Functional Pearl: Witness Me — Constructive Arguments Must Be Guided with Concrete Witness

Hiromi Ishii
h-ishii@math.tsukuba.ac.jp
DeepFlow, Inc.
Fujimi-shi Tsuruse nishi, Saitama prefecture, Japan

Abstract
Beloved Curry–Howard correspondence tells that types are intuitionistic propositions, and in constructive math, a proof of proposition can be seen as some kind of a construction, or witness, conveying the information of the proposition. We demonstrate how useful this point of view is as the guiding principle for developing dependently-typed programs.

CCS Concepts: • Theory of computation → Constructive mathematics; Type theory; Logic and verification.

Keywords: Haskell, dependent types, promotion, demotion, singletons, polymorphism, kinds, invariants, type-level programming

1 Introduction
Since Haskell had been given a Promotion [17], using Haskell with dependent types is a joy. It’s not only a joy: it gives us a neat language to express invariants of programs with type-level constraints. But it also comes with pain: writing correct but maintainable dependently-typed programs in Haskell sometimes is a hard job, as explained by Lindley and McBride [11]. It is particularly hard when one tries to bridge a gap between types and expressions, to maintain complex type constraints, and so on.

Here, we propose to borrow the wisdom from one of the greatest discovery in Computer Science and Logic: Curry–Howard correspondence. It tells us that types are propositions and programs are proofs in intuitionistic logic, in a rigorous sense. According to Brouwer–Heyting–Kolmogorov (BHK) interpretation\(^1\), its informal ancestor, a proof of an intuitionistic proposition is some kind of a construction, or witness, conveying the information of the proposition. For example, a proof of \(\varphi \land \psi\) is given by a pair of proofs \(T : \varphi\) and \(S : \psi\); that of \(\varphi \rightarrow \psi\) is given by a function taking a proof of \(P\) and returns that of \(Q\), and so on. The interpretation of our particular interest in this paper is that of disjunction:

\[
A : \varphi \lor \psi \iff A = (i, B),
\]

where \(i = 0\) and \(B : \varphi\), or \(i = 1\) and \(B : \psi\),

That is, a witness of disjunction is given by a tuple of the tag recording which case holds and its corresponding witness.

\(^1\)For the precise historical background, we refer readers to the article by Wadler [15].

Although BHK interpretation is not as rigorous as Curry–Howard correspondence, its looseness allows us much broader insights, as we shall see in the present paper.

This paper is organised as follows.

• In Section 2, taking a type-level GCD as an example, we demonstrate how to demote closed type-level functions involving pattern-matchings. We suggest adding witnessing arguments to such type-level functions to make the compiler aware of evaluation paths.
• Section 3 demonstrates that we can emulate disjunctions of type constraints, provided that constraints in question can be recovered from some statically computable witnesses. We use a field accessor for a union of extensible records as an example.
• Section 4 shows a practical example of a dependently-typed plugin system type-checkable dynamically at runtime. There, we see how the combination of the Deferrable constraint pattern [9] and witness manipulation can be used to achieve this goal. We also discuss the design of the witness of type-level equalities.
• Finally, we conclude in Section 5.

A complete working implementation is available in the demotion-examples directory of the support repository [5].

1.1 Preliminaries
In this paper, we use the standard method of singletons [2] to simulate dependent types in Haskell. Briefly, a singleton type \(\text{Sing } a\) of a type-level value \(a\) is the unique type that has the same structure as \(a\), on which we can pattern-match to retrieve its exact shape. \(\text{Sing } a\) can be identified with a type \(a\) but demoted to the expression-level. In particular, we assume the following API:

\[
\begin{align*}
\text{type family } &\text{Sing } :: \ k \to \text{Type} \\
\text{class } &\text{Known } a \text{ where } -- \text{SingI in singletons} \\
&\text{sing } :: \text{Sing } a \\
&\text{withKnown } :: \text{Sing } a \to (\text{Known } a \Rightarrow r) \Rightarrow r \\
\text{data } &\text{SomeSing } k \text{ where } -- \text{SingKind in singletons} \\
&\text{MkSomeSing } :: \text{Sing } (a :: k) \Rightarrow \text{SomeSing } k \\
\text{class } &\text{HasSing } k \text{ where} \\
&\text{type } \text{Demoted } k
\end{align*}
\]
demote :: Sing (a :: k) -> Demoted k
promote :: Demoted k -> SomeSing k

withPromoted :: HasSing k
=> Demoted k
-> (forall x. Sing (x :: k) -> r) -> r

type FromJust :: ErrorMessage -> Maybe a -> a
type family FromJust err may where
FromJust err 'Nothing = TypeError err
FromJust _ ('Just a) = a

type instance Sing = (SNat :: Nat -> Type)
sNat :: KnownNat n => SNat n

withKnownNat :: SNat n -> (KnownNat n => r) -> r
(a+) :: SNat n => SNat (n + m) -> SNat (n `Mod` m)

sMod :: SNat n -> SNat m -> SNat (GCD n m)

For the detail of singleton-based programming, we refer readers to Eisenberg–Weirich [2] and Lindley–McBride [11]. We use the following convention:

1. We prefix singleton types with the capital \( \texttt{SNat} \); e.g. \( \texttt{SNat n} \) is the type of a singleton of \( n \).
2. For a type-level function we use small \( \texttt{s} \) as a prefix for singletonised expression-level function: \( \texttt{sGCD} \) is the singletonised version of \( \texttt{GCD} \).
3. For operators, we prefix with \( \texttt{x} \); \( \texttt{x+} \) is the singleton for the type-level \( \texttt{+} \).

2 Toy Example: Demoting Type-level GCD

Let us begin with a simple example of type-level greatest common divisors (GCDs):

```haskell
import GHC.TypeLits

type family GCD n m where
GCD 0 m = m
GCD n 0 = n
GCD n m = GCD (Mod m n) n

So far, so good.

>>> :kind! GCD 12 9
GCD 12 9 :: Nat
= 3

Suppose we want to “demote” this definition of \( \texttt{GCD} \) to expression-level using singletons, that is, to implement the following function \( \texttt{sGCD} \):

```haskell
sGCD :: SNat n -> SNat m -> SNat (GCD n m)
```

First, we need to test the equality of type-level naturals. In the base package, there is a suitable type-class for it:

```haskell
-- Defined in Data.Type.Equality in base
class TestEquality f where
testEquality :: f a -> f b -> Maybe (a :-: b)
data (:-:) a b where Refl :: a :-: a
```

Assuming the \( \texttt{TestEquality SNat} \) instance, one might first attempt to write it as follows:

```haskell
sGCD :: SNat n -> SNat m -> SNat (GCD n m)
sGCD sn sm =
  case (testEquality sn (sNat 0)),
  testEquality sm (sNat 0)) of
    (Just Refl, _) -> sn
    (_, Just Refl) -> sn
    (Nothing, Nothing) -> sGCD (sMod sm sn) sn
```

The first two cases type-check as expected, but the last case results in the following type error:

```
Couldn't match type \( \texttt{GCD (Mod m n) n} \)
with \( \texttt{GCD (Mod m n) n} \)
```

Why? The definition of \( \texttt{sGCD} \) seems almost literally the same as type-level \( \texttt{GCD} \). It first match \( \texttt{m} \) against \( \texttt{n} \), then \( \texttt{m} \) against \( \texttt{n} \), and finally fallbacks to \( \texttt{GCD (Mod m n) n} \).

Carefully analysing the first two cases, one can realise that there are additional type-level constraints introduced by \( \texttt{GADT constructor:} \)

```haskell
Refl :: a ~ b => a ~: b
```

Thus, in the first two cases, the compiler can tell either \( n = m \) or \( m = n \). Since the \( \texttt{GCD} \) is defined as a closed type family, the compiler can match clauses in a top-down manner and successfully apply either of the first two clauses of the definition of \( \texttt{GCD} \). In other words, the constructor \( \texttt{Refl} \) witnesses the evaluation path of type-level function \( \texttt{GCD} \) in the first two cases.

In the last case, however, no additional type-level constraint is available. Despite humans can still think “as all the \( \texttt{Refl} \) clauses failed to match, hence the non-equal clause must apply here”, this intuition is not fully expressed in the type-level constraint!

So we have to give the compiler some witness also in the last case. What kind of a witness is needed here? Well, we need to teach the compiler which clause was actually used.
In this case, branching is caused by type-level equality: the evaluation path depends on whether \((n \equiv m)\) or \((n \equiv \_\_\_)\) is \(\text{True}\) or not. First, let us make this intuition clear in the definition of \(\text{GCD}\):

```haskell
import Data.Type.Equality (type (==)) -- from base

-- \(\text{type}\) \(\text{GCD}\) \(n m = \text{GCD}\_\_ (n == 0) (m == 0) n m\)
-- \(\text{type}\) \(\text{family}\) \(\text{GCD}\_\_\ nEq0 mEq0 n m :: \text{Nat}\) where
-- \(\text{GCD}\_\_ \text{\_True} \_ m = m -- n \equiv 0; \text{return} m\)
-- \(\text{GCD}\_\_ \text{\_False} \text{\_True} n \_ = n -- m \equiv 0; \text{return} n\)
-- \(\text{GCD}\_\_ \text{\_False} \text{\_False} n m = -- \text{Neither; recur!}\)
-- \(\text{GCD}\_\_ (\text{Mod} m n == 0) \text{\_False} (\text{Mod} m n) n\)
```

Here, we have two type-level functions: newly defined one, \(\text{GCD}\_\_,\) is the main loop implementing Euclidean algorithm, and \(\text{GCD}\) is redefined to call \(\text{GCD}\_\_,\) with the needed information. Now, \(\text{GCD}\_\_,\) takes not only natural numbers but also a type-level \(\text{Bool}\) \(\text{witnessing}\) equality of \((n \equiv m)\) and \((n \equiv \_\_\_)\) with \(\_\_\_.\) From this, GHC can tell which clause is taken from the first two type-arguments. As clauses in closed type families can be viewed as a mutually exclusive alternatives, this approach shares the spirit with the constructive BHKs interpretation of \(\land\).

Now that we can give the compiler witnesses as the first two type-arguments of \(\text{GCD}\_\_,\) we are set to implement \(\text{SBool}\). First, we need \text{demoted} version of type-level \((==)\). The first attempt might go as follows:

```haskell
(==) :: TestEquality f
    => f a -> f b -> SBool (a == b)
sa %== sb = case testEquality sa sb of
    Just Refl -> STrue
    Nothing -> SFalse
```

Unfortunately, this doesn’t work as expected. The first error on \(\text{STrue}\) says:

```
Could not deduce: \((a == a) \equiv \text{\_True}\)
from the context: \((b = a)\)
  bound by a pattern with constructor:
    \(\text{Refl} \equiv \text{forall} k (a :: k). a :: a,\)
  in a case alternative
at \(\ldots/\text{GCD}\_\_.hs:33:8-11\)
Expected type: \(\text{SBool}\ (a == b)\)
Actual type: \(\text{SBool} \text{\_True}\)
```

This is due to the definition of type-level \((==)\) in GHC base library:

```haskell
type family a == b where
    f a == g b = (f == g) && (a == b)
```

As described in the documentation [4], the intuition behind the definition of the first clause is to let the compiler to infer, e.g. \(\text{Just} a == \text{Just} b\) from \((a == b)\).

This behaviour is desirable when one treats equalities involving compound types, like \((f a == g b) \equiv \text{\_True}\). But when one wants to give a witness of \((a == b) \equiv \text{\_True}\), we cannot make use of \(\text{Refl}\). This is, again, due to the lack of witness of being distinct.

A solution here is just to define another type-family, which requires the reflexivity only:

```haskell
(==) :: TestEquality f
    => f a -> f b -> SBool (a == b)
sa %== sb = case testEquality sa sb of
    Just Refl -> STrue
    Nothing -> SFalse
```

Although this equality cannot treat equalities between compound types inductively, it suffices for \(\text{GCD}\) case. We will revisit to a treatment of type-level equality in Section 4. Demoted version of this now gets:

```haskell
(==) :: TestEquality f
    => f a -> f b -> SBool (a == b)
sa %== sb = case testEquality sa sb of
    Just Refl -> STrue
    Nothing -> SFalse
```

Now, the type-error remains on the last clause: \(\text{SFalse}\). This is, again, due to the lack of witness of being distinct. But wait! We are just struggling to produce such a negative witness of non-equality, which in turn requires itself. A vicious cycle! At this very point, there is no other way than resorting to the ancient cursed spell \(\text{unsafeCoerce}\):

```haskell
import Unsafe.Coerce
	
sa %== sb = case testEquality sa sb of
    Just Refl -> STrue
    Nothing -> unsafeCoerce SFalse
```

This use of \(\text{unsafeCoerce}\) is inherently inevitable. Fortunately, provided that \(\text{TestEquality}\) instance is implemented soundly, this use of \(\text{unsafeCoerce}\) is not cursed: this is just postulating an axiom that is true but there is no way to tell it to the compiler safely. If one wants to construct evidence of type-level (non-)equality solely from the expression, we must assume some axiom and introduce it by \(\text{unsafeCoerce}\). This
is how library builders usually do when they implement basic (expression-level) operators to manipulate type-level values. Such “trust me” axioms can be found, for example, in instances in singletons [1] package. Anyway, we are finally at the point of implementing working , replacing every occurrence of with our custom :

```hs

```

```hs

```

Finally, the compiler gets happy with all the definitions! We can confirm that the above works just as expected:

```hs

```
The idea is that if the concrete value of \texttt{FindIndex' k ks} is at the compile-time, we can use it to retrieve a field in a record. Note that if the value of \texttt{FindIndex' key keys} was 'Nothing \texttt{FindIndex' key keys} reduces to a type-error. In such a case, since the type-level language of Haskell is strict, the entire constraint \texttt{Known (FindIndex' key keys)} throws a type-error during instance resolution:

\[
\begin{array}{l}
\text{getRecField \texttt{@Bool (EmptyRecord \texttt{@Maybe)}}}
\end{array}
\]

\[
\begin{array}{l}
\text{Key 'Bool' is absent in the list: '[]}\]
\end{array}
\]

\[
\begin{array}{l}
\text{getRecField \texttt{@Bool (['}a\texttt{']): [< [True, False]}}
\end{array}
\]

\[
\begin{array}{l}
\text{:< ([]}:: [\texttt{[]}]): <\texttt{EmptyRecord)}}\]
\end{array}
\]

\[
\begin{array}{l}
\text{[True, False]}
\end{array}
\]

A type \texttt{Index k ks} is a witness of the membership of a label \texttt{k} in a type-level list \texttt{ks}. The function \texttt{walkIndex} walks an extensible record along a given \texttt{Index k ks} and retrieves a value of \texttt{h ks}.4

We can also compute \texttt{Index} at type-level in an obvious way\textsuperscript{5}, where \texttt{<<(-->)} is a type-level analogue of \texttt{fmap}.

A type \texttt{Index k ks} is a witness of the membership of a label \texttt{k} in a type-level list \texttt{ks}. The function \texttt{walkIndex} walks an extensible record along a given \texttt{Index k ks} and retrieves a value of \texttt{h ks}.

We can also compute \texttt{Index} at type-level in an obvious way\textsuperscript{5}, where \texttt{<<(-->)} is a type-level analogue of \texttt{fmap}.

4The same remark on efficiency as above also applies here: it is much more practical to use an index number represented as a newtyped \texttt{Int} if one needs \texttt{O(1)} random access on fields. In such a case, one has to use \texttt{unsafeCoerce} carefully to convince the compiler.

5Oh, you noticed something? Well, in Section 4, we will turn to the implementation of \texttt{FindIndex} again.
instance Known (UnionedIndex' a ls rs) => HasFactor (h a) (RecUnion h ls rs) where
gETFactor (UnionRec l r) =
case sing @(UnionedIndex' a ls rs) of
  SLeft pth -> withKnown pth $ getFactor l
  SRight pth -> withKnown pth $ getFactor r

Everything seems fine, but then GHC complains on call-sites of getFactor:

\[
\text{Could not deduce} \quad (\text{Known (FromJust (... (FindIndex a ls))} ...)
\]

The error seems weird at first glance: we are just giving the dictionary with withKnown pth, where \( \text{pth} \) is of type either Known Index a ls or withKnown Index a rs. Why?
The root cause of this error is that the type-checker doesn’t know the following facts:

1. \( \text{FromJust 'Text m} \) commutes with \( \text{<=>} \) and
2. If \( \text{FromJust x m} \) reduces, then \( \text{m - 'Just (FromJust z m)} \)

for any \( z \).

Although these two facts seem rather obvious, it needs some non-trivial axioms to infer. Hence, if we want to convince the compiler without modifying the instance definitions, we have to augment the compiler with type-checker plugins [3]. Although writing type-checker plugins is a fun, it is not so easy to implement it correctly. Is there any other way to avoid this obstacle?

At this point, we must notice one fact: we can still use \textit{any} value of type Index k ks to retrieve a value of type \textit{h k} from \textit{Record h ks}. Indeed, as \textit{k} can have duplicated elements, there can be more than one \textit{Index k ks} at the same time, e.g. for \textit{Index [3,4,3,3]}. FindIndex was just a canonical way of computing the left-most such index, if present. To summarise, requiring \textit{Known (LookupIndex' k ks)} in the \textit{HasFactor} instance for extensible records was just too much. What we really need is a constraint demanding “there is at least one value of type \textit{Index k ks} given”, embodied by the following class and helper functions:

```
class Given a where
given :: a
give :: a -> (Given a => r) -> r
```

These are excerpted from widely-used reflection package [10] which implements Implicit Configuration [8].

With this, we can rewrite the \textit{HasFactor} instance in more robust way:

```
type Member k ks = Given (Index k ks)

gETFactor (UnionRec l r) =
case Member m of
  Member k ks => Record h keys -> h key

getRecField
  :: Member key keys => Record h keys -> h key

getRecField = walkIndex given

-- | Serves as a default instance for @Member@. Can be safely overridden by 'give' operator.
instance Known (FindIndex' k ks) => Given (Index k ks) where
  given = demote $ sing @(FindIndex' k ks)

instance (Given (Index a keys), h ~ h') => HasFactor (h' a) (Record h keys) where
gETFactor = getRecField

With this, we can now successfully implement\textit{HasFactor} instance for \textit{RecUnion} as follows:

```
-- To avoid orphan @Given@ instance
newtype IndexUnion k ls rs = WrapIdxUnion
                                  (unUnionIdx :: Either (Index k ls) (Index k rs))

instance Known (UnionedIndex' k ls rs)
  => Given (IndexUnion k ls rs) where
  given = WrapIdxUnion $ demote $ sing @(UnionedIndex' k ls rs)

instance Given (IndexUnion k ls rs)
  => HasFactor (h' a) (Record h keys) where
  getFactor (UnionRec l r) =
case unUnionIdx
  $ given @(IndexUnion k ls rs) of
    Left pth -> give pth $ getFactor l
    Right pth -> give pth $ getFactor r
```

In above two implementations, \textit{Given} instances serve as a “default instances” to calculate witnesses. As already shown in the implementation of \textit{HasFactor}, it can be overridden by \textit{give} operator without resulting in type errors complaining about overlapping instances or “Could not deduce (Known ...)”. This flexibility is the reason we chose to use \textit{Given} class instead of the ImplicitParams GHC extension providing similar functionality of dynamic scoping; the instance shadowing in ImplicitParams can have unpredictable behaviour. Such “defaulting” cannot be done for \textit{Known}, because type family applications such as \textit{Index k ks} cannot appear at RHS of instance declarations.

Let us check it works as expected:
Functional Pearl: Witness Me

Suppose we have a program that reads a given input store and returns some outputs generated by prespecified plugins. An input store is represented as an extensible record, and plugins are also specified in type-level. The following signature for the main logic illustrates the idea:

```haskell
type SPlugins ps = SList (ps :: [Plugin])
processStore :: All (RunsOn keys) ps => Store keys -> SPlugins ps -> Outputs ps
class IsPlugin (p :: Plugin) where
data PStoreType p
data POutput p
type Runnable p (ks :: [StoreKey]) :: Constraint
process :: Runnable p keys => proxy p -> Store keys -> POutput p
class Runnable p keys => RunsOn keys p
instance Runnable p keys => RunsOn keys p
```

Thus, each plugin \( p \) is given with a type-level constraint \( \text{Runnable p keys} \) (and its flipped version \( \text{RunsOn p keys} \)) to determine if it is runnable with input stores with the given keys. A function \( \text{processStore} \) takes an input store and (a singleton of) a list of plugin runnable on the given store and then returns the final outputs. This works well if one specifies type variables \( \text{keys} \) and \( \text{ps} \) statically. Suppose that, however, now the situation is changed and we want to determine \( \text{ps} \) dynamically, depending on external configurations. This poses two challenges:

1. We have to resolve constraint \( \text{All (RunsOn keys p)} \) at runtime, and
2. We have to demote type-level operators as singletonised functions.

In this section, we will see how we can apply the witness-pattern to achieve these goals safely.

4 Case Study: Dependently-typed Plugin System Type-Checked Dynamically

As an application of the methods developed in Sections 2 and 3, now we look into a more involved and practical example: demoting existing type-level constraints and resolve them dynamically.

---

A field of type ``999` not found in either:
Left: `[5, 42]`
Right: `'[94, 5]`

3.2 Summary

We saw that we could emulate a disjunction of type constraints if the constraints in question have witnessing type, and there is a canonical way of statically computing such witnesses at type-level. In the above example, \( \text{HasFactor} \) and \( \text{Member} k \text{ keys} \) are such constraints, witnessed by \( \text{Index k keys} \) and \( \text{Member k keys} \), respectively. In expressing the existence of witnesses, it allows a precise and robust handling to use the \( \text{IsPlugin} \) class. Especially, this makes it easy to give “fallback” witness when canonical witness constructor can fail. Although we treated only extensible records here, we can apply the same technique to more general settings; we refer curious readers to the accompanying repository [5].
Constraint (Listing 2) is simple: it reads the $GreetOutput::\text{VarJust}\cdot\text{Var}\cdot\text{GreetEnv}\cdot\text{Greetable_}$ keys where $\text{GreetOutput}::\text{POutput}\cdot\text{PStoreType}\cdot\text{Greetable_}$. The $\text{GreetOutput}\cdot\text{POutput}\cdot\text{PStoreType}\cdot\text{Greetable_}$ keys is given, greet once to that name; otherwise, return the default greeting message.

Listing 2. An implementation of Doubler.

Let’s see an actual implementation for plugins $\text{Doubler}$ and $\text{Greeter}$ (Listing 2) is simple: it reads the value of $\text{IntStore}$, returns the value multiplied by 2. As it requires the value associated with $\text{IntStore}$, it falls to type-check.

Another example, $\text{Greeter}$, in Listing 3, is much more complicated. Its logic is as follows:

1. If there is a field with label $\text{PStore Greeter}$, generate greeting message based on it;
2. if $\text{Name}$ is given, greet once to that name;
3. otherwise, return the default greeting message.

In short, $\text{Greeter}$ involves a fallback strategy on fields in the input store. This strategy is formulated as a $\text{Known}$ constraint; first, try to inspect the existence of the label and then fallback to the condition on the existence of $\text{PStore Greeter}$. This fallback strategy can be seen as an amalgamation of methods presented in Sections 2 and 3, giving another way to emulate disjunction of constraints\(^7\).

Let’s see some examples:

```haskell
>>> processStore (MkStoreEntry @Name "Superman")
```

\(^7\)In this case, however, it would suffice to require both $\text{Known}$ constraints in conjunction. However, in more complex cases, it saves compilation time to use such fallback strategies.
4.2 Make it dynamic

OK, let’s implement a dynamic variant of \( \text{processStore} \). In particular, we will make the resolution of constraints dynamic, implementing the following function:

\[
\text{processStoreDynamic :: Known keys} \\
\rightarrow \text{Store keys} \rightarrow [\text{Plugin}] \\
\rightarrow \text{Either String} \\
\text{SomeRec (RunsOn keys) POutput}
\]

\[
\text{data SomeRec c f where} \\
\text{MkSomeRec :: All c keys} \rightarrow \text{Sing keys} \\
\text{Record f keys} \rightarrow \text{SomeRec c f}
\]

One nontrivial challenge here is to resolve constraints of form \( \text{Known keys} \) at runtime, not compile-time. First, there is a well-known design pattern to allow such instance resolutions at runtime: the \text{deferrable constraint pattern}:

\[
\text{class Deferrable p where} \\
\text{deferEither :: proxy p} \rightarrow (p \Rightarrow r) \\
\rightarrow \text{Either String r}
\]

This is provided in \text{module from constraints package [9]}, and it was first proposed by Dimitrios Vytiniotis. For example, assume the following single-tonised version of \text{FindIndex}:

\[
\text{sFindIndex :: Sing key} \rightarrow \text{SList keys} \\
\rightarrow \text{SMaybe (FindIndex key keys)}
\]

Then we can implement \( \text{Deferrable} \) for \( \text{Member k ks} \):

```
instance (Known k, Known ks) => Deferrable (Member k ks) where
deferEither _ =
  case sFindIndex (sing @k) (sing @ks) of
  SJust _ -> give (remove n) $ Right act 
  SNothing -> Left "Not found"
```

This illustrates another advantage of using \( \text{Known} \) instead of \( \text{Given} \) in terms of \( \text{class constraint} \). Unfortunately, for the same reason, we cannot write direct \( \text{runnable} \) instances for \( \text{IsPlugin} \) in general, as it is a \text{type family} and instance declarations cannot include type family application in their header.

Instead, we provide the dedicated class \( \text{DynamicPlugin} \) of plugins which allow the deferral of corresponding \( \text{Runnable} \):

```
class IsPlugin p => DynamicPlugin p where
deferDynamicPlugin
  :: Known keys \\
  => pxy p => Proxy keys \\
  => (Runnable p keys => r) => Either String r
```

So it remains to implement \( \text{IsPlugin} \) instances for each plugin. Fortunately, all the \( \text{Runnable} \) definitions defined so far can be resolved with singleton manipulation.

Let’s see how to resolve \( \text{Runnable} \) dynamically. Recall its definition:

```
type Runnable 'Doubler ks = Member 'IntStore ks
```

By definition, and the default implementation is resolved with the super-class constraint \( \text{Known (FindIndex 'IntStore keys)} \). we can implement \( \text{Runnable} \) as follows:

```
instance DynamicPlugin 'Doubler where
  deferDynamicPlugin _ (_ :: Proxy keys) =
  deferEither @ (Member SIntStore keys)
```

We can likewise implement the instance for \( \text{Greeter} \):

```
instance DynamicPlugin 'Greeter where
  deferDynamicPlugin _ (_ :: Proxy keys) act =
  case sFindIndex (SPStore SGreeter) keys of
  SJust _ -> withKnown (remove n) $ Right act 
  SNothing -> withKnown (sFindIndex SName keys) $ Right act 
  where keys = sing @keys
```
Hence, it remains to implement `sFindIndex`. Recall our current implementation of `FindIndex`:

```haskell
type family FindIndex k ks where
  FindIndex _ '[] = 'Nothing
  FindIndex k (k' :<#> ks) = 'Just 'Here
  FindIndex k (_, :<#> ks) =
    'There <$> FindIndex k ks

sFindIndex : TestEquality (Sing @a) => Sing (k :: a) -> SList keys
sFindIndex (k :: s) = case k %=== k' of
  STrue -> SJust 'Here
  SFalse -> SThere <$> sFindIndex k ks
```

This tells us the reason why we got stuck: in `STrue`-branch, GHC knows that `(k == x) ~ True`, but GHC cannot infer `k ~ x` from it!

To avoid such information loss, it is convenient to pack all extensionally equivalent constraints. Recall we have three type-level (homogeneous) equalities:

1. `a ~ b`, the built-in equality constraint,
2. `a ~ b`, a type-level boolean predicate that plays well with compound types but lacks automatic reflexivity, and
3. `a = b`, a type-level boolean predicate which takes only the reflexivity into account.

These three equalities can play their roles case-by-case, although their extensions must coincide. In addition, GHC cannot tell that the latter two equalities are symmetric. So, it is useful to pack all these into a single witness, as follows:

```haskell
data Equality a b where
  Equal :: ((a ~ b) ~ True, (b ~ a) ~ True, (a ~ a) ~ True, a ~ b) => Equality a b
  NonEqual :: ((a ~ b) ~ False, (b ~ a) ~ False, (a ~ a) ~ False) => Equality a b

ok, now we can implement `sFindIndex`:

```haskell
sFindIndex :: TestEquality (Sing @a) => Sing (k :: a) -> SList keys
sFindIndex (k :: s) = case k %=== k' of
  STrue -> SJust 'Here
  SFalse -> SThere <$> sFindIndex k ks
```

As expressed by the constructor names, here `Equal` witnesses the equality of given two types, and `NonEqual` witnesses non-equality. In this way, `Equality` packages both positive (equal) and negative (non-equal) witnesses.

One might wonder why we didn’t mention equality constraint `(a ~ b)` in `NonEqual` case. This is because GHC can infer that `(a ~ b)` is not inhabited from `(a ~ b) ~ False`!  

```haskell
{-# LANGUAGE EmptyCase, LambdaCase #-}
import Data.Void
fromFalseEq :: (a ~ b) ~ False => a ~: b -> Void
fromFalseEq = \case {}
```

To allow equality test with witnesses, we use the following class:

```haskell
class SEqual k where
  (%) :: Sing (a :: k) -> Sing b -> Equality a b
```

Some reader might realise that this looks similar to `sEqual` class in singletons package [1]. The difference lies in the non-equal case: `sEqual` returns `a ~: b -> Void` when non-equal. This design decision works well if one only do with `(~)` however, we cannot derive `a ~ b ~ False`.
or \((a \equiv b) \sim 'False\) by the very same reason why we cannot use \((\equiv\equiv)\) in the definition of \(\text{sFindIndex}\).

Anyway, since those three equalities coincides extensionally, we can derive instance definitions of \(\text{SEqual}\) from either \(\text{TestEquality}\) (from base) or \(\text{SEqual}\). One can of course directly implement \(\text{SEqual}\) by pattern matching. For example:

```haskell
instance \text{SEqual} \text{Plugin}\ where
  \text{SDoubler} \%\text{SDoubler} = \text{Equal}
  \text{SDoubler} \%\text{SGreeter} = \text{NonEqual}
  \text{SGreeter} \%\text{SGreeter} = \text{Equal}
  \text{SGreeter} \%\text{SDoubler} = \text{NonEqual}
```

For \(\text{StoreKey}\), however, we need some hacks as it involves compound constructors:

```haskell
instance \text{SEqual} \text{StoreKey}\ where
  \text{SIntStore} \%\text{SIntStore} = \text{Equal}
  \text{SIntStore} \%\text{SName} = \text{NonEqual}
  \text{SIntStore} \%\text{SPStore} \{} = \text{NonEqual}
  ... \text{Similar for} \text{SName} \%\text{x} ...
  \text{SPStore} p \%\text{SPStore} q = \text{case} p \% q of
    \text{Equal} -> \text{Equal}
    \text{NonEqual} -> \text{nestNonEqual}
  \text{SPStore} \{} \%\text{SIntStore} = \text{NonEqual}
  \text{SPStore} \{} \%\text{SName} = \text{NonEqual}
```

The reason we cannot use \(\text{NonEqual}\) in the \(\text{SPStore}\) \(\sim\) case is that, as state before, \((\equiv\equiv)\) won’t inspect inside compound types. On the other hand, since \((\equiv\equiv)\) from base can correctly handle compound types, we can call combinator \(\text{nestNonEqual}\) here. By constraining with \(a \equiv b \sim 'False\) it tries to reject illegal usages as much as possible.

With \(\text{SEqual}\) above, we can now implement \(\text{sFindIndex}\):

```haskell
\text{sFindIndex} :: \text{SEqual} a \Rightarrow \text{Sing} \ (k :: a)
  \Rightarrow \text{SList} \text{keys} \Rightarrow \text{SMaybe} \ (\text{FindIndex k} \text{keys})
\text{sFindIndex} _ \text{SNil} = \text{SNil}
\text{sFindIndex} k \ (\text{SCons} k' \text{ks}) =
  \text{case} k \% k' \ of
    \text{Equal} -> \text{SJust SHere}
    \text{NonEqual} -> \text{SHere} \%<$> \text{sFindIndex k ks}
```

As this filled the last gap, we can now implement our last goal \(\text{processStoreDynamic}\) as listed in Listing 4.

```haskell
\begin{lstlisting}
data \text{PluginsOn} \text{keys} \ where
    \text{MkSomePlugins} :: \text{All} \ (\text{RunsOn} \text{keys}) \text{ps}
      \Rightarrow \text{SPlugins} \text{ps} \Rightarrow \text{PluginsOn} \text{keys}
\text{toList} :: \text{forall} \text{keys} \Rightarrow \text{Known} \text{keys}
      \Rightarrow \text{Either} \text{String} \ (\text{PluginsOn} \text{keys}) \text{ps}
      \Rightarrow \text{MkSomePlugins} \text{ps}
      \Rightarrow \text{toList} \text{ps} \Rightarrow \text{toList} \text{ps}
\text{toList} [] = \text{pure} \$ \text{MkSomePlugins} \text{SNil}
\text{toList} (p : rest) = \text{do}
  \text{MkSomePlugins} \text{ps} \leftarrow \text{toList} \text{ps} \text{keys rest}
  \text{withPromoted} \text{p} \text{\$ \case}
    \text{SDoubler} \Rightarrow
      \text{deferDynamicPlugin}
        (\text{Proxy} @ '\text{Doubler}')(\text{Proxy} \text{keys})
        (\text{MkSomePlugins} \text{\$ SCons SDoubler} \text{ps})
    \text{SGreeter} \Rightarrow
      \text{deferDynamicPlugin}
        (\text{Proxy} @ '\text{Greeter}')(\text{Proxy} \text{keys})
        (\text{MkSomePlugins} \text{\$ SCons SGreeter} \text{ps})
\text{processStoreDynamic} :: \text{forall} \text{keys. Known} \text{keys}
      \Rightarrow \text{Store} \text{keys} \Rightarrow \text{[Plugin]}
      \Rightarrow \text{Either} \text{String} \ (\text{SProcRec} \text{RunsOn} \text{keys} \text{POutput})
\text{processStoreDynamic} \text{store} \text{ps} = \text{do}
  \text{MkSomePlugins} \ (\text{sps :: SPlugins} \text{ps})
  \leftarrow \text{toList} \text{ps} \text{keys rest}
  \text{pure} \$ \text{MkSomeRec} \text{sps} \text{ps} \text{processStore} \text{store} \text{sps}
\end{lstlisting}
```

Listing 4. The implementation of \text{processStoreDynamic}

Now, we can test \(\text{processStoreDynamic}\):

```haskell
>>> \text{processStoreDynamic}
  (\text{MkStoreEntry '@Name} "\text{Superman}"
    \Rightarrow \text{MkStoreEntry '@IntStore} 42
    \Rightarrow \text{EmptyRecord})
  \text{[Doubler, Greeter]} \text{-- it is a value!}
\Right \text{OutputA} 84
  \Rightarrow \text{GreetOutput} "\text{Hi, Superman, from Greeter!}"\n  \Rightarrow \text{EmptyRecord})
```

Left "Doubler requires IntStore key"

4.3 Summary

We discussed the design of a statically-typed plugin system with dynamic instance resolution at runtime. This was achieved by combining witness-aware constraint handling and the
Deferrable class. We also discussed the design of the equality witness that can treat three distinct type-level equalities.

5 Conclusions

We demonstrated how we could use the constructive point of view, paying attention to witnesses, as a useful guiding principle in designing embedded dependent type-systems in Haskell. As a concrete example, we have demonstrated:

1. Type-level arguments witnessing evaluation paths in a type-family enables us to safely write corresponding singletonised function much easier afterwards.
2. Disjunctions of type constraints can be emulated if it is recoverable from witnesses that is statically computable at type-level.
3. Combining witnesses with Deferrable class, we can implement a dependently-typed plugin system, which can be dynamically type-checked at runtime.
4. It is convenient to provide a unified witness type for extensionally equivalent, but not definitionally equivalent constraints; we took type-level equalities as an example.

To summarise, what we demonstrated in this paper is not a single method, but an insightful way of thinking when one designs dependently typed programs.

5.1 Related and future works

The examples in this paper incorporates many existing works on dependent types in Haskell: those include type-families [7], data-type promotion [17], singletons [2], Implicit Configuration [8], to name a few. The Hasochism paper [11] contains many examples of dependently-typed programming and obstacles in Haskell. We demonstrated how useful the constructive point of view is when we write dependently-typed codes incorporating these prominent methods. There is a minor difference in the direction of interest, though. In dependently-typed programming in Haskell, the method to promote functions and data-types is widely discussed [2, 11, 17]. On the other hand, many examples in the present paper arise when one tries to demote type-level hacks down to the expression-level. Such needs of demotion arise when one wants to resolve dependently-typed constraints at runtime, as we saw in Section 4. The demotion-based approach has another advantage compared to the promotion-based one: the behaviours of macros for deriving definitions are much predictable. For example, in singletons [1] package, there are plenty of macro-generated type-families with names with seemingly random suffixes. This makes, for example, writing type-checker plugins to work with them rather tedious.

We have used user-defined witnesses to carry instance information, such as singletons and equality witness. If once coherent explicit dictionary applications [16] get implemented in GHC, we will be able to directly treat instance dictionaries as another kind of witnesses.

Since the contribution of this paper is a general point of view, there is much room for exploration of synergy with other methods with dependent types. For example, the method of Ghosts of Departed Proofs [12] shares the witness-aware spirit with the present paper. It suggests aggressive uses of phantom types to achieve various levels of type-safety. It can be interesting to explore the synergy of such methods with our examples. For example, we can use the Deferrable class with default instances instead of Given to switch selection strategies based on phantom type parameters.

The compilation performance matters much when one tries to apply involved type-level hacks in the industry and there are many possibilities for exploration in this direction. For example, suppose we promote container types, such as trees, and provide basic construction as a type-function, rather than data-constructors. Then, as the current GHC doesn’t come with the ability to inline type-level terms, it can take so much time to normalise such type-level constructions when they appear repeatedly. In such cases, the compiler plugins to expand type families at compile time can help to improve the compilation time; but this is only a partial workaround and more investigations must be taken.

Acknowledgments

Many of techniques presented in this paper has been found while developing products at DeepFlow, Inc. I thank my employer and colleagues for fruitful discussions and their challenging spirit which allows me to use type-level features so aggressively.

References

[1] Richard Eisenberg. 2012–2013. singletons. http://hackage.haskell.org/package/singletons
[2] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependent Typed Programming with Singletons. SIGPLAN Not. 47, 12 (Sept. 2012), 117–130. https://doi.org/10.1145/2430532.2364522
[3] GHC Team. 2020. type checker Wiki Glasgow Haskell Compiler / GHC GitLab. https://gitlab.haskell.org/gc/gc/wikis/plugins/type-checker
[4] GHC Team. 2021. Data.Type.Equality. https://hackage.haskell.org/package/base-4.14.1.0/docs/src/Data.Type.Equality.html
[5] Hiromi Ishii. 2021. konn/demotion-paper. https://github.com/konn/demotion-paper
[6] Fumiaki Kinoshita. 2020. extensible: Extensible, efficient, optics-friendly data types and effects. https://hackage.haskell.org/package/extensible
[7] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. 2010. Fun with Type Functions. Springer, 301–331. https://doi.org/10.1007/978-1-84882-912-1_14
[8] Oleg Kiselyov and Chung-chieh Shan. 2004. Functional pearl: Implicit configurations - Or, type classes reflect the values of types. (01 2004), 33–44. https://doi.org/10.1145/1017472.1017481
[9] Edward A. Kmett. 2020. constraints: Constraint manipulation. https://hackage.haskell.org/package/constraints
[10] Edward A. Kmett. 2020. The reflection package. https://hackage.haskell.org/package/reflection
[11] Sam Lindley and Conor McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Boston, Massachusetts, USA) (Haskell ’13). ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/2503778.2503786

[12] Matt Noonan. 2018. Ghosts of Departed Proofs (Functional Pearl). In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (St. Louis, MO, USA) (Haskell 2018). Association for Computing Machinery, New York, NY, USA, 119–131. https://doi.org/10.1145/3242744.3242755

[13] Jonathan Sterling. 2020. vinyl: Extensible Records. https://hackage.haskell.org/package/vinyl

[14] Alexander Thiemann. 2020. superrecord: Supercharged anonymous records. https://hackage.haskell.org/package/superrecord

[15] Philip Wadler. 2015. Propositions as Types. Commun. ACM 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407

[16] Thomas Winant and Dominique Devriese. 2018. Coherent Explicit Dictionary Application for Haskell. In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (St. Louis, MO, USA) (Haskell 2018). Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3242744.3242752

[17] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (Philadelphia, Pennsylvania, USA) (TLDI ’12). ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795