Type-Safe Cast

Stephanie Weirich[*]
Department of Computer Science
Cornell University
Ithaca, NY 14853
sweirich@cs.cornell.edu

ABSTRACT

In a language with non-parametric or ad-hoc polymorphism, it is possible to determine the identity of a type variable at run time. With this facility, we can write a function to convert a term from one abstract type to another, if the two hidden types are identical. However, the naive implementation of this function requires that the term be destructed and rebuilt. In this paper, we show how to eliminate this overhead using higher-order type abstraction. We demonstrate this solution in two frameworks for ad-hoc polymorphism: intensional type analysis and type classes.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [**Logics and Meanings of Programs**]: Software—*type structure, control primitives, functional constructs*

General Terms

Languages

Keywords

Ad-hoc polymorphism, dynamic typing, intensional type analysis, type classes

1. THE SETUP

Suppose you wanted to implement a heterogeneous symbol table — a finite map from strings to values of any type. You could imagine that the interface to this module would declare an abstract type for the table, a value for the empty table, and two polymorphic functions for inserting items into and retrieving items from the table. In a syntax resembling Standard ML [10], this interface is

```
sig
  type table
  val empty  : table
  val insert : ∀α. table → (string × α) → table
  val find : ∀α. table → string → α
end
```

However, looking at the type of `find` reveals something odd: If this polymorphic function behaves parametrically with respect to α, that is it executes the same code regardless of the identity of α, there cannot possibly be any correct implementation of `find` [15]. All implementations must either diverge or raise an exception. Let us examine several possible implementations to see where they go wrong.

We will assume that the data structure used to implement the symbol table is not the issue, so for simplicity we will use an association list.

```
val empty = []
```

Given a string and a list, the following version of `find` iterates over the list looking for the matching string:

```
let rec find =
  Λα. fn table => fn symbol =>
     case table of
       [] => raise NotFound
     | ( (name, value) :: rest ) =>
        if symbol = name then value
        else find[α] rest symbol
```

Note that, unlike in SML, we make type application explicit in a manner similar to the polymorphic lambda calculus or System F [4, 14]. The notation $\Lambda\alpha$ abstracts the type α and `find[α]` instantiates this type for the recursive call. Unfortunately, this function is of type

```
∀α. (string × α) list → string → α.
```

If we were looking up an `int`, the list passed to `find` could only contain pairs of `string`s and `int`s. We would not be

[*]This paper is based on work supported in part by the National Science Foundation under Grant No. CCR-9875536. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of this agency.

able to store values of any other type than `int` in our symbol table.

The problem is that, in a statically typed language, all elements of a list need to have the same type: It would be incorrect to form an association list `[ ("x", 1) ; ("y", (2,3)) ]`. As we do not want to constrain the symbol table to contain only one type, we need to hide the type of the value for each entry in the list. One possibility is to use an existential type [12]. The instruction

```
pack v as ∃β.τ hiding τ′
```

coerces a value of type τ with τ' substituted for β into one of type $\exists\beta.\tau$. Conversely, the instruction

```
unpack (β,x) = e in e'
```

destructs an existential package `e` of type $\exists\beta.\tau$, binding a new type variable β and a new term variable `x` of type τ within the expression `e'`.

Therefore, we can create a symbol table of type `(string` $\times$ $\exists\beta.\beta$`) list` with the expression

```
  [ ("x", pack 1 as ∃β.β hiding int) ;
    ("y", pack (2,3) as ∃β.β hiding int × int ]
```

So the code for insert should just package up its argument and cons it to the rest of the table.

```
val insert =
   Λα. fn table => fn (symbol, obj) =>
      (symbol, pack obj as ∃β.β hiding α) :: table
```

However, the existential type has not really solved the problem. We can create the list, but what do we do when we look up a symbol? It will have type $\exists\beta.\beta$, but `find` must return a value of type α. If we use the symbol table correctly, then α and β will abstract the same type, but we need to verify this property before we can convert something of type β to type α. We can not compare α and β and still remain parametric in α and β. Therefore, it seems as though we need a language that supports some sort of non-parametric polymorphism (also called run-time type analysis, overloading, or “ad hoc” polymorphism). Formulations of this feature include Haskell type classes [16], type Dynamic [1, 7, 9], extensional polymorphism [3], and intensional polymorphism [6].

For now, we will consider the last, in Harper and Morrisett's language λ_i^{ML}. This language contains the core of SML plus an additional `typerec` operator to recursively examine unknown types at run time. For simplicity, we will separate recursion from type analysis and use the operator `typecase` to discriminate between types, and `rec` to compute the least fixed point of a recursive equation.

This language is interpreted by a *type-passing* semantics. In other words, at run time a type argument is passed to a type abstraction of type $\forall\alpha.\tau$, and can be analyzed by `typecase`. Dually, when we create an object of an existential type, $\exists\alpha.\tau$, the hidden type is included in the package, and when the package is opened, α may also be examined. In λ_i^{ML}, universal and existential types have different properties from a system with a *type-erasure* semantics, such as the polymorphic lambda calculus. In a type-erasure system, types may have no effect on run-time execution and therefore may be erased after type checking. There, $\forall\alpha.\alpha$ is an empty type (such as `void`), and $\exists\beta.\beta$ is equivalent to the singleton type `unit`. However, in λ_i^{ML}, $\forall\alpha.\alpha$ is not empty, as we can use `typecase` to define an appropriate value for each type, and $\exists\beta.\beta$ is the implementation of type Dynamic, as we can use `typecase` to recover the hidden type.

In λ_i^{ML}, a simple function, `sametype`, to compare two types and return true if they match, can be implemented with nested `typecase`s. The outer `typecase` discovers the head normal form of the first type, and then the inner `typecase` compares it to the head normal form of the second.[1] For product and function types, `sametype` calls itself recursively on the subcomponents of the type. Each of these branches binds type variables (such as α_1 and α_2) to the subcomponents of the types so that they may be used in the recursive call.

```
let rec sametype  =
   Λα. Λβ.
      typecase (α) of
        int =>
          typecase (β) of
            int => true
          | _   => false
      | (α1 × α2)=>
          typecase (β) of
            (β1 × β2) =>
            sametype[α1][β1]
               andalso sametype[α2][β2]
          | _ => false
      | (α1 → α2) =>
          typecase (β) of
            (β1 → β2) =>
            sametype[α1][β1]
               andalso sametype[α2][β2]
          | _ => false
```

As these nested `typecase`s are tedious to write, we borrow from the pattern matching syntax of Standard ML, and abbreviate this function as:

[1]For brevity, we only include `int`, product types, and function types in the examples.

```
let rec sametype  =
   Λα. Λβ.
      typecase (α,β) of
        (int,int) => true
      | (α₁ × α₂,β₁ × β₂) =>
        sametype[α₁][β₁]
           andalso sametype[α₂][β₂]
      | (α₁ → α₂,β₁ → β₂) =>
        sametype[α₁][β₁]
           andalso sametype[α₂][β₂]
      | (_,_) => false
```

However, though this function does allow us to determine if two types are equal, it does not solve our problem. In fact, it is just about useless. If we try to use it in our implementation of `find`

```
let rec find =
   Λα. fn table => fn symbol =>
      case table of
        [] => raise NotFound
      | ( (name, package) :: rest ) =>
        unpack (β,value) = package in
        if symbol = name
              andalso sametype[α][β]
        then value
        else find[α] rest symbol
```

we discover that this use does not type check. The return type for `find` is the existentially bound β which escapes its scope. Even though we have added a dynamic check that α is equivalent to β, the check does not change the type of `value` from β to α.

Our problem is that we did not use the full power of the type system. In a standard case expression (as opposed to a `typecase`), each branch must be of the same type. However, in λ_i^{ML} the type of each branch of a `typecase` can depend on the analyzed type.

```
typecase τ of
   int => fn (x:int) => x + 3
 | α → β =>
   fn(x:α → β) => x
 | α × β =>
   fn(x:α × β) => x
```

For example, although the first branch above is of type `int` $\to$ `int`, the second of type $(\beta \to \gamma) \to (\beta \to \gamma)$, and the third of type $(\beta \times \gamma) \to (\beta \times \gamma)$, all three branches are instances of the type schema $\gamma \to \gamma$, when γ is replaced with the identity of τ for that branch. Therefore, this entire `typecase` expression can be safely assigned the type $\tau \to \tau$.

With this facility, in order to make typechecking a `typecase` term syntax-directed, it is annotated with a type variable and a type schema where that variable may occur free. For example we annotate the previous example as

```
typecase [γ.γ → γ] τ of
      int => fn (x:int) => x + 3
    | ...
```

In later examples, when we use the pattern matching syntax for two nested typecases, we will need the schema to have two free type variables.

We now have everything we need to write a version of `sametype` that changes the type of a term and allows us to write `find`. In the rest of this paper we will develop this function, suggestively called `cast`, of type $\forall\alpha.\forall\beta.\alpha \to \beta$. This function will just return its argument (at the new type) if the type arguments match, and raise an exception otherwise.[2]

An initial implementation appears in Section 2. Though correct, its operation requires undesirable overhead for what is essentially an identity function. We improve it, in Section 3, through the aid of an additional type constructor argument to `cast`. To demonstrate the applicability of this solution to other non-parametric frameworks, in Section 4, we develop the analogous two solutions in Haskell using type classes. In Section 5, we compare these solutions with several implementations of type Dynamic. Finally, in Section 6, we conclude by eliminating the type classes from the Haskell solution to produce a symbol table implementation using only parametric polymorphism. As such, the types of the functions `insert` and `find` must be modified.

```
cast : ∀α. ∀β. α → β
let rec cast =
   Λα. Λβ.
      typecase [δ₁, δ₂. δ₁ → δ₂](α,β) of
        (int, int) =>
          fn (x:int) => x
      | (α₁ × α₂, β₁ × β₂) =>
          let val f = cast [α₁][β₁]
              val g = cast [α₂][β₂]
          in
             fn (x:α₁ × α₂) =>
               (f (fst x), g (snd x))
          end
      | (α₁ → α₂, β₁ → β₂) =>
          let val f = cast [β₁][α₁]
              val g = cast [α₂][β₂]
          in
            fn (x:α₁ → α₂) =>
               g ∘ x ∘ f
          end
      | (_,_) => raise CantCast
```

Figure 1: First Solution

2. FIRST SOLUTION

An initial implementation of `cast` using the facilities of λ_i^{ML} appears in Figure 1. In the first branch of the `typecase`, α

[2]It would also be reasonable to produce a function of type $\alpha \to (\beta$ `option`), but checking the return values of recursive calls for `NONE` only lengthens the example.

and β have been determined to both be to `int`. Casting an `int` to an `int` is easy; just an identity function.

In the second branch of the `typecase`, both α and β are product types ($\alpha_1 \times \alpha_2$ and $\beta_1 \times \beta_2$ respectively). Through recursion, we can cast the subcomponents of the type (α_1 to β_1 and α_2 to β_2). Therefore, to cast a product, we break it apart, cast each component separately, and then create a new pair.

The code is a little different for the next branch, when α and β are both function types, due to contravariance. Here, given `x`, a function from α_1 to α_2, we want to return a function from β_1 to β_2. We can apply cast to α_2 and β_2 to get a function, `g`, that casts the result type, and compose `g` with the argument `x` to get a function from α_1 to β_2. Then we can compose that resulting function with a reverse cast from β_1 to α_1 to get a function from β_1 to β_2.

Finally if the types do not match we raise the exception `CantCast`.

However, there is a problem with this solution. Intuitively, all a cast function should do at run time is recursively compare the two types. But unless the types τ_1 and τ_2 are both `int`, the result of `cast[`τ_1`][`τ_2`]` does much more. Every pair in the argument is broken apart and remade, and every function is wrapped between two instantiations of cast. This operation resembles a virus, infecting every function it comes in contact with and causing needless work for every product.

The reason we had to break apart the pair in forming the coercion function for product types is that all we had available was a function (from $\alpha_1 \to \beta_1$) to coerce the first component of the pair. If we could somehow create a function that coerces this component while it was still a part of the pair, we could have applied it to the pair as a whole. In other words, we want two functions, one from $(\alpha_1 \times \alpha_2) \to (\beta_1 \times \alpha_2)$ and one from $(\beta_1 \times \alpha_2) \to (\beta_1 \times \beta_2)$.

3. SECOND SOLUTION

Motivated by the last example, we want to write a function that can coerce the type of *part* of its argument. This will allow us to pass the same value as the `x` argument for each recursive call and only refine part of its type. We can not eliminate `x` completely, as we are changing its type. Since we want to cast many parts of the type of `x`, we need to abstract the relationship between the type argument to be analyzed and the type of `x`.

The solution in Figure 2 defines a helper function `cast'` that abstracts not just the types α and β for analysis, but an additional *type constructor*[3] argument γ. When γ is applied to the type α we get the type of `x`, when it is applied to β we get the return type of the cast. For example, if γ is $\lambda\delta{:}*.\delta \times \alpha_2$, we get a function from type $\alpha \times \alpha_2$ to $\beta \times \alpha_2$.

[3]We create type constructors with λ-notation, and annotate the bound variable with its *kind*. Kinds classify types and type constructors: $*$ is the kind of types, and if κ_1 and κ_2 are kinds, $\kappa_1 \to \kappa_2$ is the the kind of type constructors from κ_1 to κ_2.

```
cast' : $\forall\alpha,\beta{:}*.\forall\gamma{:}* \to *.(\gamma\ \alpha) \to (\gamma\ \beta)$
let rec cast' =
  $\Lambda\alpha{:}*.\Lambda\beta{:}*.\Lambda\gamma{:}* \to *.$
    typecase [$\delta_1$, $\delta_2$. ($\gamma$ $\delta_1$) $\to$ ($\gamma$ $\delta_2$)]($\alpha$,$\beta$) of
       (int, int) =>
         fn (x:$\gamma$ int) => x
     | ($\alpha_1 \times \alpha_2$,$\beta_1 \times \beta_2$) =>
         let val f = cast'[$\alpha_1$][$\beta_1$][$\lambda\delta{:}*.\ \gamma(\delta \times \alpha_2)$]
             val g = cast'[$\alpha_2$][$\beta_2$][$\lambda\delta{:}*.\ \gamma(\beta_1 \times \delta)$]
         in
           fn (x:$\gamma(\alpha_1 \times \alpha_2)$) =>
             g (f x)
         end
     | ($\alpha_1 \to \alpha_2$,$\beta_1 \to \beta_2$) =>
         let val f = cast'[$\alpha_1$][$\beta_1$][$\lambda\delta{:}*.\ \gamma(\delta \to \alpha_2)$]
             val g = cast'[$\alpha_2$][$\beta_2$][$\lambda\delta{:}*.\ \gamma(\beta_1 \to \delta)$]
         in
           fn (x:$\gamma(\alpha_1 \to \alpha_2)$) =>
             g (f x)
         end
     | (_,_) => raise CantCast
```

Figure 2: Second Solution

Since we abstract both types and type constructors, in the definition of cast we annotate α, β, and γ with their kinds. As α and β are types, they are annotated with kind $*$, but γ is a function from types to types, and so has kind $* \to *$. We initially call `cast'` with the identity function.

```
let cast =
    $\Lambda\alpha{:}*.\Lambda\beta{:}*.$ cast'[$\alpha$][$\beta$][$\lambda\delta{:}*.\delta$]
```

With the recursive call to `cast'`, in the branch for product types we create a function to cast the first component of the tuple (converting α_1 to β_1) by supplying the type constructor $\lambda\delta{:}*.\gamma(\delta \times \alpha_2)$ for γ. As `x` is of type $\gamma(\alpha_1 \times \alpha_2)$, this application results in something of type $\gamma(\beta_1 \times \alpha_2)$. In the next recursive call, for the second component of the pair, the first component is already of type β_2, so the type constructor argument reflects that fact.

Surprisingly, the branch for comparing function types is analogous to that of products. 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.

4. HASKELL

The language Haskell [13] provides a different form of ad hoc polymorphism, through the use of type classes. Instead of defining one function that behaves differently for different types, Haskell allows you to define several functions with the same name that differ in their types.

For example, the Haskell standard prelude defines the class of types that support a conversion to a string representation.

This class is declared by

```
class Show $\alpha$ where
  show :: $\alpha$ $\rightarrow$ string
```

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

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

We can also define `show` for type constructors like product. In order to convert a product to a string we need to be able to `show` each component of the product. Haskell allows you to express these conditions in the instantiation of a type constructor:

```
instance (Show $\alpha$, Show $\beta$) => Show ($\alpha$,$\beta$) where
  show (a,b) = "(" ++ show a ++ "," ++ show b ")".
```

This code declares that as long as α and β are members of the class `Show`, then their product (Haskell uses `,` instead of $\times$ for product) is a member of class `Show`. Consequently, the function `show` for products is defined in terms of the `show` functions for its subcomponents.

4.1 First Solution in Haskell

Coding the cast example in Haskell is a little tricky because of the two nested `typecase` terms (hidden by the pattern-matching syntax). For this reason we will 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 will implement all of the inner `typecases`. The first class just defines the cast function, but the second class needs to include three functions, describing how to complete the cast using the knowledge that the first type was an integer, product, or function. Note the contravariance in the type of `doFn` below: Because we will have to cast from β_1 to α_1, we need α_1 to be in the class `CT` instead of `CF`.

```
class CF $\alpha$ where
  cast :: CT $\beta$ => $\alpha$ $\rightarrow$ $\beta$
class CT $\beta$ where
  doInt  :: Int $\rightarrow$ $\beta$
  doProd :: (CF $\alpha_1$, CF $\alpha_2$) => ($\alpha_1$, $\alpha_2$) $\rightarrow$ $\beta$
  doFn   :: (CT $\alpha_1$, CF $\alpha_2$) => ($\alpha_1 \rightarrow \alpha_2$) $\rightarrow$ $\beta$
```

Just as in λ_i^{ML}, where the outer `typecase` led to an inner `typecase` in each branch, the separate instances of the first class just dispatch to the second, marking the head constructor of the first type by the function called:

```
instance CF Int where
    cast = doInt
instance (CF $\alpha$, CF $\beta$) => CF ($\alpha$, $\beta$) where
    cast = doProd
instance (CT $\alpha$, CF $\beta$) => CF ($\alpha \rightarrow \beta$) where
    cast = doFn
```

The instances of the second type class either implement the branch (if the types match) or signal an error. In the `Int` instance of `CT`, the `doInt` function is an identity function as before, while the others produce the error `"CantCast"`.

```
instance CT Int where
    doInt x = x
    doProd x = error "CantCast"
    doFn x = error "CantCast"
```

The `doProd` function of the product instance should be of type `(CF` α_1`, CF` α_2`) => (`α_1`, `α_2`)` $\rightarrow$ `(`β_1`, `β_2`)`. This function calls `cast` recursively on `x` and `y`, the subcomponents of the product (taken apart via pattern matching). As the types of `x` and `y` are α_1 and α_2, and we are allowed to assume they are in the class `CF` , we can call `cast`. The declaration of `cast` requires that its result type be in the class `CT`, so we require that β_1 and β_2 be in the class `CT` for this instantiation.

```
instance (CT $\beta_1$, CT $\beta_2$) => CT ($\beta_1$, $\beta_2$) where
    doInt x = error "CantCast"
    doProd (x,y) = (cast x,  cast y)
    doFn x = error "CantCast"
```

Finally, in the instance for the function type constructor, `doFn` needs to wrap its argument in recursive calls to `cast`, as in the first λ_i^{ML} solution. As the type of this function should be `(CT` α_1`, CF` α_2`) => (`$\alpha_1 \rightarrow \alpha_2$`)` $\rightarrow$ `(`$\beta_1 \rightarrow \beta_2$`)`, the type of the argument `x` is of a function of type $\alpha_1 \rightarrow \alpha_2$. To `cast` the result of this function, we need the α_2 instance of `CF`, that requires that β_2 be in the class `CT`. However, we would also like to `cast` the argument of the function in the opposite direction, from β_1 to α_1. Therefore we need β_1 to be in the class `CF`, and α_1 to be in the class `CT`. The instance for function types is then (Haskell uses `f . g` for function composition)

```
instance (CF $\beta_1$, CT $\beta_2$) => CT ($\beta_1 \rightarrow \beta_2$) where
    doInt x = error "CantCast"
    doProd x = error "CantCast"
    doFn x = cast . x . cast
```

Now with these definitions we can define the symbol table using an existential type to hide the type of the element in the table. Though not in the current language definition, several implementations of Haskell support existential types. Existential types in Haskell are denoted with the `forall` keyword as the data constructors that carry them may be instantiated with any type.

```
data EntryT = forall $\alpha$ . CF $\alpha$ =>  Entry String $\alpha$
type table = [ EntryT ]
```

The `find` function is very similar to before, except that Haskell infers that the existential type should be unpacked.

```
find ::  CT α => table → String → α
find ( (Entry s2 val) : rest ) s1 =
       if s1 == s2 then
         cast val
       else find rest s1
find [] s = error "Not in List"
```

4.2 Second Solution in Haskell

To implement the second version in Haskell, we need to change the type of cast to abstract the type constructor γ as well as α and β. This addition leads to the new definitions of CT and CF. Note that in the type of doFn, α_1 should be in the class CF instead of the class CT, reflecting that we are going to avoid the contravariant cast of the function argument that we needed in the previous solution.

```
class CF α where
   cast' :: CT β => γ α → γ β
class CT β where
   doInt  :: γ Int → γ β
   doProd :: (CF α₁, CF α₂) =>
                   γ (α₁, α₂) → γ β
   doFn   :: (CF α₁, CF α₂) =>
                   γ (α₁→α₂) → γ β
```

The instances for CF remain the same as before, dispatching to the appropriate functions. Also the instance CT Int is still the identity function. But recall the branch for products:

```
typecase (α,β) of
   ...
   | (α₁ × α₂,β₁ × β₂) =>
       let val f = cast'[α₁][β₁][λδ:∗. γ(δ × α₂)]
           val g = cast'[α₂][β₂][λδ:∗. γ(β₁ × δ)]
       in
         fn (x:γ(α₁ × α₂)) =>
           g (f x)
       end
```

The strategy was 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 examineded. In Haskell we cannot explicitly instantiate the type constructor argument as we did in λ_i^{ML}, but we can give Haskell enough hints to infer the correct one.[4] To represent the two type constructor arguments above we use the data constructors LP and RP, defined below.

```
newtype LProd α γ δ = LP (γ (δ, α))
newtype RProd α γ δ = RP (γ (α, δ))
instance (CT β₁, CT β₂) => CT (β₁, β₂) where
   doInt x = error "CantCast"
   doProd z = x
        where LP y = cast' (LP z)
              RP x = cast' (RP y)
   doFn x = error "CantCast"
```

[4]Jones [8] describes this sort of implicit higher-order polymorphism in Haskell.

How does this code typecheck? In this instance, doProd should be of type

```
(CF α₁, CF α₂) => γ (α₁,α₂) → γ (β₁,β₂).
```

Therefore z is of type γ (α_1, α_2) so LP z is of type LProd α_2 γ α_1. At first glance, it seems like we cannot call cast' on this argument because we have not declared an instance of CF for the type constructor LProd. However, the instances of cast' are all of type $\forall\gamma'$.(CT β) => $\gamma'\alpha \rightarrow \gamma'\beta$, so to typecheck the call cast' (LP z), we only need to find an α in the class CT and a γ' such that $\gamma'\alpha$ is equal to the type of LP z. As Haskell does not permit the creation of type-level type abstractions [8], the type of LP z must explicitly be a type constructor applied to a type in order to typecheck. Therefore determining γ' and α is a simple match – γ' is the partial application LProd α_2 γ and α is α_1. Thus, the result of cast' is of type (LProd α_2 γ) β, for some β in CT, and y is of type γ (β, α_2).

Now RP y is of type RProd β γ α_2, so we need the α_2 instance of cast' for the second call. This instance is of type $\forall\gamma''$. CT β' => γ'' α_2 $\rightarrow$ γ'' β'. This γ'' unifies with the partial application (RProd β γ) so the return type of this cast is RProd β γ β', the type of RP x. That makes x of type γ (β, β'). Comparing this type to the return type of doProd, we unify β with β_1 and β' with β_2. This unification satisfies our constraints for the two calls to cast', as we assumed that both β_1 and β_2 are in the class CT.

Just as in the second λ_i^{ML} solution, function types work in exactly the same way as product types, using similar declarations of LArrow and RArrow.

```
newtype LArrow α γ δ = LA (γ (δ → α))
newtype RArrow α γ δ = RA (γ (α → δ))
```

To encapsulate cast', we provide the identity type constructor:

```
newtype Id α = I α
cast :: (CF α, CT β) => α → β
cast x = y where (I y) = cast' (I x)
```

The complete Haskell code for this solution appears in Appendix A.

5. IMPLEMENTING TYPE DYNAMIC

Just as $\exists\alpha.\alpha$ implements a dynamic type in λ_i^{ML}, $\exists\alpha$. CF α => α is a dynamic type in Haskell. Adding type Dynamic to a statically typed language is nothing new, so it is interesting to compare this implementation to previous work.

One way to implement type Dynamic (explored by Henglein [7]) 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 only 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` $\rightarrow$ `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


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 our first cast solutions, in that it must destruct the argument to recover its type. Also, projecting a function from type Dynamic results in a wrapped version. Henglein, in order to make this strategy efficient, designs an algorithm to produce well-typed code with as few coercions to and from the dynamic type as possible.

Another way to implement type Dynamic is to pair an expression with the *full* description of its type [1, 9]. 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. At first glance this scheme is surprisingly similar to an implementation of cast using `typecase` in λ_R [2], a version of λ_i^{ML} in which types are explicitly represented by dependently typed terms. However there is an important distinction: Though type classes can create appropriate term representations for each type, there is no support for dependency, so the last step of the projection requires an unsafe type coercion.

Although the `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, for 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. Furthermore, all current instantiations of `CT` need to be extended with error functions for each additional type constructor. In contrast, the library implementation can be extended to support new type constructors without modifying previous code.

A third implementation of type Dynamic that is type safe, efficient and easily extensible uses references (see Weeks [17] for the original SML version). Though the use of mutable references is not typically encouraged (or even possible) in a purely functional language, GHC and Hugs support their use by encapsulation in the IO monad. While the previous implementations of type Dynamic defined the description of a type at compile time, references allow the creation of a description for any type at run time, and so are 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.[5]

```
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 α)
make = do { newtag <- newIORef ()
          ; r <- newIORef Nothing
          ; return
            (\a -> Dyn { tag = newtag,
                    get = writeIORef r (Just a) },
             \d -> if (newtag == tag d)
             then
             do { get d
                ; Just x <- readIORef r
                ; return x
                }
             else error "Ill typed")
          }
```

This implementation of type dynamic is more difficult to use, as it must be threaded through the IO monad. Furthermore, because the tag is created dynamically, it cannot be used in an implementation for marshalling and unmarshalling. Also, the user must be careful to call `make` only once for each type, lest she confuse them. (Conversely, the user is free to create more distinctions between types, in much the same manner as the newtype mechanism). Unlike the previous versions that could not handle types with binding structure (such as `forall a.  a -> a`), this solution can hide any type. Additionally, the complexity of projection from a dynamic type does not depend on the type itself. However, the above solution has a redundant comparison – if the tags match then the pattern match `Just x` will never fail, but the implementation must still check that that the reference does not contain `Nothing`.

If we wander outside of the language Haskell, we find language support for a more natural implementation of tagging, thereby eliminating this redundant check. For example, if the language supports an extensible sum type (such as the exception type in SML) then that type can be viewed as

[5]Which can be viewed as hiding the value within an existential type [11].

type Dynamic [17]. The declaration of a new exception constructor, `E`, carrying some type τ provides an injection from τ into the exception type. Coercing a value from the dynamic type to τ is matching the exception constructor with `E`.

Alternatively, if the language supports subtyping and downcasting, then a maximal supertype serves as a dynamic type. Ignoring the primitive types (such as int), Java [5] 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 subtype of the given class.

6. PARAMETRIC POLYMORPHISM

The purpose of this paper is not to find the best implementation of dynamic typing, but instead to explore what language support is necessary to implement it and at what cost. With the addition of first-class polymorphism (such as supported by GHC or Hugs), Haskell type classes may be completely compiled away [16]. Therefore, the final Haskell typeclass solution can be converted to a framework for implementing heterogeneous symbol tables in a system of parametric polymorphism. Appendix B shows the result of a standard translation to dictionary-passing style, plus a sample run.

The original problem we had with the function `find` was with its type; though α was universally quantified, it did not appear negatively. The translation of the Haskell solution shows us exactly what additional argument we need to pass to `find` (the `CT` dictionary for the result type), and exactly what additional information we need to store with each entry in the symbol table (the `CF` dictionary for the entry's type) in order to recover the type.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, Sept. 1998. Extended version published as Cornell University technical report TR98-1721.

[3] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, Jan. 1995.

[4] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[7] F. Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 233–253. Springer-Verlag, Feb. 1992.

[8] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), Jan. 1995.

[9] X. Leroy and M. Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 406–426. Springer-Verlag, Aug. 1991.

[10] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[11] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, Jan. 1996.

[12] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[13] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, Feb. 1999. Available from http://www.haskell.org/definition/.

[14] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, 1974.

[15] P. Wadler. Theorems for free! In *Fourth Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.

[16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[17] S. Weeks. NJ PearLS – dynamically extensible data structures in Standard ML. Talk presented at New Jersey Programming Languages and Systems Seminar, Sept. 1998. Transparencies available at http://www.star-lab.com/sweeks/talks.html.

APPENDIX

A. HASKELL LISTING FOR SOLUTION 2

```
newtype LProd a g d = LP (g (d, a))
newtype RProd a g d = RP (g (a, d))
newtype LArrow a g d = LA (g (d -> a))
newtype RArrow a g d = RA (g (a -> d))
newtype Id a = I a

class CF a where
    cast' :: CT b => g a -> g b
instance CF Int where
    cast' = doInt
instance (CF a, CF b) => CF (a, b) where
    cast' = doProd
instance (CF a, CF b) => CF (a->b) where
    cast' = doFn

class CT b where
    doInt  :: g Int -> g b
    doProd :: (CF a1, CF a2) =>
                  g (a1, a2) -> g b
    doFn   :: (CF a1, CF a2) =>
                  g (a1->a2) -> g b
instance CT Int where
    doInt x = x
    doProd x = error "CantCF"
    doFn x = error "CantCast"
instance (CT b1, CT b2) => CT (b1, b2) where
    doInt x = error "CantCast"
    doProd z = x
        where LP y = cast' (LP z)
        where RP x = cast' (RP y)
    doFn x = error "CantCast"
instance (CT b1, CT b2) => CT (b1->b2) where
    doInt x = error "CantCast"
    doProd x = error "CantCast"
    doFn z = x
        where LA y = cast' (LA z)
        where RA x = cast' (RA y)

cast :: (CF a, CT b) => a -> b
cast x = y where I y = cast' (I x)
```

B. PARAMETRIC SYMBOL TABLE

B.1 Dictionary-passing Implementation

```
newtype LProd a g d = LP (g (d, a))
newtype RProd a g d = RP (g (a, d))
newtype LArrow a g d = LA (g (d -> a))
newtype RArrow a g d = RA (g (a -> d))
newtype Id a = I a

data CF a = CastFromDict
  { cast' :: forall b g. CT b -> g a -> g b }

data CT b =
  CastToDict
  { doInt  :: (forall g. g Int -> g b),
    doProd :: (forall a1 a2 g.
      CF a1 -> CF a2 -> g (a1,a2) -> g b),
    doFn   :: (forall a1 a2 g. CF a1 ->
      CF a2 -> g (a1->a2)-> g b) }
```

```
-- CF dictionary constructors

cfInt :: CF Int
cfInt  = CastFromDict { cast' = doInt }

cfProd :: CF a -> CF b -> CF (a,b)
cfProd = \x -> \y -> CastFromDict
   { cast' = (\ct -> (doProd ct x y)) }

cfFn :: CF a -> CF b -> CF (a->b)
cfFn   = \x -> \y -> CastFromDict
   { cast' = (\ct -> (doFn ct x y)) }

-- CT dictionary constructors

ctInt :: CT Int
ctInt = CastToDict
  { doInt  = (\x -> x),
    doProd = (\x -> error "CantCast"),
    doFn   = (\x -> error "CantCast") }

ctProd :: CT b1 -> CT b2 -> CT (b1, b2)
ctProd = \ctb1 -> \ctb2 -> CastToDict
  { doInt  = (\x -> error "CantCast"),
    doProd = (\cfa1 -> \cfa2 -> \z ->
     let LP y = cast' cfa1 ctb1 (LP z)
         RP x = cast' cfa2 ctb2 (RP y)
     in x),
    doFn   = (\x -> error "CantCast") }

ctFn :: CT b1 -> CT b2 -> CT (b1 -> b2)
ctFn = \ctb1 -> \ctb2 -> CastToDict
  { doInt  = (\x -> error "CantCast"),
    doProd = (\x -> error "CantCast"),
    doFn   = (\cfa1 -> \cfa2 -> \z ->
      let LA y = cast' cfa1 ctb1 (LA z)
          RA x = cast' cfa2 ctb2 (RA y)
      in x) }

-- Wrapping up cast'

cast :: CF a -> CT b -> a -> b
cast cfa ctb x = y
    where (I y) = cast' cfa ctb (I x)
```

B.2 Symbol Table Implementation

```
data EntryT = forall b . Entry String b (CF b)
type Table = [ EntryT ]

empty :: Table
empty = []

-- Insertion requires the correct CF dictionary
insert :: CF a -> Table -> (String, a) -> Table
insert cf t (s,a) = (Entry s a cf) : t

-- The first argument to find is a Cast To
-- dictionary
find :: CT a -> Table -> String -> a
find ct [] s = error "Not in List"
find ct (( Entry s2 val cf) : rest) s1 =
       if s1 == s2 then cast cf ct val
       else find ct rest s1
```

```
-- Create a symbol table by providing the
-- CF dictionary for each entry

table :: Table
t1 = insert ctInt empty (‘‘foo’’, 6)
t2 = insert (cfProd cfInt cfInt) t1 ("bar", (6,6))
table = insert (cfFn cfInt cfInt) t2
           ("add1" (\x->x+1)
```

B.3 Sample Run

```
Main> (find (ctFn int int) table "add1") 7
8
Main> find (ctProd int int) table "bar"
(6,6)
Main> find (ctProd int (ctProd int int)) table "bar"

Program error: CantCast
```