A reflection on types*

Simon Peyton Jones[1], Stephanie Weirich[2], Richard A. Eisenberg[2], and Dimitrios Vytiniotis[1]

[1] Microsoft Research, Cambridge
[2] Department of Computer and Information Science, University of Pennsylvania

Abstract. The ability to perform type tests at runtime blurs the line between statically-typed and dynamically-checked languages. Recent developments in Haskell's type system allow even programs that use reflection to themselves be statically typed, using a type-indexed runtime representation of types called *TypeRep*. As a result we can build dynamic types as an ordinary, statically-typed library, on top of *TypeRep* in an open-world context.

1 Preface

If there is one topic that has been a consistent theme of Phil Wadler's research career, it would have to be *types*. Types are the heart of the Curry-Howard isomorphism, occupying the intersection of *logic* and *practical programming*. Phil has always been fascinated by this remarkable dual role, and many of his papers explore that idea in more detail.

One of his most seminal ideas was that of *type classes*, which (with his student Steve Blott) he proposed, fully-formed, to the Haskell committee in February 1988 [WB89]. At that time we were wrestling with the apparent compromises necessary to support equality, numerics, serialisation, and similar functions that have type-specific, rather than type-parametric, behaviour. Type classes completely solved that collection of issues, and we enthusiastically adopted them for Haskell [HHPJW07]. What we did not know at the time is that, far from being a niche solution, type classes would turn out to be a seed bed from which would spring all manner of remarkable fruit: before long we had multi-parameter type classes; functional dependencies; type classes over type constructors (notably the *Monad* and *Functor* classes, more recently joined by a menagerie of *Foldable*, *Traversable*, *Applicative* and many more); implicit parameters, derivable classes, and more besides.

One particular class that we did not anticipate, although it made an early appearance in 1990 , was *Typeable*. The *Typeable* class gives Haskell a handle on *reflection*: the ability to examine types at runtime and take action based on those tests. Many languages support reflection in some form, but Haskell is moving steadily towards an unusually statically-typed form of reflection, contradictory

* This paper appears in the proceedings of Phil Wadler's 60th birthday festschrift, Edinburgh, April 2016.

though that sounds, since reflection is all about dynamic type tests. That topic is the subject of this paper, a reflection on types in homage to Phil.

2 Introduction

Static types are the world's most successful formal method. They allow programmers to specify properties of functions that are proved on every compilation. They provide a design language that programmers can use to express much of the architecture of their programs before they write a single line of algorithmic code. Moreover this design language is not divorced from the code but part of it, so it cannot be out of date. Types dramatically ease refactoring and maintenance of old code bases.

Type systems should let you say what you mean. Weak type systems get in the way, which in turn give types a bad name. For example, no one wants to write a function to reverse a list of integers, and then duplicate the code to reverse a list of characters: we need polymorphism! This pattern occurs again and again, and is the motivating force behind languages that support sophisticated type systems, of which Haskell is a leading example.

And yet, there comes a point in every language at which the static type system simply cannot say what you want. As Leroy and Mauny put it *"there are programming situations that seem to require dynamic typing in an essential way"* [LM91]. How can we introduce dynamic typing into a statically typed language without throwing the baby out with the bathwater? In this paper we describe how to do so in Haskell, making the following contributions:

- We motivate the need for dynamic typing (Section 3), and why it needs to work in an *open* world of types (Section 4). Supporting an open world is a real challenge, which we tackle head on in this paper. Many other approaches are implicitly closed-world, as Section 9 discusses.
- Dynamic typing requires a runtime test of type equality, so some structure that represents a type—a *type representation*—must exist at runtime, along with a way to get the type representation for a value. We describe a *type-indexed* form of type representation, *TypeRep a* (Sections 5.1 and 5.2), and explain how to use it for a type-safe dynamic type test (Section 5.3).
- We show that simply *comparing* type representations is not enough; in some applications we must also safely *decompose* them (Section 5.4).
- Rather unexpectedly, it turns out that supporting decomposition for type representations requires GADT-style *kind equalities*, a feature that has only just been added to GHC 8.0 (Section 5.5). Type-safe reflection requires a very sophisticated type system indeed!

Our key result is a way to build open-world dynamic typing as an ordinary statically-typed library (i.e. not as part of the trusted code base), using a very small (trusted) reflection API for *TypeRep*. We also briefly discuss our implementation (Section 6), metatheory (Section 7), and other applications (Section 8), before concluding with a review of related work (Section 9). This paper is literate Haskell and our examples compile under GHC 8.0.

3 Dynamic types in a statically typed language

Haskell's type system is so expressive that it is remarkably hard to find a compelling application for dynamic typing. But here is one. Suppose you want to write a Haskell library to implement the following familiar state-monad API[3,4]:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{ST}\ s\ a & \text{-- Abstract type for state monad} \\
\textbf{data}\ \mathit{STRef}\ s\ a & \text{-- Abstract type for references (to value of type a)} \\
\mathit{runST} & :: (\forall\ s.\ \mathit{ST}\ s\ a) \to a \\
\mathit{newSTRef} & :: a \to \mathit{ST}\ s\ (\mathit{STRef}\ s\ a) \\
\mathit{readSTRef} & :: \mathit{STRef}\ s\ a \to \mathit{ST}\ s\ a \\
\mathit{writeSTRef} & :: \mathit{STRef}\ s\ a \to a \to \mathit{ST}\ s\ ()
\end{array}
$$

Papers about state monads usually assume that the implementation is built in, but what if it were not? This is not a theoretical question: actively-used Haskell libraries, such as *vault*[5] face exactly this challenge. To implement *ST* we need some kind of "store" that maps a key (a *STRef*) to its value. This *Store* should have the following API (ignore the *Typeable* constraints for now):

$$
\begin{array}{l}
\mathit{extendStore} :: \mathit{Typeable}\ a \Rightarrow \mathit{STRef}\ s\ a \to a \to \mathit{Store} \to \mathit{Store} \\
\mathit{lookupStore} :: \mathit{Typeable}\ a \Rightarrow \mathit{STRef}\ s\ a \to \mathit{Store} \to \mathit{Maybe}\ a
\end{array}
$$

It makes sense to implement the *Store* by a finite map, keyed by *Int* or some other unique key, which itself is kept inside the *STRef*. For that purpose, we can use the standard Haskell type *Map k v*, mapping keys *k* to values *v*. But what type should *v* be? As the type of *extendStore* declares, we must be able to insert *any* type of value into the *Store*. This is where type *Dynamic* is useful:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Key} = \mathit{Int} \\
\textbf{data}\ \mathit{STRef}\ s\ a = \mathit{STR}\ \mathit{Key} \\
\textbf{type}\ \mathit{Store} = \mathit{Map}\ \mathit{Key}\ \mathit{Dynamic}
\end{array}
$$

Dynamic suffices if we have the following operations available, used to create and take apart *Dynamic* values:[6]

$$
\begin{array}{ll}
\mathit{toDynamic} & :: \mathit{Typeable}\ a \Rightarrow a \to \mathit{Dynamic} \\
\mathit{fromDynamic} & :: \mathit{Typeable}\ a \Rightarrow \mathit{Dynamic} \to \mathit{Maybe}\ a
\end{array}
$$

[3] There is another connection with Phil's work here: an API like this was first proposed in *"Imperative functional programming"* [PJW93], a collaboration between one of the present authors and Phil, directly inspired by Phil's ground-breaking paper *"Comprehending monads"* [Wad90].

[4] For our present purposes you can safely ignore the "*s*" type parameter; the paper *"State in Haskell"* explains what is going on [LPJ95].

[5] `https://hackage.haskell.org/package/vault`

[6] These types also motivate the *Typeable* contraint above. We discuss that constraint further in Section 5.2, but without looking that far ahead, Phil's insight about "theorems for free" tells us that the type $\mathit{fromDynamic} :: \mathit{Dynamic} \to \mathit{Maybe}\ a$ is a non-starter [Wad89]. Any function with that type must return *Nothing*, *Just* $\perp$, or diverge.

We can now implement the functions on *Store*, thus:

$$
\begin{aligned}
&\mathit{extendStore}\ (\mathit{STR}\ k)\ v\ s = \mathit{insert}\ k\ (\mathit{toDynamic}\ v)\ s\\
&\mathit{lookupStore}\ (\mathit{STR}\ k)\quad s = \textbf{case}\ \mathit{lookup}\ k\ s\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ d \quad \to \mathit{fromDynamic}\ d\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{Nothing}
\end{aligned}
$$

In *lookupStore* the dynamic type check made by *fromDynamic* will always succeed. (That is, when looking up a *STRef s a*, we should always find a value of type *a*.) The runtime tests compensate where the static type system is inadequate.

In summary, there are a few occasions when even a type system as sophisticated as Haskell's is not powerful enough to give the static guarantees we seek. A *Dynamic* type, equipped with *toDynamic* and *fromDynamic*, can plug the gap.

4 The challenge of an open world

Where does type *Dynamic* come from? One classic approach is to make *Dynamic* a tagged union of all the types we care about, like this:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Dynamic} = \mathit{DInt}\ \mathit{Int}\\
&\qquad\qquad\quad\ \mid \mathit{DBool}\ \mathit{Bool}\\
&\qquad\qquad\quad\ \mid \mathit{DChar}\ \mathit{Char}\\
&\qquad\qquad\quad\ \mid \mathit{DPair}\ \mathit{Dynamic}\ \mathit{Dynamic}\\
&\qquad\qquad\qquad \ldots\\
&\mathit{toDynInt} :: \mathit{Int} \to \mathit{Dynamic}\\
&\mathit{toDynInt} = \mathit{DInt}\\
&\mathit{fromDynInt} :: \mathit{Dynamic} \to \mathit{Maybe}\ \mathit{Int}\\
&\mathit{fromDynInt}\ (\mathit{DInt}\ n) = \mathit{Just}\ n\\
&\mathit{fromDynInt}\ \_ \qquad\quad\ = \mathit{Nothing}\\
&\mathit{toDynPair} :: \mathit{Dynamic} \to \mathit{Dynamic} \to \mathit{Dynamic}\\
&\mathit{toDynPair} = \mathit{DPair}\\
&\mathit{dynFst} :: \mathit{Dynamic} \to \mathit{Maybe}\ \mathit{Dynamic}\\
&\mathit{dynFst}\ (\mathit{DPair}\ x1\ x2) = \mathit{Just}\ x1\\
&\mathit{dynFst}\ \_ \qquad\qquad\quad\ = \mathit{Nothing}
\end{aligned}
$$

For each type constructor (*Int*, *Bool*, pairs, lists, etc) we define a data constructor (e.g. *DPair*) in the *Dynamic* type, plus a constructor function (e.g. *toDynPair*) and one or more destructors (e.g. *dynFst*, *dynSnd*).

This approach has a fundamental shortcoming: it is not extensible. Concretely, what is the "..." above? The data type declaration for *Dynamic* can enumerate only a fixed set of type constructors (integers, booleans, pairs, etc)[7]. We call this the *closed-world assumption*. Sometimes a closed world is acceptable.

[7] Although the set of type *constructors* is fixed, you can use them to build an infinite number of *types*; e.g. $(\mathit{Int}, \mathit{Bool})$, $(\mathit{Int}, (\mathit{Bool}, \mathit{Int}))$, etc.

For example, if we were writing an evaluator for a small language we would need *Dynamic* to have only enough data constructors to represent the types of the object language.

But in general the world is simply not closed; it is unreasonable to extend *Dynamic*, whenever the user defines a new data type! In the *ST* example, it is fundamental that the monad be able to store values of user-declared types.

So in this paper we focus exclusively on the challenge of open-world extensibility. Before moving on, it is worth noting that a surprisingly large fraction of the academic literature on dynamics in a statically typed language makes a closed-world assumption (see Section 9). Moreover, even if we accept a closed world, the approach sketched above has other difficulties, discussed in Section 9.2.

5 *TypeRep*: runtime reflection in an open world

We can implement an open-world *Dynamic* as an ordinary, type-safe Haskell library, on top of a new abstraction, that of *type-indexed type representations* or *TypeRep*. In fact, Haskell has supported (un-type-indexed) type representations and an open-world *Dynamic* for years, but in a rather unsatisfactory way (the "old design"). However, recent developments in Haskell's type system—notably GADTs [XCC03, PJVWW06], kind polymorphism [YWC$^+$12], and kind equalities [WHE13]—have opened up new opportunities (the "new design"). A major purpose of this paper is to motivate and describe this new design. For readers familiar with the old design, we compare it with the new one in Section 9.1.

5.1 Introducing *TypeRep*

The key to our approach is our type-indexed type representation *TypeRep*. But what is a type-indexed type representation? It is best understood by example:

- The representation of *Int* is the value of type *TypeRep Int*.
- The representation of *Bool* is the value of type *TypeRep Bool*.
- And so on.

That is, the index in a type-indexed type representation is itself the represented type. *TypeRep* is abstract, and thus we don't write the *TypeRep* value in the examples above. Note, however, that we have said *the* value, not *a* value—there is precisely one value of type *TypeRep Int*[8]. *TypeRep* thus defines a family of singleton types [EW12]; indeed, *TypeRep* is the singleton family associated with the kind $\star$ of types.

As we build out the API for *TypeRep*, we will consider how to build an efficient, type-safe, and open-world implementation of *Dynamic*. Converting to and from *Dynamic* should not touch the value itself; instead we represent a dynamic value as a pair of a value and a runtime-inspectable representation of its type. Thus:

[8] Recall that $\perp$ is not a value.

```
data Dynamic where
  Dyn :: TypeRep a → a → Dynamic
```

Here we are using GADT-style syntax to declare the constructor *Dyn*, whose payload includes a runtime representation of a type *a* and a value of type *a*. The type *a* is existentially bound; that is, it does not appear in the result type of the data constructor.

Now we have two challenges: where do we get the *TypeRep* from when creating a *Dynamic* in *toDynamic* (Section 5.2); and what do we do with it when unpacking it in *fromDynamic* (Section 5.3)?

5.2 The *Typeable* class

Because each type has its own *TypeRep*, the obvious approach is to use a type class, thus:

```
class Typeable a where
  typeRep :: TypeRep a
```

This class has only one operation, a nullary function (or simple value) that is the type representation for the type. Now we can write *toDynamic*:

```
toDynamic :: Typeable a ⇒ a → Dynamic
toDynamic x = Dyn typeRep x
```

The type of the data constructor *Dyn* ensures that that the call of *typeRep* produces a type representation for the type *a*. Easy!

But where do *instances* of *Typeable* come from? The magic of type classes gives us a simple way to solve the open-world challenge, by using a single piece of built-in compiler support: every data type declaration gives rise to a *Typeable* instance for that type (Section 6). Furthermore, because *Typeable* and its instances are built in, we can be sure that these representations uniquely define types; for example, the user cannot write bogus instances of *Typeable* that use the same *TypeRep* for two different types.

5.3 Type-aware equality for *TypeRep*s

The second challenge is to unpack dynamics. We need a function with this signature:

```
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a
```

The function *fromDynamic* takes a *Dynamic* and tests whether it wraps a value of the desired type; if so, it returns the value wrapped in *Just*; if not, it returns *Nothing*. The *Typeable* constraint allows *fromDynamic* to know what the “desired type” is.

But how is *fromDynamic* implemented? Patently it must compare type representations, so we might try this:

$$
\begin{aligned}
&\mathit{fromDynamic} :: \forall\, d.\ \mathit{Typeable}\ d \Rightarrow \mathit{Dynamic} \rightarrow \mathit{Maybe}\ d\\
&\mathit{fromDynamic}\ (\mathit{Dyn}\ (\mathit{ra} :: \mathit{TypeRep}\ a)\ (x :: a))\\
&\quad \mid \mathit{rd} == \mathit{ra}\ = \mathit{Just}\ x \quad \text{-- Eeek! Type error!}\\
&\quad \mid \mathit{otherwise} = \mathit{Nothing}\\
&\quad \textbf{where}\\
&\qquad \mathit{rd} = \mathit{typeRep} :: \mathit{TypeRep}\ d
\end{aligned}
$$

The type signatures for *ra* and *x* could be omitted, but we have put them in to remind ourselves that the types of *ra* and *x* are connected through the existentially-bound type *a*. The value *rd* is (the runtime representation of) the "desired type", disambiguated by a type signature. We compare *rd* with *ra*, (the representation of) *x*'s type, and return *Just x* if they match. The operational behaviour is just what we want, but the type checker will reject it. It has no reason to believe that *x* actually has type *d*: the type-checker surely does not understand that we have just compared *TypeRep*s linking up *x*'s type with *d*.

Fortunately, we have a fine tool to use whenever a runtime operation needs to inform us about types: generalised algebraic data types, or GADTs. We need an equality on *TypeRep* that returns a GADT; pattern-matching on the return value gives the type-checker just the information it needs. Here are the definitions[9]:

$$
\begin{aligned}
&\mathit{eqT} :: \mathit{TypeRep}\ a \rightarrow \mathit{TypeRep}\ b \rightarrow \mathit{Maybe}\ (a :\approx: b)\\
&\qquad \text{-- Primitive; implemented by compiler}\\
&\textbf{data}\ a :\approx: b\ \textbf{where}\\
&\quad \mathit{Refl} :: a :\approx: a
\end{aligned}
$$

Here *eqT* returns *Nothing* if the two *TypeRep*s are different, and (*Just Refl*) if they are the same. The data constructor *Refl* is the sole, nullary data constructor of the GADT ($a :\approx: b$). Pattern matching on *Refl* tells the type checker that the two types are the same. In short, when the argument types are equal, *eqT* returns a proof of this equality, in a form that the type checker can use.

To be concrete, here is the definition of *fromDynamic*:

$$
\begin{aligned}
&\mathit{fromDynamic} :: \forall\, d.\ \mathit{Typeable}\ d \Rightarrow \mathit{Dynamic} \rightarrow \mathit{Maybe}\ d\\
&\mathit{fromDynamic}\ (\mathit{Dyn}\ (\mathit{ra} :: \mathit{TypeRep}\ a)\ (x :: a))\\
&\quad = \textbf{case}\ \mathit{eqT}\ \mathit{ra}\ (\mathit{typeRep} :: \mathit{TypeRep}\ d)\ \textbf{of}\\
&\qquad\quad \mathit{Nothing}\ \ \rightarrow \mathit{Nothing}\\
&\qquad\quad \mathit{Just}\ \mathit{Refl} \rightarrow \mathit{Just}\ x
\end{aligned}
$$

We use *eqT* to compare the two *TypeRep*s, and pattern-match on *Refl*, so that in the second case alternative we know that *a* and *d* are equal, so we can return *Just x* where a value of type *Maybe d* is needed.

Since *Maybe* is a monad, we can use **do** notation for this code, and instead write it like this:

[9] Here we are using GHC's ability to define infix type constructors.

```
fromDynamic (Dyn ra x)
   = do Refl ← eqT ra (typeRep :: TypeRep d)
        return x
```

When we make multiple matches this style is more convenient, so we use it from now on.

More generally, *eqT* allows to implement *type-safe* cast, a useful operation in its own right [Wei04, LPJ03, LPJ05].

```
cast :: ∀ a b. (Typeable a, Typeable b) ⇒ a → Maybe b
cast x = do Refl ← eqT (typeRep :: TypeRep a)
                       (typeRep :: TypeRep b)
            return x
```

5.4 Decomposing type representations

So far, the only operation we have provided over *TypeRep* is *eqT*, which compares two type representations for equality. But that is not enough to implement *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn pab x)
  =   -- Check that pab represents a pair type
      -- Take (fst x) and wrap it in Dyn
```

How can we decompose the type representation *pab*, to check that it indeed represents a pair, and extract its first component? Since types in Haskell are built via a sequence of type applications (much like how an expression applying a function to multiple arguments is built with several nested term applications), the natural dual is to provide a way to decompose type applications:

```
splitApp :: TypeRep a → Maybe (AppResult a)
    -- Primitive; implemented by compiler
data AppResult t where
  App :: TypeRep a → TypeRep b → AppResult (a b)
```

The primitive operation *splitApp* allows us to observe the structure of types. If *splitApp* is applied to a type constructor, such as *Int*, it returns *Nothing*; otherwise, for a type application, it decomposes one layer of the application, and returns (*Just* (*App ra rb*)) where *ra* and *rb* are representations of the sub-components. Like *eqT*, it returns a GADT, *AppResult*, to expose the type equalities it has discovered to the type checker.

Now we can implement *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn rpab x)
```

```
  = do App rpa rb ← splitApp rpab
       App rp  ra ← splitApp rpa
       Refl       ← eqT rp (typeRep :: TypeRep (,))
       return (Dyn ra (fst x))
```

We check that the type of x, whose *TypeRep*, *rpab*, is of form $(,)\ a\ b$, by decomposing it twice with *splitApp*. Then we must check that *rp*, the *TypeRep* of the function part of this application, is indeed the pair type constructor $(,)$; we can do that using *eqT*. These three GADT pattern matches combine to tell the type checker that the type of x, which began life in the (*Dyn rpab x*) pattern match as an existentially-quantified type variable, is indeed a pair type (a, b). So we can safely apply *fst* to x, to get a result whose type representation *ra* we have in hand.

In the same way we can use *splitApp* to implement *dynApply*, which applies a function *Dynamic* to an argument *Dynamic*, provided the types line up:

```
dynApply :: Dynamic → Dynamic → Maybe Dynamic
dynApply (Dyn rf f) (Dyn rx x) = do
  App ra  rt2 ← splitApp rf
  App rtc rt1 ← splitApp ra
  Refl        ← eqT rtc (typeRep :: TypeRep (→))
  Refl        ← eqT rt1 rx
  return (Dyn rt2 (f x))
```

In both cases, the code is simple enough, but the type checker has to work remarkably hard behind the scenes to prove that it is sound. Let us take a closer look.

5.5 Kind polymorphism and kind equalities

There is something suspicious about our use of *typeRep* :: *TypeRep* $(,)$. So far we have discused type representations for only types of kind $\star$. But $(,)$ has kind $(\star \to \star \to \star)$; does it too have a *TypeRep*? Of course it must, to allow *TypeRep Int*, *TypeRep Maybe*, and *TypeRep* $(,)$. So the type constructor *TypeRep* must be polymorphic in the kind of its type argument, or *poly-kinded*, and so must be its accompanying class *Typeable*, thus:

```
data TypeRep (a :: k)   -- primitive, indexed by type and kind
class Typeable (a :: k) where
  typeRep :: TypeRep a
```

Fortunately, GHC has offered kind polymorphism for some years [YWC+12]. Similarly, the result GADT *AppResult* must be kind-polymorphic. Here is its definition with kind signatures added[10]:

[10] The kind signatures are optional. With *PolyKinds* enabled, GHC infers them, but we often add them for clarity.

```
data AppResult (t :: k) where
  App :: ∀ k1 k (a :: k1 → k) (b :: k1).
         TypeRep a → TypeRep b → AppResult (a b)
```

In *AppResult*, note that k_1, the kind of b, is *existentially* bound in this data structure, meaning that it does not appear in the kind of the result type $(a\ b)$. We know the result kind of the type application but there is no way to know the kinds of the subcomponents.

With kind polymorphism in mind, let's add some type annotations to see what existential variables are introduced by the two calls to *splitApp* in *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn (rpab :: TypeRep pab) (x :: pab))
  = do App (rpa :: TypeRep pa) (rb :: TypeRep b) ← splitApp rpab
              -- introduces kind k2, and types pa :: k2 → ⋆, b :: k2
       App (rp  :: TypeRep p)  (ra :: TypeRep a) ← splitApp rpa
              -- introduces kind k1, and types p :: k1 → k2 → ⋆, a :: k1
       Refl ← eqT rp (typeRep :: TypeRep (,))
              -- introduces p ∼ (,) and (k1 → k2 → ⋆) ∼ (⋆ → ⋆ → ⋆)
       return (Dyn ra (fst x))
```

Focus on the arguments to the call to eqT in the third line. We know that:

- rp :: $TypeRep\ p$ and p :: $k_1 \to k_2 \to \star$
- $typeRep$ :: $TypeRep\ (,)$ and $(,) :: \star \to \star \to \star$

So eqT must compare the *TypeReps* for two types of different kinds; if the runtime test succeeds, we know not only that $p \sim (,)$, but also that $k_1 \sim \star$ and $k_2 \sim \star$. That is, the pattern match on *Refl* GADT constructor brings local *kind equalities* into scope, as well as *type equalities*.

We can make this more explicit by writing out kind-annotated definitions for (:≈:) and eqT, thus:

```
eqT :: ∀ k1 k2 (a :: k1) (b :: k2). TypeRep a → TypeRep b → Maybe (a :≈: b)
data (a :: k1) :≈: (b :: k2) where
  Refl :: ∀ k (a :: k). a :≈: a
```

If the two types are the same, then eqT returns a proof that the types are equal *and* simultaneously a proof that the kinds are equal: a heterogeneous (often referred to as "John Major") equality [McB02].

In the case of *dynFst*, if eqT succeeds, the type checker can conclude $(k_1 \to k_2 \to \star) \sim (\star \to \star \to \star)$ and $p \sim (,)$. The GHC constraint solver uses these equalties to conclude that the type of x is (a, b), validating the projection *fst x*.

The addition of first-class kind equalities, to accompany first-class type equalities, is the most recent innovation in GHC 8.0. Indeed, they motivate a systemic change, namely collapsing types and kinds into a single layer, so that we have

$\star :: \star$. This change is described and motivated in a recent paper [WHE13]. Type-safe decomposition of type representations is a compelling motivation for kind equalities.

5.6 Visible vs. invisible type representations

Here are two functions with practically identical functionality:

$$
\begin{aligned}
&\mathit{cast} \;\; :: (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow a \to \mathit{Maybe}\ b \\
&\mathit{castR} :: \mathit{TypeRep}\ a \to \mathit{TypeRep}\ b \to a \to \mathit{Maybe}\ b
\end{aligned}
$$

A *Typeable* class constraint is represented at runtime by a value argument, a *Typeable* "dictionary" in the jargon of type classes [WB89]. A dictionary is just a record of the methods of the class. Since *Typeable a* has only one method, a *Typeable a* dictionary is represented simply by a *TypeRep a* value. So, in implementation terms the function *cast* actually takes two *TypeRep* arguments exactly like *castR*. It's just that *castR* takes visible *TypeRep* arguments, while *cast* takes invisible (compiler-generated) *Typeable* arguments. So which is "better"?

The answer is primarily stylistic. Sometimes, in library code that manipulates many different *TypeRep* values, it is much clearer to name them explicitly, as we have done in the earlier examples in this section. But in other places (usually client code) it is vastly more convenient to take advantage of type classes to construct relevant *Typeable* evidence.

The two are, of course, equally expressive, since the implementation is the same in either case. Going from an implicit type representation (*Typeable*) to an explicit one (*TypeRep*) is easy, if inscrutable: just use the method *typeRep*. For example, here is how to define *cast* using *castR*:

$$
\begin{aligned}
&\mathit{cast} :: (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow a \to \mathit{Maybe}\ b \\
&\mathit{cast} = \mathit{castR}\ \mathit{typeRep}\ \mathit{typeRep}
\end{aligned}
$$

The two calls to *typeRep* are at different types, but that is not very visible in the code. That is why it is often clearer to pass *TypeRep* values explicitly. But for the *caller* of *cast* is it much easier to pass invisible arguments; for example, in the call:

$$(\mathit{cast}\ x) :: \mathit{Maybe}\ \mathit{Bool}$$

the compiler will construct a *TypeRep* for x's type and one for *Bool*, both wrapped as *Typeable* dictionaries, and pass them to *cast*.

However, going from explicit to implicit is not as easy. Suppose we have a *TypeRep a* and we wish to call a function with a *Typeable a* constraint. We essentially need to invent an **instance** *Typeable a* on the spot. Haskell provides no facility for local instances, chiefly because doing so would imperil class coherence.[11] In the context of type representations, though, incoherence is impossible:

[11] Though, some Haskellers have hacked around this restriction with abandon. See Kiselyov and Shan [KS04] and Edward Kmett's *reflection* package (at `http://hackage.haskell.org/package/reflection`).

there really is only one $\mathit{TypeRep}\ a$ in existence, and so one $\mathit{Typeable}\ a$ instance is surely the same as any other. Our API thus provides the following additional function $\mathit{withTypeable}$, which we can use to close the loop by writing castR in terms of cast:

$$\begin{aligned}
&\mathit{withTypeable} :: \mathit{TypeRep}\ a \to (\mathit{Typeable}\ a \Rightarrow r) \to r\\
&\mathit{castR} :: \mathit{TypeRep}\ a \to \mathit{TypeRep}\ b \to a \to \mathit{Maybe}\ b\\
&\mathit{castR}\ \mathit{ta}\ \mathit{tb} = \mathit{withTypeable}\ \mathit{ta}\ (\mathit{withTypeable}\ \mathit{tb}\ \mathit{cast})
\end{aligned}$$

We cannot implement $\mathit{withTypeable}$ in Haskell source. But we *can* implement it in GHC's statically-typed intermediate language, System FC [SCPJD07]. The definition is simple, roughly like this:

$$\mathit{withTypeable}\ \mathit{tr}\ k = k\ \mathit{tr} \quad \text{-- Not quite right}$$

Its second argument k expects a $\mathit{Typeable}$ dictionary as its value argument. But since a $\mathit{Typeable}$ dictionary is represented by a $\mathit{TypeRep}$, we can simply pass tr to k. When written in System FC there is a type-safe coercion to move from $\mathit{TypeRep}\ a$ to $\mathit{Typeable}\ a$, but that coercion is erased at runtime. Since the definition can be statically type checked, $\mathit{withTypeable}$ does not form part of the trusted code base.

5.7 Comparing *TypeRep*s

It is sometimes necessary to use type representations in the key of a map. For example, Shake [Mit12] uses a map keyed on type representations to look up class instances (dictionaries) at runtime; these instances define class operations for the types of data stored in a collection of $\mathit{Dynamic}$s. Storing the class operations once per type, instead of with each $\mathit{Dynamic}$ package, is much more efficient.[12]

More specifically, we would like to implement the following interface:

$$\begin{aligned}
&\mathbf{data}\ \mathit{TyMap}\\
&\mathit{empty}\ \ :: \mathit{TyMap}\\
&\mathit{insert}\ \ :: \mathit{Typeable}\ a \Rightarrow a \to \mathit{TyMap} \to \mathit{TyMap}\\
&\mathit{lookup} :: \mathit{Typeable}\ a \Rightarrow \mathit{TyMap} \to \mathit{Maybe}\ a
\end{aligned}$$

But how should we implement these type-indexed maps? One option is to use the standard Haskell library $\mathit{Data.Map}$. We can define the typed-map as a map between the type representation and a dynamic value.

$$\begin{aligned}
&\mathbf{data}\ \mathit{TypeRepX}\ \mathbf{where}\\
&\quad \mathit{TypeRepX} :: \mathit{TypeRep}\ a \to \mathit{TypeRepX}\\
&\mathbf{type}\ \mathit{TyMap} = \mathit{Map}\ \mathit{TypeRepX}\ \mathit{Dynamic}
\end{aligned}$$

[12] See also `http://stackoverflow.com/q/32576018/791604` for another use case for a map keyed on type representations.

Notice that we must wrap the *TypeRep* key in an existential *TypeRepX*, otherwise all the keys would be for the same type, which would rather defeat the purpose! The *insert* and *lookup* functions can then use *toDynamic* and *fromDynamic* to ensure that the right type of value is stored with each key.

```
insert   :: ∀ a. Typeable a ⇒ a → TyMap → TyMap
insert x = Map.insert (TypeRepX (typeRep :: TypeRep a)) (toDynamic x)

lookup :: ∀ a. Typeable a ⇒ TyMap → Maybe a
lookup = fromDynamic <=< Map.lookup (TypeRepX (typeRep :: TypeRep a))
```

Because *Data.Map* uses balanced binary trees to achieve efficient lookup, *TypeRepX* must be an instance of *Ord*:

```
instance Ord TypeRepX where
  compare (TypeRepX tr1) (TypeRepX tr2) = compareTypeRep tr1 tr2

compareTypeRep :: TypeRep a → TypeRep b → Ordering   -- primitive
```

The *TypeRep* API includes a comparison function *compareTypeRep* that compares two *TypeRep*s, indexed by possibly-different types *a* and *b*. Notice that we cannot make an instance for *Ord* (*TypeRep a*): if we compare two values both of type *TypeRep t*, following the signature of *compare*, they should always be equal!

Alternatively, we could use a more strongly typed data structure that internally keeps track of the dependency between the key and the element type. A simple example might be the following binary search tree:

```
data TyMap = Empty | Node Dynamic TyMap TyMap
```

Of course, much more general structures are also possible.[13]

Looking up values in this tree requires comparing the ordering of type representations. We could implement this comparison using the ordering for *TypeRepX*, but that implementation is clumsy. Once we have found the value, we must do an extra cast to show that it has the correct type.

```
lookup :: TypeRep a → TyMap → Maybe a
lookup tr1 (Node (Dyn tr2 v) left right) =
  case compareTypeRep tr1 tr2 of
    LT → lookup tr1 left
    EQ → castR tr2 tr1 v   -- know this cast will succeed
    GT → lookup tr1 right
lookup tr1 Empty = Nothing
```

However, we can improve this implementation using the following *more informative* comparison function, thereby avoiding this redundant check. In particular, when the two type representations are equal, this function will return an equality proof, just like *eqT*.

[13] The *dependent-map* library is an example of such a data structure. See `https://hackage.haskell.org/package/dependent-map-0.1.1.3/docs/Data-Dependent-Map.html`.

```
cmpT :: TypeRep a → TypeRep b → OrderingT a b
    -- definition is primitive

data OrderingT a b where
  LTT :: OrderingT a b
  EQT :: OrderingT t t
  GTT :: OrderingT a b
```

It is, of course, trivial to define *compareTypeRep* in terms of *cmpT*.

5.8 Representing polymorphic and kind-polymorphic types

Our interface does not support representations of polymorphic types, such as *TypeRep* ($\forall$ *a*. *a* $\rightarrow$ *a*). Although plausible, supporting those in our setting brings in a whole new range of design decisions that are as of yet unexplored (e.g. higher-order representations vs. de-Bruijn?). Furthermore, it requires the language to support impredicative polymorphism (the ability to instantiate quantified variables with polymorphic types, for instance the *a* variable in *TypeRep a* or *Typeable a*), which GHC currently does not. Finally, representations of polymorphic types have implications on semantics and possibly parametricity, an issue that we discuss in the next section.

Similarly, constructors with polymorphic kinds would require impredicative *kind* polymorphism. A representation of type *TypeRep* (*Proxy* :: $\forall$ *kp*. *kp* $\rightarrow \star$) would require the kind parameter *k* of *TypeRep* (*a* :: *k*) to be instantiated to the polymorphic kind $\forall$ *kp*. *kp* $\rightarrow \star$. Type inference for impredicative kind polymorphism is no easier than for impredicative type polymorphism and we have thus excluded this possibility.

5.9 Summary

It is time to draw breath. We used the *ST* example to motivate a *Dynamic* type (Section 3); then we used *Dynamic* to motivate type-indexed type representations (Section 5.1); and in the rest of Section 5 we have discussed the various operations we need over those representations. Our final API for *TypeRep* is summarised in Figure 1.

Why do we make *TypeRep* primitive rather than *Dynamic*? When we design a primitive, built-in feature for a language, we seek the smallest, most sharply-focused feature that serves the need. We need built-in support for something like *TypeRep* and the *Typeable* class to implement *Dynamic*. If the *Dynamic* library becomes just an ordinary library, with no uses of *unsafeCoerce*, that usefully shrinks the trusted code base. Moreover, *TypeRep* is independently useful to support other abstractions (not just *Dynamic*), as we describe in Section 8.

```
-- Related definitions ----------------------------------------

-- Informative propositional equality
data (a :: k1) :≈: (b :: k2) where
  Refl :: ∀ k (a :: k). a :≈: a

-- An informative ordering type, asserting type equality in the EQ case
data OrderingT a b where
  LTT :: OrderingT a b
  EQT :: OrderingT t t
  GTT :: OrderingT a b

-- Data.Typeable ----------------------------------------
data TypeRep (a :: k)
    -- primitive, indexed by type and kind

instance Show (TypeRep a)

class Typeable (a :: k) where
  typeRep :: TypeRep a
    -- Typeable instances automatically generated for all type constructors

-- class access
withTypeable :: TypeRep a → (Typeable a ⇒ r) → r

-- existential version
data TypeRepX where
  TypeRepX :: TypeRep a → TypeRepX
instance Eq   TypeRepX
instance Ord  TypeRepX
instance Show TypeRepX

-- comparison
eqT  :: TypeRep a → TypeRep b → Maybe (a :≈: b)
cmpT :: TypeRep a → TypeRep b → OrderingT a b

-- construction
mkTyApp :: TypeRep a → TypeRep b → TypeRep (a b)

-- pattern matching
splitApp :: TypeRep a → Maybe (AppResult a)

data AppResult (t :: k) where
  App :: TypeRep a → TypeRep b → AppResult (a b)

-- information about the "head" type constructor
tyConPackage :: TypeRep a → String
tyConModule  :: TypeRep a → String
tyConName    :: TypeRep a → String
```

Fig. 1. New Typeable interface

6 Implementation

How do we implement type representations? We use a GADT like this:

$$
\begin{array}{l}
\textbf{data}\ \mathit{TypeRep}\ (a :: k)\ \textbf{where} \\
\quad \mathit{TrApp} \quad :: \mathit{TypeRep}\ a \to \mathit{TypeRep}\ b \to \mathit{TypeRep}\ (a\ b) \\
\quad \mathit{TrTyCon} :: \mathit{TyCon} \to \mathit{TypeRep}\ k \to \mathit{TypeRep}\ (a :: k) \\
\textbf{data}\ \mathit{TyCon} = \mathit{TyCon}\ \{\mathit{tc\_module} :: \mathit{Module}, \mathit{tc\_name} :: \mathit{String}\} \\
\textbf{data}\ \mathit{Module} = \mathit{Module}\ \{\mathit{mod\_pkg} :: \mathit{String}, \mathit{mod\_name} :: \mathit{String}\}
\end{array}
$$

The *TyCon* type is a runtime representation of the "identity" of a type constructor. For every datatype declaration, GHC silently generates a binding for a suitable *TyCon*. For example, for *Maybe* GHC will generate:

$$
\begin{array}{l}
\$\mathit{tcMaybe} :: \mathit{TyCon} \\
\$\mathit{tcMaybe} = \mathit{TyCon}\ \{\mathit{tc\_module} = \mathit{Module}\ \{\mathit{mod\_pkg} \quad = \texttt{"base"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , \mathit{mod\_name} = \texttt{"Data.Maybe"}\} \\
\qquad\qquad\qquad\quad , \mathit{tc\_name} \quad = \texttt{"Maybe"}\}
\end{array}
$$

The name $\$\mathit{tcMaybe}$ is not directly available to the programmer. Instead (this is the second piece of built-in support), GHC's type-constraint solver has special behaviour for *Typeable* constraints, as follows.

To solve *Typeable* $(t_1\ t_2)$, GHC simply solves *Typeable* t_1 and *Typeable* t_2, and combines the results with *TrApp*. To solve *Typeable* *T* where *T* is a type constructor, the solver uses *TrTyCon*. The first argument of *TrTyCon* is straightforward: it is the (runtime representation of the) type constructor itself, e.g. $\$\mathit{tcMaybe}$.

But *TrTyCon* also stores the representation of the kind of this very constructor, of type *TypeRep k*. Recording the kind representations is important, otherwise we would not be able to distinguish, say, $\mathit{Proxy} :: \star \to \star$ from $\mathit{Proxy} :: (\star \to \star) \to \star$, where *Proxy* has a polymorphic kind ($\mathit{Proxy} :: \forall\ k.\ k \to \star$). We do not support direct representations of kind-polymorphic constructors like *Proxy*, for reasons outlined in Section 5.8; rather *TrTyCon* encodes the *instantiation* of a kind-polymorphic constructor (such as *Proxy*). There is a limitation here: the kind of the instantiated type constructor must be monomorphic (again, for reasons outlined in Section 5.8), so (a) the type constructor must have a rank-1 prenex-polymorphic kind, and (b) it can be instantiated only with monomorphic kinds (i.e. predicatively).

Notice that *TrTyCon* is fundamentally insecure: you could use it to build a *TypeRep t* for any *t* whatsoever. That is why we do not expose the representation of *TypeRep* to the programmer. Instead the part of GHC's *Typeable* solver that builds *TrTyCon* applications is part of GHC's trusted code base.

Another reason for keeping *TypeRep* abstract is that it allows us to vary details of the representation. For example, the SYB pattern (Section 8.2) makes many equality tests for *TypeRep*. If we stored a fingerprint, or even a hash-value, in every node, we could make comparisons work in constant time. Another aspect that we might want to vary is how much meta-data we make available in a *TyCon*.

7 Properties of a language with reflection

The addition of features such as *Dynamic* and *TypeRep* has implications to the semantics and metatheory of a programming language.

Strong normalization The addition of *Dynamic* (whether implemented with *TypeRep* or not) creates the possibility for loops without explicitly using recursion (nor recursive data types):

```
delta :: Dynamic → Dynamic
delta dn = case fromDynamic dn of
  Just f   → f dn
  Nothing → dn
loop = delta (toDynamic delta)
```

Effectively *Dynamic* behaves like a negative recursive datatype, a feature that breaks strong normalization.

A similar example can be encoded with the more primitive *TypeRep*, adapting an example from previous work [VW10] (Section 5.3):

```
data Rid = MkT (∀ a. TypeRep a → a → a)
rt :: TypeRep Rid
rt = typeRep
delta :: ∀ a. TypeRep a → a → a
delta ra x = case (eqT ra rt) of
  Just Refl → case x of MkT y → y rt x
  Nothing   → x
loop = delta rt (MkT delta)
```

These examples demonstrate that primitives like *TypeRep* or *Dynamic* cannot be incorporated in languages where logical consistency is required, such as Coq or Agda, without further restrictions (e.g. predicative polymorphism, or weaker elimination forms). Fortunately Haskell is not one of those.

Parametricity Now that we have added *TypeRep*, *Typeable* and automatically-generated type representations as built-in features, the alert reader might wonder whether we have perhaps accidentally weakened parametricity, and thereby lost Phil's free theorems.

Fortunately, the type system makes it explicit when run-time reflection is used, and must do so *even though every type is Typeable*. Phil's free theorems would go out of the window if we allowed *cast* to have the type $cast :: a \rightarrow Maybe\ b$! That is the principled reason for requiring explicit *Typeable* constraints. There is an operational reason too: the *Typeable* constraint is implemented as a runtime argument; if there was no explicit *Typeable* constraint we would be forced to pass a *Typeable* dictionary to every polymorphic function, just in case it needed it, which would be disgusting.

Thanks to those explicit type representation arguments, techniques that have appeared in previous work [VW10] for a *closed world* of representations can be used to deduce "free theorems" from the types of all polymorphic functions. When those functions include *TypeRep* arguments (or *Typeable* constraints) then such theorems can *still* be derived but are often not very informative.

We conjecture that these results will carry over to (a) an open world of type representations, and (b) representations of types that hide polymorphic types in their defining equations (such as *Rid* above; see Section 5.3 of [VW10]), but both directions are open problems – perhaps new puzzles for Phil to solve!

8 Other applications of *TypeRep* and dynamic

One of the advantages of building *Dynamic* on top of *TypeRep* is that the latter is independently useful to build other abstractions. We briefly describe some of these applications in this section.

8.1 Variants of *Dynamic*

In ML, the extensible *exception* type is built-in, but not so in Haskell: it is programmed as a library using *Typeable*, using a design described by *"An extensible dynamically-typed hierarchy of exceptions"* [MPJMR01]. Here are the key definitions:

```
throw# :: SomeException → a
data SomeException where
  SomeException :: Exception e ⇒ e → SomeException
class (Typeable e, Show e) ⇒ Exception e where {...}
```

The primitive exception-raising operation is *throw#*. Its argument is the fixed type *SomeException*, whose definition is an existential rather like *Dynamic*; the difference is that as well as having a *Typeable* superclass (which makes it like *Dynamic*), *Exception* also has *Show* superclass, and some methods of its own.

The fundamental data structure of the Shake build system [Mit12] is a directed acyclic graph (DAG) in which each node contains a value, a recipe for recomputing that value, and the dependencies of the node. If the values of any of the dependent nodes changes, the node's own value should be recomputed. The value in a node may be of any type, including types defined by the client of the Shake library, so again we have a fundamentally open-world problem. Shake solves this by ubiquitous use of dynamically typed values, both as keys and as values of its finite mappings [Mit12, Section 4.1]. For example, the Shake type *Any* is just like *Dynamic*, except that it includes *Binary*, *Eq*, *Hashable*, *Show*, and *NFData* constraints.

8.2 Supporting generic programming

Here is the opening example from *"Scrap Your Boilerplate"* [LPJ03]:

$$\mathit{increase} :: \mathit{Float} \to \mathit{Company} \to \mathit{Company}$$
$$\mathit{increase}\ k = \mathit{everywhere}\ (\mathit{mkT}\ (\mathit{incS}\ k))$$

A *Company* is a tree-shaped data structure describing a company; the function *increase* is supposed to find every employee in the data structure and increase his or her salary by k, using $\mathit{incS} :: \mathit{Float} \to \mathit{Salary} \to \mathit{Salary}$. The function *everywhere* applies its argument function to every node in the data structure; we will not consider it further here. Our focus is the function *mkT*, which depends critically on *Typeable*:

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
mkT f x = case (cast f) of
  Just g   → g x
  Nothing → x
```

That is, *mkT* takes a type-specific function (such as *incS*) and lifts it to work on values of any type, as follows: if types match use g, otherwise use the identity function. (*cast* was described in Section 5.7.)

So the SYB approach to generic programming depends crucially on dynamic type tests. The popular Uniplate library for generic programming also makes essential use of comparison of *TypeRep*s, for a similar purpose as SYB [MR07]. Note, however, that *TypeRep* alone is not enough to support generic programming (see Section 9.5).

8.3 Distributed programming and persistence

Cloud Haskell is an Erlang-style library for programming a distributed system in Haskell [EBPJ11]. A key component of the implementation, described in the paper, is the ability to serialise a *code pointer* and send it from one node to another in the distributed system. This code pointer could be implemented in a variety of ways, such as: a machine address, a small integer, a long string, a URL.

But regardless of how it is implemented, the receiving node must deserialise the code pointer to some code. To guarantee that the code is then applied to appropriately typed values, the receiving node must perform a dynamic type test. That way, even if the code pointer was corrupted in transit, by accident or malice, the receiving node will be type-sound. A simple way to do this is to serialise a code pointer as a key into a *static pointer table* containing *Dynamic* values. When receiving code pointer with key k, the recipient can lookup up entry k in the table, find a *Dynamic*, and check that it has the expected type. The key can be an integer, a string, or whatever; regardless, if it is corrupted, the recipient might access the wrong entry in the table, but the type test will ensure soundness.

In other variants of this idea, one might want to serialise and deserialise values of type *Dynamic*, and hence of type *TypeRep*. It turns out that this raises some quite interesting issues that are beyond the scope of this paper[14].

8.4 Meta programming

It is perhaps unsurprising that meta programming for a statically typed target language often involves type reflection. For example, it is popular to define a type-indexed version of the syntax tree of expressions, so that a value of type *Expr t* is a syntax tree for an expression of type *t*. But then what is the type of a front end for the language, which takes a *String*, parses it (presumably to an un-typed syntax tree), and then typechecks it to reject type-incorrect programs. The front end cannot have this type

```
frontEnd :: String → Maybe (Expr a)   -- No!
```

because that is too polymorphic. The type *a* has to depend on the contents of the string! So we need something more like this:

```
data DynExp where
  DE :: TypeRep a → Expr a → DynExp
frontEnd :: String → DynExp
```

Here *frontEnd* returns an existential pair of a *Expr a*, and a *TypeRep* that describes the type of the expression. The earliest paper we have found that clearly embodies this idea is *"Tagless staged interpreters for typed languages"* [PTS02], but it has become more widespread since Haskell has supported this programming style [GM08, MCGN15].

9 Related work

9.1 The old implementation of Typeable and Dynamic

GHC has supported *Dynamic* and a non-indexed version of *TypeRep* for some time. Here is the essence of the implementation in GHC 7.10:

```
data TypeRep   -- Abstract
class Typeable a where
  typeRep :: proxy a → TypeRep
data Dynamic where
  Dyn :: TypeRep → a → Dynamic
data Proxy a = Proxy
```

[14] See the tree of wiki pages rooted at `https://ghc.haskell.org/trac/ghc/wiki/DistributedHaskell` for lots more information.

Typeable had built-in support, so that newly declared data types would automatically get *Typeable* instances. But *TypeRep* was not indexed, so there was no connection between the *TypeRep* stored in a *Dynamic* and the corresponding value. Indeed, accessing the *typeRep* required a *proxy* argument to specify the type that should be represented.

Because there is no connection between types and their representations, this implementation of *Dynamic* requires *unsafeCoerce*. For example, here is the old *fromDynamic*:

```
fromDynamic :: ∀ d. Typeable d ⇒ Dynamic → Maybe d
fromDynamic (Dyn trx x)
 | typeRep (Proxy :: Proxy d) == trx = Just (unsafeCoerce x)
 | otherwise                         = Nothing
```

Likewise, *unsafeCoerce* was used in the definition of *dynApply*:

```
dynApply :: Dynamic → Dynamic → Maybe Dynamic
dynApply (Dyn trf f) (Dyn trx x) =
  case splitTyConApp trf of
    (tc, [t1, t2]) | tc == funTc && t1 == trx →
      Just (Dyn t2 ((unsafeCoerce f) x))
    _ → Nothing
```

Here *splitTyConApp*, a special definition from *Data.Typeable* that decomposes type representations, and *funTc* is the function type constructor. There is nothing special about function types. Other forms of data also need their own unsafe elimination functions to be defined. For example, the implementations of *dynFst* and *dynSnd* are similar, and also require *unsafeCoerce*.

Furthermore, the library designer cannot hide all such uses of *unsafeCoerce* from the user. If they want to use their own parameterized type with *Dynamic*, then they too must use *unsafeCoerce*. In short, the old interface to this library is not expressive enough to work with type representations and dynamics safely.

9.2 Dynamics in a closed world

We have focused entirely on an open-world setting (Section 4). If, however, your application can work with a closed world, with a predetermined set of type constructors, then simpler designs are available.

Universal datatype implementation of *Dynamic*. The most obvious way to implement closed-world dynamics is to use a *universal datatype* to represent dynamic values (see Section 4). But even in a closed-world setting this approach is unsatisfactory. Most prominently it suffers from a serious efficiency problem, because converting a value to or from *Dynamic* traverses the value itself. For example, to convert an (*Int*, *Bool*) pair to a *Dynamic*, we would have to deep-copy the value, and then do the reverse when we get it out. This is silly: in the

ST example of Section 3 we only want to store the value in the *Store*, and read it back out later; we shouldn't need to *process* the value in any way whatsoever! Processing the value has a semantic problem too: a dynamic type test forces evaluation of a term, so it will fail on diverging terms; in our ST example, we could not store bottom in the *Store*.

GADT-based type representations. In a closed-world setting, *TypeRep* can be implemented as an ordinary library without built-in support, thus:

```
data TypeRep (a :: ⋆) where
  TBool :: TypeRep Bool
  TFun  :: TypeRep a → TypeRep b → TypeRep (a → b)
  TProd :: TypeRep a → TypeRep b → TypeRep (a, b)
    ...
```

With this representation for *TypeRep*, the functions *eqT*, *dynApply*, *dynFst*, etc., are all easily written, using pattern-matching on the *TypeRep* GADT. The trouble with this approach is simply that it is not extensible: the set of representable types is limited those with data constructors in *TypeRep*.

History of encoding type representations. In concurrent work, Cheney and Hinze [CH02] and Baars and Swierstra [BS02] showed how to implement type *Dynamic* by first encoding indexed type representations, similar to the GADT shown above. (GADTs were not a part of GHC at that time). These type representation encodings were based on earlier work by Yang [Yan98] and Weirich [Wei04]. In particular, Yang showed how to encode the *TypeRep* type above using higher-order polymorphism (available in the ML module system). And Weirich used type classes to encode a version of type Dynamic that supported *type-safe* cast (i.e. *toDynamic* and *fromDynamic*) but could not destruct types (i.e. no *dynApply*).

9.3 Dynamic typing in other statically-typed languages

Several statically typed languages include a *Dynamic* type as a language primitive. Abadi et al. [ACPP91, ACPR95] laid the groundwork for such extensions, by incorporating a special type *Dynamic* to contain values of unknown type. Values of this type could be refined using a *typecase* operator that allowed pattern matching on the actual type of the value.

Clean. The Clean language includes a dynamic type as well as a class constraint similar to *Typeable*, called TC, for types that support "type codes" [Pil99]. Unlike Haskell, where *Dynamic* is defined in terms of *Typeable*, *both* of these structures are language primitives in Clean. Pil [Pil99] makes the case that type *Dynamic* is not enough on its own; languages should also include something like *Typeable*. Making *Dynamic* a language primitive is powerful. For example, Clean can support polymorphic values embedded into dynamics. In other words, types such as $\forall\, a.\, a \to a$ are *Typeable*, unlike in Haskell (Section 5.8).

Clean's interface to runtime types is also different from a user perspective. Clean includes runtime type unification using type-pattern variables. This features subsumes both runtime type equality (as in our eqT) as type patterns may be nonlinear, and type destruction (as in our $splitApp$). We conjecture that this difference is cosmetic.

OCaml. Leroy and Mauny [LM91] used similar mechanism to Abadi et al. in order to extend the ML language with a dynamic type. However, more recent extensions in support of dynamic typing in OCaml have been proposed in workshop talks [Fri11, HG13]. Like the design presented here, these extensions include a type for type representations `'a ty`, and a comparison operation that returns a GADT-based witness for type equality.

These extensions also include a mechanism to decompose type representations. In this case, every type constructor needs its own a decomposition GADT and function. For example (using Haskell syntax), the pair type constructor would be accompanied by the following:

```
data IsPair a where
  TypePair :: TypeRep a → TypeRep b → IsPair (a, b)
isPair :: TypeRep a → Maybe (IsPair a)   -- Primitive implementation
```

These definitions could then be used to implement $dynFst$.

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn tab x) = case isPair tab of
  Just (TypePair ta tb) → Just (fst x)
  Nothing → Nothing
```

Our $AppResult$ GADT and $splitApp$ function uses GHC's kind-polymorphism to generalize these definitions for any type constructor.

9.4 Reflection in Java

In a language with subtyping and down-casting, a maximal supertype acts like a dynamic type. For example, in Java, a reference type, like $String$, can be coerced to $Object$ (the maximal supertype) without runtime overhead. Furthermore, Java also supports a simple equivalent of $fromDynamic$ using a runtime cast.

Here is yet another connection to Phil, who introduced generics to Java [BOSW98]. Java's generics are limited in two (related) ways: Java has no notion of higher-kinded types, and type parameters to generic clases (such as $List\langle T\rangle$) and methods are erased, disallowing dynamic checks involving those parameters.

Generics have enhanced Java's reflection feature, which has remarkable parallels to Haskell's. From its first versions, Java had the $Class$ type, which was useful for queries about a type, but not for casting. Phil's generics then allowed $Class$ to be type-indexed [NW06], just like we are doing with $TypeRep$. For any reference type T in modern Java, the expression $T.\mathbf{class}$ is a runtime type representation, of type $Class\langle T\rangle$. With a $Class\langle T\rangle$ in hand, we can cast an arbitrary value to T, even if T is a type parameter.

9.5 Generic programming and type representations

The type representations described in this paper are not designed for full-blown *datatype generic* programming [Gib07]. *TypeRep* allows one to compare and explore just the shape of a type including names of type constructors and type arguments. But generic programming requires the additional capability to generically explore the structure of the *values* inhabiting a type; for instance by allowing one to iterate over the data constructors of some unknown type without having the actual type definition in scope.

There are many ways to support generic programming; from isomorphisms of types to sums and products (for example the *Generic* class in Haskell [MDJL10]), to providing built-in generic instances for iterators (for example the *Data* class in Haskell [LPJ03]), to advanced variants of *TypeRep* that additionally include type-indexed data structures for describing data constructors and introducing/eliminating values generically [Wei06].

10 Conclusions and further work

In this paper, we have designed a powerful API for type reflection and shown that it can be used to implement a flexible and extensible dynamic type. The next major challenge is to provide a better story for polymorphic dynamic values (Section 5.8).

Acknowledgements

Thanks to Neil Mitchell both for help in understanding Shake and for a rapid review. Kenneth Foner also provided feedback on a draft. Ben Price's internship at Microsoft Research was helpful in bringing some of the key issues to the surface. This paper was typeset with *lhs2TeX*[15].

Finally, thanks to Phil, who has spent much of his professional life reflecting on types. Increasingly, Haskell can too. Happy birthday Phil!

References

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 183–200. ACM, 1998.

[15] `http://www.andres-loeh.de/lhs2tex/`

[BS02] Arthur I. Baars and Doaitse Swierstra. Typing dynamic typing. In *International Conference on Functional Programming*, pages 157–166. ACM, 2002.

[CH02] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Workshop on Haskell*, pages 90–104. ACM, 2002.

[EBPJ11] Jeff Epstein, Andrew P. Black, and Simon Peyton Jones. Towards Haskell in the cloud. In *Haskell Symposium*. ACM, 2011.

[EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell Symposium*. ACM, 2012.

[Fri11] Alain Frisch. Runtime types in OCaml. Presentation at Meeting of the Caml Consortium, November 2011.

[Gib07] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007.

[GM08] Louis-Julien Guillemettte and Stefan Monnier. A type-preserving compiler in Haskell. In *International Conference on Functional Programming*. ACM, 2008.

[HG13] Grégoire Henry and Jacques Garrique. Dynamic typing in OCaml. Presentation at Nagoya University, 2013.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Conference on History of Programming Languages*, 2007.

[KS04] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: Implicit configurations–or, type classes reflect the values of types. In *Workshop on Haskell*, pages 33–44. ACM, 2004.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523, pages 406–426. Springer-Verlag, 1991.

[LPJ95] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp Symb. Comput.*, 8(4), December 1995.

[LPJ03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Workshop on Types in Languages Design and Implementation*. ACM, 2003.

[LPJ05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *International Conference on Functional Programming*. ACM, 2005.

[McB02] Conor McBride. Elimination with a motive. In *Workshop on Types for Proofs and Programs*, TYPES '00, pages 197–216. Springer-Verlag, 2002.

[MCGN15] Trevor McDonell, Manuel Chakravarty, Vinod Grover, and Ryan Newton. Type-safe runtime code generation. In *Haskell Symposium*, pages 201–212. ACM, 2015.

[MDJL10] Jose Pedro Magalhaes, Atze Dijkstra, Johan Jeuring, and Andres Loeh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48. ACM, 2010.

[Mit12] Neil Mitchell. Shake before building: Replacing Make with Haskell. In *International Conference on Functional Programming*. ACM, 2012.

[MPJMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Programming Language Design and Implementation*. ACM, 2001.

[MR07] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Workshop on Haskell*. ACM, 2007.

[NW06] Maurice Naftalin and Phil Wadler. *Java Generics and Collections*. O'Reilly Media, 2006.

[Pil99] Marco Pil. Dynamic types and type dependent functions. In K. Hammand, T. Davie, and C. Clack, editors, *Workshop on the Implementation of Functional Languages*, LNCS, pages 169–185. Springer Verlag, 1999.

[PJVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, pages 50–61. ACM, 2006.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*. ACM, 1993.

[PTS02] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming*. ACM, 2002.

[SCPJD07] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Workshop on Types in Languages Design and Implementation*. ACM, 2007.

[VW10] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming*, 20:175–210, 2010.

[Wad89] Philip Wadler. Theorems for free! In *International Conference on Functional Programming Languages and Computer Architecture*. ACM, 1989.

[Wad90] Philip Wadler. Comprehending monads. In *Conference on LISP and Functional Programming*. ACM, 1990.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Principles of Programming Languages*, pages 60–76. ACM, 1989.

[Wei04] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.

[Wei06] Stephanie Weirich. Replib: A library for derivable type classes. In *Workshop on Haskell*. ACM, 2006.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, pages 275–286. ACM, 2013.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Principles of Programming Languages*. ACM, 2003.

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *International Conference on Functional Programming*, pages 289–300. ACM, 1998.

[YWC+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhaẽs. Giving Haskell a promotion. In *Workshop on Types in Language Design and Implementation*. ACM, 2012.