry any proof; the only thing that links value to proof is the use of a common name.
5.2 Naming Library Functions

To help the user create domain-relevant proofs, a library author may wish to export a lemma such as “reversing a list twice gives the original list”. To express this idea, it is not sufficient to have a name “the original list”. We must also be able to name some of the library’s functions. This observation motivates an extension to the Named module (Figure 2), adding these three items:

```haskell
-- module Named, continued:
data Defn = Defn -- Type exported, constructor hidden.

type Defining f = (Coercible Defn f, Coercible f Defn)

-- Allow library authors to define introduction rules for
-- names that they have defined. The coercion is only
-- possible since this function is in the Named module.
defn :: Defining f => a -> (a ~ f)
defn = coerce
```

The idea is that a library author can introduce a new name X by defining X as a newtype alias of Defn. If the library author does not export the constructor of X, then the constraint Defining X only holds in the module where X was defined. It follows that the defn function can be used to attach the name X to an arbitrary value, but only in the module where X was defined. By exporting defn with the Defining f constraint, the Named module allows library authors to introduce new names and axioms, while users remain safely restrained.

How does the library author use this mechanism to introduce new names and axioms in practice? In the case of the list-reversing lemma, the author can write:

```haskell
newtype Rev xs = Rev Defn

reverse :: ([a] ~ xs) -> ([a] ~ Rev xs)
reverse xs = defn (Prelude.reverse (the xs))
```

Note that, in contrast with inserting from the previous case study, the lemmas about a function stand on their own. The library author can add any number of lemmas about reverse without modifying its signature. Furthermore, it also becomes easy to create lemmas that relate multiple functions, such as rev_length and rev_cons in Figure 11. A sample of client code for this library appears in Figure 12, where the user defines a dot product function that only can be applied to same-sized lists. The user then supplies evidence to convince the compiler that the dot product of a list with its reverse is legal.

On the safety of defn It is instructive to momentarily return to the “Simon” example of Section 2.4. Isn’t defn as bad as any_name? There is certainly a danger, but only for the library author who must be very careful indeed about how Simon is introduced. The library users are still unable to name arbitrary values “Simon” merely by using defn, because they do not possess the necessary Defining Simon constraint.

5.3 Building Theory Libraries

In both the St and the justified-containers case studies, the library author exported proof combinators that encoded basic facts about the algebra of sets. Such redundancy is undesirable for the library authors, who have to spend more time writing and testing, but also for the end user who has to remember dozens of variations on the same basic proof combinators.

Luckily, it is simple to factor out axioms and deduction rules for specific theories from the libraries that make use of them. For example, we could publish a small library containing basic predicates and deduction rules about sets, such as:

```haskell
-- Lemmas (all bodies are `axiom`)
rev_length :: Proof (Length (Rev xs) == Length xs)
rev_rev :: Proof (Rev (Rev xs) == xs)
rev_cons :: Proof (IsCons xs) -> Proof (IsCons (Rev xs))

data ListCase a xs = IsCons (Proof (IsCons xs))
                    | IsNil (Proof (IsNil xs))

classify :: ([a] ~ xs) -> ListCase a xs
classify xs = case the xs of
              (~~) -> IsCons axiom
              []  -> IsNil axiom
```

Figure 11. A GDP-style module for manipulating and reasoning about lists. A variety of lemmas are exported by the module, to provide the user with a rich set of building blocks for constructing safety proofs.
Ghosts of Departed Proofs

Well-typed by expressing a proof that

vec

A small, core set of axioms are placed in the

to increase confidence, the author of the theory library can

This same theory library could be used to reason about

validity of their own theory module; others can then re-use

5.4 Ghosts on the Outside, Proofs on the Inside

Factoring common lemmas into theory modules helps ensure

and carefully audited for correctness. A larger set of lemmas,
derived from these axioms using GDP proof combinators,
reside in the Derived submodule. This separation allows
the library author to build up a large collection of lemmas from
a small—and hopefully easy-to-verify—set of basic axioms.

5.5 Building Custom Proof Tactics

For simple properties, the task of writing a proof is not too dif-
ficult. But for more sophisticated properties, the deployment of proof tactics becomes crucial. A proof tactic is a search
strategy for proofs, usually targeted at proving one partic-
lar class of theorems. For example, the Coq tactic omega
is useful for proving theorems about arithmetic, while

is useful for simplifying a complex goal.

Tactics are often designed with a specific domain in mind;
to be most useful, theory creators (and library authors) should
be able to create their own tactics when needed.

One approach to providing custom tactics is to leverage
GHC’s support for type-checker plugins. These plugins hook
into GHC’s OutsideIn(X) inference algorithm [21], teaching
it to solve new kinds of type constraints.

As a proof-of-concept, we developed a simple typechecker
plugin that implements proof by analytic tableaux [15] for
propositional logic. This tactic can verify the satisfiability
of any valid formula of propositional logic; the naïve imple-
mentation takes about 60 lines of Haskell, plus 150 lines of
glue code to mediate between the tableaux solver and GHC.

To trigger the custom tactic, we introduce an empty in-
dex type family — hidden from the user—and a single
exported function tableaux:

```

type family ProofByTableaux p = p' | p' -> p

tableaux :: ProofByTableaux p

tableaux = error "proof by analytic tableaux."
```

Morally, we want to think of ProofByTableaux p as an alias
for p. The trick is that our plugin will first get a chance to
check that the proposition p is a valid formula of proposi-
tional logic. Only then will the plugin allow GHC to replace
ProofByTableaux p with p.

For the user, the effect appears to be that tableaux can act
as a value of type Proof p whenever p is a valid formula in
propositional logic. A glance at Figure 13 demonstrates why
proof tactics are so desirable: the user can just wave their
hands and say “this is true by basic facts from propositional
logic,” instead of constructing a tedious proof by hand.

```

Figure 12. A user-defined dot product function that can
only be used on same-sized lists, and a usage example. In the
implementation of dot_rev, the user makes the use of dot
well-typed by expressing a proof that vec and reverse vec
have the same length. Note that refl and rev_length are
effectively axiom schemas, and unification with the type of
dot selects the correct instances of these schemas.

```

```

Figure 13. Proving the same theorem in two different
ways. The first proof uses the proof combinators from Figure
10. The second proof uses a typechecker plugin, exposed
through the tableaux function (Section 5.5).

```

```

proof1, proof2 :: Proof ((p --> q) --> (Not q --> Not p))

proof1 =

implIntro $ \p2q ->
implIntro $ \notq ->
notIntro $ \p ->
(implElim p2q p) `contradicts` notq

proof2 = tableaux

```

```

```

```

We only have "confidence"—not "surety"—because, after all, Haskell’s type
system is inconsistent as a logic. That does not render it useless for the
evidence-pushing we need for GDP, however!
5.6 Using Reflection to Pass Implicit Proofs

The reflection library is an implementation of Kiselyov and Shan’s functional pearl about implicit configurations [7], allowing the user to pass values implicitly using Haskell’s typeclass machinery. We can combine reflection with GDP in order to pass proofs implicitly, so they seem to appear out of thin air just when they are needed. The relevant part of the reflection API consists of two functions: gives the user make a proof implicit, and given recalls an implicit proof from the current context:

```haskell
type Fact p = Given (Proof p) -- a useful constraint synonym

give :: Proof a -> (Fact a => t) -> t

given :: Fact a => Proof a
```

To make this approach practical, we also need some way to apply implications to facts in the current implicit context. Since the implications will generally be name-polymorphic, it can be slightly tricky to apply an implication to a specific fact. When the antecedent of the implication is a simple predicate, we can make use of a combinator such as this:

```haskell
using :: Fact (p n) => (Proof p -> Proof q) -> (Fact q => t) -> t

using impl x = give (impl given)
```

The named parameter x :: a ~~ n is used to help select the right proof from the context. We now are able to reflect proofs manually, but can we make the process more automatic? For example, could we automatically introduce Fact (IsCons xs) to the implicit context inside the cons branch of a pattern-match? Yes, indeed; pattern-matching on a GADT constructor can bring new constraints into scope:

```haskell
data ListCase' a xs where
    Cons :: Fact (IsCons xs) => ListCase' a xs
    Nil :: Fact (IsNil xs) => ListCase' a xs

classify' :: forall a xs. ([a] ~~ xs) -> ListCase' a xs

classify' xs = case the xs of
    ([a] ~> give (axiom :: Proof (IsCons xs)) Cons
    [] ~> give (axiom :: Proof (IsNil xs)) Nil
```

Figure 14 combines these simple ingredients to create a reflection-based version of the original GDP example from Figure 1. Comparing the two examples side-by-side, the use of reflection with GDP seems to offer a substantial improvement to user ergonomics. On the other hand, there is a small amount of run-time overhead due to the passing of typeclass dictionaries for Facts, and it is not always easy to extract the right proof from the implicit context without adding type annotations.

6 Related Work

Phantom type parameters have several well-known applications in API design, supporting typed embedded domain-specific languages (EDSLs) [12], pointer subtyping [11], and access control policies [5]. Most of these applications rely on monomorphic or universally-quantified phantom type parameters; by contrast, GDP relies on existentially-quantified phantom names, and a rich, extensible set of combinators for building arguments about the named values.

Previous designs that use existentially-quantified phantom types include lazy state threads [9] and lightweight static capabilities [8]. The GDP approach explicitly separates two orthogonal concerns within these designs: the introduction of existentially-quantified type-level names, and the manipulation of proofs about those named values.

There are a variety of other approaches to checking the correctness of Haskell code. Liquid Haskell works with an SMT solver to verify certain classes of properties about Haskell functions [19]. hs-to-coq converts Haskell code to Coq code, allowing theorems to be proved within the Coq proof assistant [16]. In both cases, the properties and proofs exist outside of (or in parallel to) the existing Haskell code. In the GDP approach, properties and proofs are carried by normal Haskell types and are checked by compilation.

7 Summary

Ghosts of Departed Proofs provides a novel approach to safe API design that enables a dialogue between the library and the user. By giving the user a vocabulary for expressing safety arguments, GDP-style APIs avoid the need for partial functions or optional returns. Using this approach, we are able to achieve many of the benefits of dependent types and refinement types, while only requiring mild and well-known extensions to Haskell 2010. You can try it with your own libraries, today!

Acknowledgments

The author would like to thank Baldur Blöndal, Hillel Wayne, Philipp Kant, Matt Parsons, and the anonymous reviewers for their helpful feedback on a draft of this paper, and Input Output HK for supporting this work.

References


