*JFP* **14** (6): 681–695, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005179 Printed in the United Kingdom


# FUNCTIONAL PEARL

## *Type-safe cast**

STEPHANIE WEIRICH
*University of Pennsylvania, Philadelphia, PA 19104, USA*
(*e-mail:* sweirich@cis.upenn.edu)

**Abstract**

Comparing two types for equality is an essential ingredient for an implementation of dynamic types. Once equality has been established, it is safe to cast a value from one type to another. In a language with run-time type analysis, implementing such a procedure is fairly straightforward. Unfortunately, this naive implementation destructs and rebuilds the argument while iterating over its type structure. However, by using higher-order polymorphism, a casting function can treat its argument parametrically. We demonstrate this solution in two frameworks for ad-hoc polymorphism: intensional type analysis and Haskell type classes.

### 1 Heterogeneous symbol tables and dynamic types

Dynamic types are a useful addition to a statically-typed language. For example, they may be used to implement a heterogeneous symbol table — a finite map from strings to values of many different, unrelated types. The interface to this data structure includes, at minimum, an abstract type, a value for the empty table, and two functions to insert items into and retrieve items from the table. However, because this table must store many different types of data, it is tricky to specify and implement in a statically-typed language.

For example, we might want this data structure to have the following interface in the functional programming language Haskell (Peyton Jones, 2003):

```
type Table
empty  :: Table
insert :: String → α → Table → Table
find   :: Table → String → Maybe α
```

In this interface, both `insert` and `find` are polymorphic over the types of values that may be added to and retrieved from the table. However, this interface is not quite right. Because the return type of `find` is polymorphic, `find` must return the correct type for every context. But as nothing is assumed about the context, there is no way to verify that `find` returns the correct type.

* A previous version of this paper appeared in the *Fifth ACM International Conference on Functional Programming (ICFP 00)*.

To make matters concrete, suppose we use an association list, a list of pairs of symbols and their values, to store the symbol table.

```
type Table = [(String,Entry)]
```

The type `Entry` must be a dynamic type. We must define it in such a way that it may contain many types of data. Furthermore, we must also modify the interface so that `find` may verify that the type of a retrieved value is the same as the type expected by the context.

One way to define `Entry` is to use a variant to allow a number of different types:

```
data Entry = Int Int | Bool Bool | Char Char | FunIntInt (Int → Int)
           | TupleIntInt (Int,Int) | TupleCharInt (Char,Int)
```

If `find` returns an `Entry`, the context can use pattern matching to ensure that the right type is returned. However, this variant constrains the entries in the table to a specific finite set of types. Although a single run of the program will use only a finite number of types, it may be difficult to say in advance what those types might be.

Another definition of `Entry` is to use an *existential type* (Mitchell & Plotkin, 1988; Odersky & Läufer, 1996). A value of type $\exists\alpha.\alpha$ may be of any type. Though not a part of the Haskell 98 definition, several implementations of Haskell support existential types. We may declare that the data type constructor `Entry` carries a value of type $\exists\alpha.\alpha$ as follows:

```
data Entry = forall α. Entry α
```

Because this constructor may be applied to values of any type, the keyword `forall` indicates the existential type.

The insert function coerces the value into an existential and adds it to the rest of the table.

```
insert :: String → α → Table → Table
insert symbol val table = (symbol, Entry val) : table
```

For example, we can create a symbol table with the expression:

```
table1 :: Table
table1 = insert "x" 1 (insert "y" True empty)
```

The existential type provides a solution for the first problem. We can create the table and add values of any type to it. However, the second problem remains. We cannot retrieve values from the symbol table. Once a type has been hidden by an existential, all information about that type has been lost. When a value is unpacked from the existential type, the Haskell type checker must assume that it is different from every other type. As a result, if we naively try to implement `find` by iterating over the list, the Haskell type checker reports the type error of returning a type containing an existentially quantified variable.

```
find :: String → Table → Maybe α
find symbol1 [] = Nothing
find symbol1 ((symbol2, Entry val) : table) =
  if symbol1 == symbol2
  then Just val              -- DOES NOT TYPE CHECK
  else find symbol1 table
```

In `find`, we must verify at run time that the type of `val` is the same as the expected result type of the function. We need to define a function `cast` that will safely convert `val` to the correct type. This operation must depend on run-time type information, so we need some sort of non-parametric polymorphism, such as Haskell's type class mechanism (Wadler & Blott, 1989). By supplying additional class constraints to the types of `insert` and `find` as well as the definition of `Entry`, we can implement this heterogeneous symbol table in Haskell.

However, type classes do not permit the most natural definition of the casting operation which must compare two run-time types for equality. Therefore, before describing how to implement `cast` with Haskell type classes, we first define it with run-time type analysis using the `typecase` operator in the language $\lambda_i^{ML}$ (Harper & Morrisett, 1995). Furthermore, that straightforward definition allows us to discover an optimization of `cast`.

The next section presents an introduction to the $\lambda_i^{ML}$ language. An initial implementation of the `cast` function in $\lambda_i^{ML}$ appears in section 3. Though correct, its operation requires an undesirable overhead for what is essentially an identity function. In section 4, we improve `cast` through the aid of an additional type constructor argument. In section 5, we develop the analogous two implementations of `cast` in Haskell using type classes. Finally, in section 6, we conclude by comparing both versions with several other implementations of dynamic types.

## 2 Intensional type analysis

Harper and Morrisett's language $\lambda_i^{ML}$ augments a small, polymorphic functional language with a `typecase` operator that can be used to examine type information that is known only at run time. The $\lambda_i^{ML}$ language is defined by a *type-passing* semantics. In other words, polymorphic functions receive run-time representations of their type arguments. Dually, type information is stored with the value of an existential type.

The `typecase` operator pattern matches run-time type information. To demonstrate a simple use of `typecase` and foreshadow the definition of `cast`, we implement a function, called `sametype`, that compares two types for equality. In this paper, we use a syntax for $\lambda_i^{ML}$ that is similar to Haskell to ease the comparisons between the different versions of `cast`. The major difference between $\lambda_i^{ML}$ and Haskell (besides `typecase`) is that in $\lambda_i^{ML}$ all type abstraction and instantiations are explicitly notated. For example, `sametype` below, has two type arguments ($\alpha$ and $\beta$) that are enclosed in square brackets.

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
sametype :: ∀α.∀β. bool
sametype[α][β]  =
  typecase (α) of
    (Int)      ⇒ typecase (β) of
                   (Int) ⇒ true
                   (_)   ⇒ false
    (α1,α2)    ⇒ typecase (β) of
                   (β1,β2) ⇒ sametype[α1][β1] && sametype[α2][β2]
                   (_)     ⇒ false
    (α1 → α2) ⇒ typecase (β) of
                   (β1 → β2) ⇒ sametype[α1][β1] && sametype[α2][β2]
                   (_)       ⇒ false
```

The first `typecase` discovers the outermost form of the first type, whether it is `Int`, a product type $(\alpha_1, \alpha_2)$ or a function type $(\alpha_1 \to \alpha_2)$. Then in each branch, an inner `typecase` compares this form to the form of the second type. For product and function types, `sametype` calls itself recursively on the subcomponents of the type. Each of these branches binds type variables (such as $\alpha_1$ and $\alpha_2$) to the subcomponents of the types so that they may be used in the recursive call.

Because nested `typecase`s are tedious to write, we use pattern matching syntax to abbreviate this function as:

```
sametype :: ∀α.∀β. bool
sametype[α][β] =
    typecase (α,β) of
        (Int,Int)           ⇒  true
        ((α1,α2),(β1,β2))   ⇒  sametype[α1][β1] && sametype[α2][β2]
        (α1 → α2,β1 → β2)   ⇒  sametype[α1][β1] && sametype[α2][β2]
        (_,_)               ⇒  false
```

This function is the core of the `cast` function, because `cast` also compares two types for equality. However, if the types match, `cast` must also change the type of a term from the first type to the second. To do so requires an important property about type checking `typecase`. In a standard case expression each branch must be of the same type. In a `typecase` expression, the type of each branch may depend on the matched type (or types).

For example, consider the following expression.

```
 typecase τ of
   (Int)    ⇒  λ(x :: Int) → x + 3
   (α → β)  ⇒  λ(x :: α → β) → x
   (α,β)    ⇒  λ(x :: (α,β)) → x
```

Even though the first branch is of type `Int` → `Int`, the second branch is of type $(\alpha \to \beta) \to (\alpha \to \beta)$, and the third branch is of type $(\alpha, \beta) \to (\alpha, \beta)$, all three branches are instances of the type schema $\gamma \to \gamma$, where $\gamma$ is replaced with the match for $\tau$ for that branch. Therefore, this entire `typecase` expression can be safely assigned the type $\tau \to \tau$.

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
cast :: ∀α. ∀β. Maybe (α → β)
cast[α][β] =
    typecase [δ1, δ2. δ1 → δ2] (α,β) of
       (Int, Int) ⇒
          Just (λ(x :: Int) → x)
       ((α1,α2), (β1,β2)) ⇒
          do f ← cast [α1][β1]
             g ← cast [α2][β2]
             return (λ(x :: (α1,α2)) → (f (fst x), g (snd x)))
       (α1 → α2, β1 → β2) ⇒
          do f ← cast [β1][α1]
             g ← cast [α2][β2]
             return (λ(x :: α1 → α2) → g . x . f)
       (_,_) ⇒ Nothing
```

Fig. 1. First solution.

To make type checking syntax directed, we annotate `typecase` with a type variable and a type schema where that variable may occur free. For example, we annotate the previous `typecase` with the following schema:

```
typecase [γ.γ → γ] τ of
      (Int) ⇒ λ(x :: Int) → x + 3
```

In `cast`, which pattern matches two types, the schema has two free type variables. We now have everything that we need to write `cast` in $\lambda_i^{ML}$.

## 3 First solution

An initial $\lambda_i^{ML}$ implementation of `cast` appears in Figure 1. This operation compares the types $\alpha$ and $\beta$. If they match, `cast` returns a conversion function from type $\alpha$ to type $\beta$. Otherwise, `cast` returns the data constructor `Nothing`, signaling the error.

In the first branch of `typecase`, $\alpha$ and $\beta$ both match `Int`. Casting an `Int` to an `Int` is easy; this branch just returns an identity function.

In the second branch of `typecase`, both $\alpha$ and $\beta$ match product types, $(\alpha_1, \alpha_2)$ and $(\beta_1, \beta_2)$, respectively. Through recursion, we create functions to cast the subcomponents, $\alpha_1$ to $\beta_1$ and $\alpha_2$ to $\beta_2$. The coercion of a product breaks it apart, casts each component separately, and then creates a new pair.

This branch uses Haskell's `do` notation for error propagation. Each of the two recursive calls to cast could produce either a conversion function or `Nothing`. If they produce functions, everything continues smoothly, but if either returns `Nothing`, then the entire `do` expression returns `Nothing`.

The code is a little different for the next branch, when $\alpha$ and $\beta$ are both function types. Here, given a function from $\alpha_1$ to $\alpha_2$, we want to return a function from $\beta_1$ to $\beta_2$. We can apply cast to $\alpha_2$ and $\beta_2$ to get a function, g, that casts the result type,

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

and compose g with the argument x to get a function from $\alpha_1$ to $\beta_2$. Then we can compose that resulting function with a reverse cast from $\beta_1$ to $\alpha_1$ to get a function from $\beta_1$ to $\beta_2$.

However, there is a problem with this solution. The specification of the cast function is to recursively compare two types and produce an identity function when the types are the same. This solution follows that specification, which we can prove by induction. The product and arrow cases map the recursive calls across the argument x, and mapping the identity function is the same as the identity function. However, the result of cast is a particularly inefficient identity function. Unless the types $\tau_1$ and $\tau_2$ are both `Int`, `cast`$[\tau_1][\tau_2]$ does much more work than $\lambda$`x` $\rightarrow$ `x`. Every pair in the argument is broken apart and remade and every function is wrapped between two instantiations of cast.

The reason we had to break apart the pair in forming the coercion function for product types is that we may only coerce the components of the pair individually. It would be better if we could coerce each component while part of the pair. In other words, we want two functions, first one that casts the first component of the pair, of type $(\alpha_1, \alpha_2) \rightarrow (\beta_1, \alpha_2)$, and then one that casts the second component, of type $(\beta_1, \alpha_2) \rightarrow (\beta_1, \beta_2)$.

## 4 Second solution

Motivated by this reasoning, we want to write a function that can coerce the type of *part* of its argument. This will allow us to produce a coercion that is a composition of several identity functions, each changing only part of the type of its argument x. Because we want to cast many parts of the type of x, we need to abstract the relationship between the part of the type to analyze and the complete type of x.

The solution in Figure 2 defines a helper function `cast'` that abstracts not only the types $\alpha$ and $\beta$ for analysis, but also a *type constructor* $\gamma$ (or function from types to types). In the definition of `cast'` we annotate $\alpha$, $\beta$, and $\gamma$ with their kinds — $\alpha$ and $\beta$ with kind $*$, and $\gamma$, a function from types to types, with kind $* \rightarrow *$.

When $\gamma$ is applied to the type $\alpha$ we get the argument type of `cast'`; when it is applied to $\beta$ we get the return type of `cast'`. For example, to produce a function that casts the first component of a tuple but leaves the second component alone, we instantiate $\gamma$ with $\lambda\delta :: *.(\delta, \alpha_2)$. The result is then a function from type $(\alpha, \alpha_2)$ to $(\beta, \alpha_2)$. The `cast'` operation is more generally useful than `cast` because it may convert types within data structures. We can specialize `cast'` to produce `cast` by calling it with the identity function on types.

`cast`$[\alpha][\beta]$ = `cast'`$[\alpha][\beta][\lambda\delta :: *.\delta]$

The implementation of `cast'` again uses pattern matching to determine the top level form of the arguments $\alpha$ and $\beta$.

In the branch for product types we create a function to coerce the first component of the tuple (converting $\alpha_1$ to $\beta_1$) by supplying the type constructor $\lambda\delta :: *.\gamma(\delta, \alpha_2)$ for $\gamma$. In the recursive call for the second component, the first component is already

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
cast' :: ∀α :: *. ∀β :: *.∀γ :: * → *. Maybe (γ α)→(γ β)
cast'[α][β][γ] =
    typecase [δ₁, δ₂. (γ δ₁)→(γ δ₂)](α,β) of
       (Int, Int) ⇒
          Just ( λ(x :: γ Int) → x )
       ((α₁,α₂),(β₁,β₂)) ⇒
          do f ← cast'[α₁][β₁][λδ :: *. γ(δ,α₂)]
             g ← cast'[α₂][β₂][λδ :: *. γ(β₁,δ)]
             return (λ(x :: γ(α₁,α₂)) → g (f x))
       (α₁ → α₂,β₁ → β₂) ⇒
          do f ← cast'[α₁][β₁][λδ :: *. γ(δ → α₂)]
             g ← cast'[α₂][β₂][λδ :: *. γ(β₁ → δ)]
             return (λ(x :: γ(α₁ → α₂)) → g (f x))
       (_,_) ⇒ Nothing

cast :: ∀α :: *. ∀β :: *. Maybe (α → β)
cast[α][β] = cast'[α][β][λδ :: *.δ]
```

Fig. 2. Second solution.

of type $\beta_2$ so the type constructor argument reflects that fact. The returned function does not need to destruct its argument; it only applies the two conversions.

Surprisingly, the branch for comparing function types is similar to the branch for product types. We coerce the argument type of the function in the same manner as we coerced the first component of the tuple: calling `cast'` recursively to produce a function to cast from type $\gamma(\alpha_1 \to \alpha_2)$ to type $\gamma(\beta_1 \to \alpha_2)$. A second recursive call handles the result type of the function.

This version of cast is more efficient because it does not destruct and rebuild its argument. However, we have added an additional type constructor argument and because $\lambda_i^{ML}$ has a type-passing semantics, it must pass this additional argument at run time. Creating this argument could slow down the execution of `cast'`. Fortunately, `typecase` never examines that argument, so an optimizer is free to eliminate it in an implementation of $\lambda_i^{ML}$.

## 5 Haskell

The Haskell language provides a form of ad-hoc polymorphism, called type classes, that differs from `typecase`. In this section, we quickly review Haskell type classes before describing how to implement `cast` with them.

Instead of defining an ad-hoc polymorphic operation through case analysis of some type argument, with type classes one defines such operations by listing the instances for each type separately. For example, the Haskell Prelude (Peyton Jones, 2003) defines the class of types that support a conversion to a string representation.

```
class Show α where
  show :: α → String
```

This declaration states that a type $\alpha$ is in the class `Show` if there is some function named `show` defined with type $\alpha \rightarrow$ `String`. We can define `show` for integers with a built-in primitive

```
instance Show Int where
  show x = primIntToString x
```

We can also define `show` for compound types like product types. To convert a product to a string we need a version of `show` for each component of the product.

```
instance (Show α, Show β) ⇒ Show (α,β) where
  show (a,b) = "(" ++ show a ++ "," ++ show b ++ ")".
```

This code declares that as long as $\alpha$ and $\beta$ are members of the class `Show`, then their product is a member of class `Show`. Consequently, `show` for products is defined in terms of the `show` functions for its subcomponents.

### *5.1 First solution in Haskell*

The Haskell version of `cast` is complicated by the two nested `typecase` terms hidden by the pattern-matching syntax. For this reason, we define two type classes — one called `CF` (for cast from) that corresponds to the outer `typecase`, and the other called `CT` (for cast to) that corresponds to all of the inner `typecase`s. (We may also combine these two classes into a single class, if desired.)

The `CF` class contains `cast`. To cast from type $\alpha$ to type $\beta$, $\alpha$ must be in the `CF` type class and $\beta$ must be in the `CT` type class.

```
class CF α where
  cast :: CT β ⇒ Maybe (α → β)
```

The `CT` class includes three functions, each completing the cast assuming that the first type was an integer, product, or function. This class also defines *default* values for the three functions, for the case when the types do not match. Each instance of `CT` overrides one of these functions.

```
class CT β where
  doInt  :: Maybe (Int → β)
  doProd :: (CF α1, CF α2) ⇒ Maybe ((α1, α2) → β)
  doFn   :: (CT α1, CF α2) ⇒ Maybe ((α1 → α2) → β)
  doInt  = Nothing
  doProd = Nothing
  doFn   = Nothing
```

The type of `doProd` requires that $\alpha_1$ and $\alpha_2$ be in the type class `CF` because `doProd` calls `cast` to convert from these types. Likewise, `doFn` calls `cast` to convert from the type $\alpha_2$. However, it converts *to* the type $\alpha_1$, so $\alpha_1$ must be in the `CT` type class.

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

As in $\lambda_i^{ML}$, where the outer typecase led to an inner typecase in each branch, each instance of CF dispatches to one of the functions of CT to record the form of the first type. Haskell syntax does not allow type declarations for methods in class instances (their types are inferred from the class declaration), but for clarity we include those types in comments before each method.

```
instance CF Int where
 -- cast :: CT β ⇒ Maybe (Int → β)
    cast = doInt
instance (CF α1, CF α2) ⇒ CF (α1, α2) where
 -- cast :: CT β ⇒ Maybe ((α1, α2) → β)
    cast = doProd
instance (CT α1, CF α2) ⇒ CF (α1 → α2) where
 -- cast :: CT β ⇒ Maybe ((α1 → α2) → β)
    cast = doFn
```

In the Int instance of CT, the doInt function is an identity function.

```
instance CT Int where
 -- doInt :: Maybe (Int → Int)
    doInt = Just id
```

The product instance of CT overrides doProd. This function calls cast for the subcomponents of the product. For these calls, $\alpha_1$ and $\alpha_2$ must be in the CF type class and $\beta_1$ and $\beta_2$ must be in the CT type class.

```
instance (CT β1, CT β2) ⇒ CT (β1, β2) where
 -- doProd :: (CF α1, CF α2) ⇒ Maybe ((α1, α2) → (β1, β2))
    doProd = do f ← cast      -- from α1 to β1
                g ← cast      -- from α2 to β2
                return ( λx → (f (fst x), g (snd x)) )
```

Finally, in the instance for the function type constructor, doFn needs to wrap the argument x (of the returned conversion function) in calls to cast, as in the first $\lambda_i^{ML}$ solution. The type of x is a function of type $\alpha_1 \rightarrow \alpha_2$. To cast the result of this function, we require that $\alpha_2$ be in the type class CF and $\beta_2$ be in the class CT. We also need to cast the argument of the function in the opposite direction, from $\beta_1$ to $\alpha_1$. Therefore we need $\beta_1$ to be in the class CF, and $\alpha_1$ to be in the class CT.

```
instance (CF β1, CT β2) ⇒ CT (β1 → β2) where
 -- doFn :: (CT α1, CF α2) ⇒ Maybe ((α1 → α2) → (β1 → β2))
    doFn = do f ← cast      -- from α1 to β1
              g ← cast      -- from β2 to α2
              return (λx → g . x . f)
```

With these definitions we can define the symbol table using an existential type to hide the type of the elements in the table. However, this time the existential requires that the hidden type is in the CF type class. Because of that restriction, the insert function also constrains the type of values added to the table.

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
data Entry = forall α. CF α ⇒  Entry α
type table = [ (String, Entry) ]
insert :: CF α ⇒ (String, α) → Table →  Table
insert symbol val table = (symbol, Entry val) : table
```

The find function is very similar to before. However, the type of the value to be found must be in the CT type class so that we may cast it to type $\alpha$.

```
find ::  CT α ⇒ Table → String → Maybe α
find symbol1 [] = Nothing
find symbol1 ((symbol2, Entry val) : table) =
       if symbol1 == symbol2
       then do f ← cast
               return (f val)
       else find symbol1 table
```

### 5.2 Second solution in Haskell

To implement the more efficient version in Haskell, we need to replace cast with cast' in the CF type class. Again cast' abstracts the type constructor $\gamma$ as well as $\alpha$ and $\beta$. This change leads to the following new definitions of CT and CF. This time the type of doFn requires that $\alpha_1$ be in the class CF instead of the class CT, reflecting that we avoid the contravariant cast of the function argument of the previous solution.

```
class CF α where
   cast' :: CT β ⇒ Maybe (γ α → γ β)
class CT β where
   doInt  :: Maybe (γ Int → γ β)
   doProd :: (CF α1, CF α2) ⇒ Maybe (γ (α1, α2) → γ β)
   doFn   :: (CF α1, CF α2) ⇒ Maybe (γ (α1→ α2) → γ β)
   doInt  = Nothing
   doProd = Nothing
   doFn   = Nothing
```

The instances for CF are the same as in the first version, dispatching to the appropriate methods of CT. The instance CT Int is also unchanged.

```
instance CF Int where
 -- cast' :: CT β ⇒ Maybe (γ Int → γ β)
    cast' = doInt
instance (CF α1, CF α2) ⇒ CF (α1, α2) where
 -- cast' :: CT β ⇒ Maybe (γ (α1, α2) → γ β)
    cast' = doProd
instance (CT α1, CF α2) ⇒ CF (α1 → α2) where
 -- cast' :: CT β ⇒ Maybe (γ (α1 → α2) → γ β)
    cast' = doFn
instance CT Int where
 -- doInt :: Maybe (γ Int → γ Int)
    doInt = Just id
```

However, the instances of CT are more complicated. Recall the branch for products in the $\lambda_i^{ML}$ version:

```
((α₁,α₂),(β₁,β₂)) ⇒
    do f ← cast'[α₁][β₁][λδ :: *. γ(δ,α₂)]
       g ← cast'[α₂][β₂][λδ :: *. γ(β₁,δ)]
       return (λ(x :: γ(α₁,α₂)) → g (f x))
```

The core idea is to cast the left side of the product first and then to cast the right side, using the type-constructor argument to relate the type of the term argument to the types being examined. In Haskell, we cannot explicitly instantiate the type constructor argument as in $\lambda_i^{ML}$—it must be inferred. However, to match $\gamma(\alpha_1, \alpha_2)$ with the type constructor $\lambda\delta :: *.\gamma(\delta, \alpha_2)$ applied to the type $\alpha_1$ requires higher-order unification, which is undecidable. As a tractable alternative, the Haskell language requires that instantiated type constructors be constants applied to some number of arguments (Jones, 1995). Using the newtypes LP and RP defined below, we create new constants that can be used in a call to cast'.

```
newtype LP γ β α = LP (γ (α, β))
newtype RP γ α β = RP (γ (α, β))
```

The last argument of each newtype is the type that should be changed by cast' in the definition of doProd.

```
instance (CT β₁, CT β₂) ⇒ CT (β₁, β₂) where
-- doProd :: (CF α₁, CF α₂) ⇒ Maybe (γ (α₁,α₂) → γ (β₁,β₂))
   doProd = do f ← cast'     -- from (LP γ α₂) α₁ to (LP γ α₂) β₁
               g ← cast'     -- from (RP γ β₁) α₂ to (RP γ β₁) β₂
               return (λx → let LP y = f (LP x)
                                RP w = g (RP y)
                            in w)
```

How does this code type check? The type of x is $\gamma$ ($\alpha_1$, $\alpha_2$) so LP x is of type LP $\gamma$ $\alpha_2$ $\alpha_1$. The call f (LP z), uses the $\alpha_2$ instance of cast' of type (CT $\beta$) ⇒ Maybe ($\gamma'$ $\alpha_1$→ $\gamma'$ $\beta$). Determining $\gamma'$ is a simple match — it is the partial application (LP $\gamma$ $\alpha_2$). Thus, the result of the first cast is of type (LP $\gamma$ $\alpha_2$) $\beta$, for some $\beta$ in CT, and y is of type $\gamma$ ($\beta$, $\alpha_2$).

Therefore, RP y is of type RP $\gamma$ $\beta$ $\alpha_2$, so we need the $\alpha_2$ instance of cast' for the second call. This instance is of type CT $\beta'$ ⇒ Maybe ($\gamma''$ $\alpha_2$ → $\gamma''$ $\beta'$). This $\gamma''$ unifies with the partial application (RP $\gamma$ $\beta$) so the return type of this cast is RP $\gamma$ $\beta$ $\beta'$, the type of RP w. That makes w of type $\gamma$ ($\beta$, $\beta'$). Comparing this type to the return type of doProd, we unify $\beta$ with $\beta_1$ and $\beta'$ with $\beta_2$. This unification satisfies our constraints for the two calls to cast' as we assumed that both $\beta_1$ and $\beta_2$ are in the class CT.

Like the second $\lambda_i^{ML}$ solution, function types work in exactly the same way as product types, using similar declarations of LA and RA.

```
newtype LA γ β α = LA (γ (α → β))
newtype RA γ α β = RA (γ (α → β))
```

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
instance (CT β1, CT β2) ⇒ CT (β1 → β2) where
-- doFn :: (CF α1, CF α2) ⇒ Maybe (γ (α1 → α2) → γ (β1 → β2))
   doFn = do f ← cast'  -- from LA γ α2 α1 to LA γ α2 β1
             g ← cast'  -- from RA γ β1 α2 to RA γ β1 β2
             return (λx → let LA y = f (LA x)
                              RA w = g (RA y)
                          in w)
```

Finally, cast can be defined in terms of cast':

```
newtype I α = I α
cast :: (CF α, CT β) ⇒ Maybe (α → β)
cast = do f ← cast'
          return (λ x → let (I y) = f (I x) in y)
```

## 6 Implementing a dynamic type

Just as $\exists\alpha.\alpha$ implements a dynamic type in $\lambda_i^{ML}$, `forall α. CF α ⇒ α` is a dynamic type in Haskell. A limitation of this dynamic type is that it does not support pattern matching of the hidden type. The only projection from the dynamic type is `cast`. If we do not know the complete type of the value, there is no way to discover it other than a brute force search. More recent implementations of dynamics in Haskell (Baars & Swierstra, 2002; Cheney & Hinze, 2002) provide the ability to determine the form of the hidden type using a similar encoding of type equality as found in this paper. More interestingly, they point out that the type of the second version of `cast`, $\forall\gamma.\gamma\alpha \rightarrow \gamma\beta$, corresponds to Leibnitz equality. The only total member of this type is the identity function, providing a correctness argument for this version.

Adding a dynamic type to a statically typed language is not new, so it is interesting to compare this implementation with other versions of dynamic types. One previous implementation is to use a universal datatype.

```
data Dynamic = Base Int
             | Pair (Dynamic, Dynamic)
             | Fn   (Dynamic → Dynamic)
```

Here, in creating a value of type `Dynamic`, a term is tagged with the head constructor of its type. However, before a term may be injected into this type, if it is a pair, its subcomponents must be coerced, and if it is a function, it must be converted to a function from `Dynamic → Dynamic`. We could implement this injection and its associated projection with Haskell type classes as follows:

```
class UD α where
  toD   :: α → Dynamic
  fromD :: Dynamic → α
```

https://doi.org/10.1017/S0956796804005179 Published online by Cambridge University Press

```
instance UD Int where
  toD x = Base x
  fromD (Base x) = x
instance (UD α, UD β) ⇒ UD (α,β) where
  toD (x1,x2) = Pair (toD x1, toD x2)
  fromD (Pair (d1, d2)) = (fromD d1, fromD d2)
instance (UD α, UD β) ⇒ UD (α → β) where
  toD f =  Fn (toD . f . fromD)
  fromD (Fn f) = fromD . f . toD
```

This implementation resembles the first version of cast, in that it must modify the argument to recover its type. To make this strategy efficient, Henglein (1992) designed an algorithm to produce well-typed code with as few coercions to and from the dynamic type as possible.

Another way to implement a dynamic type is to pair an expression with the *full* description of its type (Abadi *et al.*, 1991; Leroy & Mauny, 1991). The implementations GHC and Hugs use this strategy to provide a library supporting type Dynamic in Haskell. This library uses type classes to define term representations for each type. Injecting a value into type Dynamic involves tagging it with its representation, and projecting it compares the representation with a given representation to check that the types match. Though type classes can create appropriate term representations for each type, there is no support for the type comparison, so the last step of the projection requires an unsafe type coercion.

Although the second `cast` solution is more efficient than the universal datatype and more type safe than the GHC/Hugs library implementation, it suffers in terms of extensibility. The example implementations of `cast` only consider three type constructors, integers, products and functions. Others may be added, both primitive (such as `Char`, `IO`, or `[]`) and user defined (such as from datatype and newtype declarations), but only through modification of the `CT` type class. In contrast, the library implementation can be extended to support new type constructors without modifying previous code, as long as the invariant is maintained that each new type has a unique term representation.

A third implementation of a dynamic type that is type safe, efficient and easily extensible uses references (Weeks, 1998). Though references are not traditionally a component of a purely functional language, the Haskell implementations GHC and Hugs allow their use by encapsulation in the IO monad. While the previous implementations of the dynamic type defined the description of a type at compile time, this version creates the run-time description for a type at run time, and so is easily extendable to new types. Because each reference created by `newIORef` is unique, a unit reference can be used to implement a unique tag for a given type. A member of type Dynamic is then a pair of a tag and a computation that hides the stored value in its closure — a process that is similar to hiding the value within an existential type (Minamide *et al.*, 1996).

```
data Dyn = Dyn { tag :: IORef (), get ::  IO () }
```

To recover the value hidden in the closure, the `get` computation writes that value to a reference stored in the closure of the projection from the dynamic type. The computation `make` below creates injection and projection functions for any type.

```
make ::  IO (α → Dyn, Dyn → IO (Maybe α))
make = do newtag ← newIORef ()
          r ← newIORef undefined
          return ( λa → Dyn { tag = newtag,
                                get = writeIORef r a },
                   λd → if newtag == tag d
                        then do get d
                                x ← readIORef r
                                return (Just x)
                        else Nothing)
```

Unlike the previous versions that could not handle types with binding structure (such as `forall a. a → a`), this implementation can hide any type. Also, the complexity of projection from a dynamic type does not depend on the size of the type itself.

However, this implementation suffers from a number of drawbacks. It is more difficult to use, as it must be threaded through the IO monad. Furthermore, it would need additional synchronization to work in a concurrent program. In addition, because the tag is created dynamically, it cannot be used in an implementation for marshalling and unmarshalling. Finally, the user must be careful to call `make` only once for each type. (Conversely, the user is free to create more distinctions between types, in the same manner as the newtype mechanism).

Many languages support a natural implementation of tagging. For example, an extensible sum type (such as the exception type in SML) can be viewed as a dynamic type (Weeks, 1998). The declaration of a new exception constructor, `E`, carrying some type $\tau$ provides an injection from $\tau$ into the exception type. Coercing a value from the dynamic type to $\tau$ is matching the exception constructor with `E`.

In addition, if the language supports subtyping and downcasting, then a maximal supertype serves as a dynamic type. Ignoring the primitive types (such as int), Java (Gosling *et al.*, 1996) is an example of such a language. Any reference type may be coerced to type Object, without any run time overhead. Coercing from type Object requires checking whether the value's class (tagged with the value) is a subclass of the given class.

## Acknowledgements

Thanks to Karl Crary, Fergus Henderson, Chris Okasaki, Greg Morrisett, Dave Walker, Steve Zdancewic, and the anonymous reviewers for their many comments on earlier drafts of this paper.

## References

Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G. (1991) Dynamic typing in a statically-typed language. *ACM Trans. Program. Lang. Syst.* **13**(2), 237–268.

Baars, A. and Swierstra, S. D. (2002) Typing dynamic typing. In: Peyton Jones, S. (editor), *2002 ACM International Conference on Functional Programming*, pp. 157–166.

Cheney, J. and Hinze, R. (2002) Poor man's dynamics and generics. In: Chakravarty, M. M. T. (editor), *ACM SIGPLAN Haskell Workshop*, pp. 90–104.

Gosling, J., Joy, B. and Steele, G. (1996) *The Java Language Specification*. Addison-Wesley.

Harper, R. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. In: Lee, P. (editor), *Twenty-Second ACM Symposium on Principles of Programming Languages*, pp. 130–141.

Henglein, F. (1992) Dynamic typing. In: Krieg-Brückner, B. (ed), *Fourth European Symposium on Programming: LNCS 582*, pp. 233–253. Springer-Verlag.

Jones, M. P. (1995) A system of constructor classes: overloading and implicit higher-order polymorphism. *J. Funct. Program.* **5**(1).

Leroy, X. and Mauny, M. (1991) Dynamics in ML. In: Hughes, J. (editor), *Functional Programming Languages and Computer Architecture: LNCS 523*, pp. 406–426. Springer-Verlag.

Minamide, Y., Morrisett, G. and Harper, R. (1996) Typed closure conversion. In: Steele, G. (editor), *Twenty-Third ACM Symposium on Principles of Programming Languages*, pp. 271–283.

Mitchell, J. C. and Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* **10**(3), 470–502.

Odersky, M. and Läufer, K. (1996) Putting type annotations to work. In: Steele, G. (editor), *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 54–67. St. Petersberg Beach, FL.

Peyton Jones, S. (editor) (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. In: O'Donnell, M. J. and Feldman, S. (editors), *Sixteenth ACM Symposium on Principles of Programming Languages*, pp. 60–76. ACM.

Weeks, S. (1998) *NJ PearLS – Dynamically Extensible Data Structures in Standard ML*. Talk presented at New Jersey Programming Languages and Systems Seminar.