Exploring Dynamic Types in Haskell

Omar Navarro Leija

University Of Pennsylvania

December 19, 2016

Introduction

Haskell enjoys a wealth of innovation and application of ongoing research through type system extensions. Many extensions have expanded the expressiveness of Haskell's type system such as GADTs, data kinds, kind polymorphism, kind equality, and soon, dependent types! Haskell's rich type system allows us to express complicated constrains on our functions and data. Moreover, proponents and critics of Haskell agree the strong type system is one of Haskell's strength, giving rise to the phrase "if a Haskell program compiles, it probably works". Even so, dynamic typing still has place in this brave new world. Recent work [1] motivates and implements type safe dynamics types in Haskell, further showing off the power of Haskell's type system.

The dynamic type

Recent advances in Haskell's type system (needing all the way up to GHC 8.0!) allow for a new implementation of *Typeable* and *Dynamic* in Haskell. Dynamic types require reflection, the ability to examine types at runtime and make decisions based on that type, which is made possible by the *Typeable* typeclass, a typed-indexed runtime representation of types, *TypeRep a*. The *Dynamic* type carries a type representation and a value, while the *Typeable* typeclass contains a single type representation:

```haskell
data Dynamic where
  Dyn :: TypeRep a -> a -> Dynamic

class Typeable (a :: k) where
  typeRep :: TypeRep a
```

Notice the type *a* is existentially bound so it does not appear as part of our data constructor for *Dynamic*. All types are automatically given a *Typeable* instance to guarantee unique type representations, this requires built in compiler support (the current implementation is a "fake" interface implementing the typeclass for only a few types, but enough for our purposes). Therefore, Haskell dynamic types can be used for all built-in and user defined types i.e. an open world implementation.

The *Dynamic* library provides the following functions for converting to and from dynamic types. Note the *fromDynamic* function returns *Nothing* if the value cannot be cast to an *a* type. Hence a user may cast any type to and from a dynamic type giving us the flexibility we seek from dynamic typing:

```
toDynamic :: Typeable a => a -> Dynamic

fromDynamic :: forall a. Typeable a => Dynamic -> Maybe a
```

A dynamically typed language in Haskell

Consider the following simple dynamic language *Lang*. This language is the lambda calculus extended with if-expressions, integers, booleans, strings, chars, tuples, and lists. The syntax for *Lang* is as follows:

⟨*expr*⟩ ::= ‘(fun’ ⟨*id*⟩ ‘.’ ⟨*expr*⟩‘)’
  | ‘(’ ⟨*expr*⟩ ⟨*expr*⟩ ‘)’
  | ‘(’⟨*binOp*⟩ ⟨*expr*⟩ ⟨*expr*⟩‘)’
  | ‘(’⟨*uniOp*⟩ ⟨*expr*⟩‘)’
  | ⟨*literal*⟩
  | variable
  | ‘(if’ ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩‘)’

⟨*literal*⟩ ::= ‘(’ ⟨*expr*⟩ ‘,’ ⟨*expr*⟩ ‘)’
  | number
  | char
  | string
  | ‘true’
  | ‘false’
  | ‘[’ ⟨*expr*⟩ , ... ‘]’

⟨*binOp*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘||’ | ‘&&’ | ‘<=’ | ‘=>’ | ‘<’ | ‘>’ | ‘==’ | ‘cons’

⟨*uniOp*⟩ ::= ‘~’ | ‘fst’ | ‘snd’ | ‘head’ | ‘tail’

Basically, a simple lisp like language. Using the dynamic library we may implement a dynamically typed language embedded in Haskell. All dynamically typed languages must perform runtime checks on objects to ensure operators and functions match the expected types. Using facilities from the *Dynamic* library we may perform this runtime check in Haskell. As it is implemented in Haskell, our language has lazy semantics and call by name evaluation. Example expressions for this language look like:

```
(+ 3 2) ; binary operations
(fun x. x)
(fun x. (cons x [])) ; function declaration
(fun x. fun y. (x, y) ; make tuple
; y-combinator
(fun f.
  ((fun x. (f (x x))) (fun x. (f (x x)))))
; factorial function
(yComb (fun f.
         (fun n.
           (if (== n 0)
             1
             (* n (f (- n 1)))))))
```

We may implement the AST for the language as a GADT:

```
data Exp where
  Lam  :: String -> Exp -> Exp      -- ^ Function abstraction
  (:@) :: Exp -> Exp -> Exp         -- ^ Function application
  Bin  :: BinOp -> Exp -> Exp -> Exp
  Uni  :: UniOp -> Exp -> Exp
  Lit  :: forall t. (LangType t) => t -> Exp
  Var  :: String -> Exp
  If   :: Exp -> Exp -> Exp -> Exp
  Nill :: Exp                       -- ^ lists

class (Typeable a, Eq a, Show a) => LangType a
```

LangType is a simple typeclass to restrict the types we allow in our language to a small subset of types: integers, chars, booleans, and strings. Tuples and Lists are implemented as language constructs and operators. This allows tuples and lists literals to have arbitrary expressions inside their body.

The evaluator for *Lang* has a straight forward implementation. At every step, it takes an expression and an environment, recurses on the subexpressions, and returns the results as a dynamic type. The environment is simply a map from Strings (representing bound variables) to dynamic values (showing us one use of dynamic types!). While simple, it is inefficient as we will cast all expressions from and to dynamic.

Lists and tuples are more difficult to implement as a consequence of our evaluator. Originally, I wanted the following invariant for tuples/lists: all tuples/lists would be represented using Haskell's tuples/lists where the elements are actual values, then the entire tuple/list would be 'wrapped' in a Dynamic type e.g:

```
toDynamic (5, 6)
```

This won't work as the results of evaluating the tuples left and right sides return dynamics (at this point we cannot know the types of each side to cast from dynamic). Instead our lists and tuples are represented as types holding dynamics:

```
data List where
  Add :: Dynamic -> List -> List
  End  :: List

data Tuple where
  Tuple :: Dynamic -> Dynamic -> Tuple
```

Then the expression

```
(fun x. (fun y. [1, x, y]))
```

is syntatic sugar for

```
(Lam "x" (Lam "y" (Cons (Lit 1) (Cons (Var "x") (Cons (Var "y") Nill)))))
```

Another consequence of this approach is the implementation of equality. To compare to dynamic types we must get their actual type and value, ensure the types match, and then compare. I define the Eq class for dynamics using the function

```
  dynEq d1 d2 =
  case (fromDynamic d1, fromDynamic d2) of                 -- Integers.
      (Just (i1 :: Integer), Just (i2 :: Integer)) -> i1 == i2
      (_,_) -> case (fromDynamic d1, fromDynamic d2) of    -- Booleans.
        (Just (b1 :: Bool), Just (b2 :: Bool)) -> b1 == b2
        (_,_) -> case (fromDynamic d1, fromDynamic d2) of  -- Chars.
          (Just (c1 :: Char), Just (c2 :: Char)) -> c1 == c2
        -- ... Enumerate all types we care about here...
```

As a design choice, if types type don't match this is an error, as when would we want to compare different types for equality? This makes the Eq instances for lists and tuples easy, as it's just element-wise comparisons. However, we will have to cast the value from dynamic, and recursively cast every element of the data structure making comparisons expensive.

A statically typed target language

Ideally, we want the flexibility of dynamic types without giving up performance or static analysis of our code. Therefore, we will compile our *Lang* into a statically typed language with dynamic types omitting as many casts as we can. *StaticLang* is the first iteration of this target language. All expressions are typed to enforce well-typed expressions:

```
data StaticExp t where
  Lam  :: String -> StaticExp a -> StaticExp (Dynamic -> a)
  (:@) :: StaticExp (Dynamic -> t2) -> StaticExp Dynamic -> StaticExp t2
  -- ...
  BinEq :: StaticExp Dynamic -> StaticExp Dynamic -> StaticExp Bool
  Var   :: String -> StaticExp Dynamic
  To    :: Typeable t => StaticExp t -> StaticExp Dynamic
  From :: Typeable t => StaticExp Dynamic -> StaticExp t
```

Several uninteresting constructors are omitted for brevity. Regrettably, the types of variables and function arguments are dynamic. Furthermore, equality is done on dynamic values, i.e. in all cases, values are cast to dynamics and compared using *dynEq* above. This approach is simpler to implement but limits the amount checks we can omit (This will change in the next section). Currently there is no working implementation of lists for *StaticLang*.

The evaluator for this language is also straight forward: a dynamic enviorment is carried through the evaluation and expressions return the type specified by their constructor:

```
staticEval :: Env -> StaticExp t -> t
```

In the next section we will see several fatal short commings of this target language, forcing me to eventually abandon this implementation.

A compiler from *Lang* to *staticLang*

We will implement a compilation function from *Lang* to *staticLang*. At every step, we will pass type information down to the subexpressions. This will allow us to omit several classes of dynamic checks. The compilation function looks as follows for the lambda case:

```
compile :: TypeRep t -> Exp -> StaticExp t
compile tr (Lam str e) =
  -- Ensure that we expect a function from our contex.
  case arrowUnify tr of
    Just (ArrowRep t1 t2) ->
    -- Our first argument must be a Dynamic. For now...
      case eqT t1 trDynamic of
        Just Refl -> Lam str (compile t2 e) -- Type of exp must be of t2.
       Nothing   -> error "Lam: First argument of arrow should be dynamic."
  -- Not a function, maybe a dynamic type? cast.
    Nothing -> case eqT tr trDynamic of
        Just Refl -> To (Lam str (compile trDynamic e))
        Nothing   -> error "Lam: Type is not arrow or dynamic."
compile tr (f :@ e) =
  let trArrow = mkTyApp (mkTyApp (typeRep :: TypeRep (->)) trDynamic) tr in
     (compile trArrow f) :@  (compile trDynamic e)
```

The application case knows what the return type of the function should be, but has no information about the argument. So we infer an arrow from t1 to dynamic. For the lambda case, first we ensure the context expects an arrow type. If our *typeRep* is from t1 → t2 we compile the body of the lambda expecting it to be of type t2. If we do not have an arrow type, it may be the case we expect a dynamic type. So we simply cast to dynamic. Otherwise we know there is a type error. We have no information about the type of the argument so we simply ensure it is dynamic.

Similarly, for a binary expression we can know the types of the subexpressions based on the operator, e.g. (+) always expects both arguments to be integers. We can match based on the operator to infer the correct type:

```
compile tr (Bin op e1 e2) =
  if intIntOperands op then
    checkTr tr $ BinInt (opToInt op) e1I' e2I' else
  if intBoolOperands op then
    checkTr tr $ BinIntBool (opIntToBool op) e1I' e2I' else
  if boolOperands op then
    checkTr tr $ BinBool (opToBool op) (compile trBool e1) (compile trBool e2) else
    error $ show op ++ ": Unimplemented Bin operator."
  where e1I' = compile trInte e1
        e2I' = compile trInte e2
```

Where *intIntOperands* returns true if the operator takes integers arguments and returns an integer, and so forth. Not only does the compilation step omit unnecessary casts, we are also able to type check the expressions and report obviously wrong code.

As we saw earlier equality requires both subexpressions to be of dynamic type. Why? Consider the compiler case for equality:

```
compile tr (Bin Eq e1 e2) = checkTr tr $
  BinEq (compile trDynamic e1) $ compile trDynamic e2
```

We know an equality comparison should return a bool. This is checked through our call to *checkTr* but what should the type of the subexpressions be? It could be any of our implemented types. With the current implementation we have no way of knowing, so we assume dynamic types.

Currently we pass information down the AST. In both the lambda case and the equality case we could extend the compiler to return type information up the AST. So when no type information is known, we would work our way down the subexpressions, infer the type, and propagate this information up the tree. Unfortunately, I did not have time to try this functionality.

In the equality case all that happened is we added a few unnecessary casts. In other instances this actually leads to incorrect compilation. Consider the factorial function implemented using the y-combinator. When compiling factorial we expect the return type to be an integer. So we call *compile trInt factorial*. The top level node of this AST is a function application (the y-combinator applied to factorial), so the right hand side is expected to have type Dynamic. We actually wanted this side to have type integer, but this information has been lost after the the function application as can be seen in this pretty printed version of compiling factorial to *StaticLang*:

```
(yComb (to (\ f. to (\ n.
             if n == to 0
               to 1
               to (from n * (from f to (from n - 1)))))))
```

In this example the calls to *to* in the branches of the if expression are wrong and cause a runtime error in the static evaluator. Therefore it is necessary to track argument and variable type information in our static language.

A nameless representation static language

The current iteration of our static language uses De Brujin indices to implement a nameless representation of bound variables. This is where the really fun and interesting part begins. Consider the following code:

```
data Index l a where
  Z :: Index (a ‘: b) a
  S :: Index b a -> Index (c : b) a
-- an environment (or heterogeneous list)
data HList a where
  HNil :: HList ‘[]
  HCons :: a -> HList b -> HList (a ‘: b)
```

HList is our environment carrying type information as well as values. *Index* is a type representing De Brujin indices for our variables bound by lambdas. The type variable *l* keeps track of types as a list and *a* is the type of the expression. This the point where I am unsure about some of the details of the implementation. I believe the “quote” followed by a “colon” represent datatype promotions [2]. I mostly get what is happening but some of the subtleties escape me. We can now implement a nameless lambda calculus:

```
data Lc l a where
  Var :: Index l a -> Lc l a
  Lam :: Lc (a : l) b -> Lc l (a -> b)
  App :: Lc l (a -> b) -> Lc l a -> Lc l b
```

Variables are simply their De Brujin index. Lambdas take a term and return a term from *a* to *b*. Applications expect a function and a term of the same type as the argument to the function, and return a term of type *b*. This code amazes me, it works beautifully but I don’t think I have convinced myself why. I build off this to create our desired static language called *LangSnl* (lang static nameless):

```
data LangSnl l a where
  Lam :: LangSnl (a : l) b -> LangSnl l (a -> b)
  (:@) :: LangSnl l (a -> b) -> LangSnl l a -> LangSnl l b
  Bin ::  BinT a b c -> LangSnl l a -> LangSnl l b -> LangSnl l c
  Uni :: UniT a b -> LangSnl l b
  Lit :: a -> LangSnl l a
  Var :: Index l a -> LangSnl l a
  If  :: LangSnl l Bool -> LangSnl l a -> LangSnl l a -> LangSnl l a
  From :: Typeable a => LangSnl l Dynamic -> LangSnl l a
  To :: Typeable a => LangSnl l a -> LangSnl l Dynamic

-- Types for our binary expressions.
data BinT a b c where
  Plus   :: BinT Integer Integer Integer
  And    :: BinT Bool Bool Bool
  Or     :: BinT Bool Bool Bool
  EqBool :: BinT Bool Bool Bool
  -- ...
```

This definition is surprisingly straight forward. Moreover, it seems to works great! Haskell's type inference is quite amazing as it's able to get the types for these expressions. Below are several examples of the type inference by asking *ghci*:

```
-- (fun x. (+ x 3))
(Lam (Bin Plus (Var Z) 3)) :: LangSnl l (Integer -> Integer)
-- (fun x. (fun y. (fun z. (if z y x))))
(Lam (Lam (Lam (If (Var Z) (Var (S Z)) (Var (S (S Z)))))))
:: LangSnl l (a -> a -> Bool -> a)
-- (fun x. (+ (from x) 3))
Lam (Bin Plus (From (Var Z)) 3) :: LangSnl l (Dynamic -> Integer)
-- (fun f. (f "hello"))
(Lam ((Var Z) :@ (Lit "hello"))) :: LangSnl l (([Char] -> b) -> b)
```

Up to implementing *LangSnl* I was mostly wrestling the type checker. These implementations were (mostly) intuitive and the types really worked to help me. The evaluator is far simpler than the previous versions:

```
eval :: HList l -> LangSnl l a -> a
eval env (Lam e) = \y -> eval (HCons y env) e
eval env (e1 :@ e2) = (eval env e1) (eval env e2)
eval env (Lit a) = a
eval env (Bin op e1 e2) = (mapOp op) (eval env e1) (eval env e2)
eval env (Uni Not e) = ~(eval env e2)
eval env (Var x) = look env x
eval env (If cond e1 e2) = if (eval env cond) then (eval env e1) else (eval env e2)
eval env (From e) = gFromDynamic (eval env e)
eval env (To e) = toDynamic (eval env e)

look :: HList l -> Index l a -> a
look (HCons x _) Z = x
look (HCons _ y) (S n) = look y n
```

This is the entire evaluator. *look* simply recurses down the list until and returns the value based on the given index. I think it cannot go out of bounds as this is enforced by the types, woah. *mapOp* maps the binary operators to their function:

```
mapOp :: BinT a b c -> a -> b -> c
mapOp Plus  = (+)
mapOp EqInt = (==)
-- ...
```

The y-combinator requires us to use our *To* and *From* facilities for dynamic types. However, no matter how I wrote it I wasn't able to the get y-combinator to type check. This implementation gives us an "occurs check: cannot construct the infinite type a2$\sim$ a2$\rightarrow$a1" which makes sense but I'm unsure how to fix it:

```
yComb = Lam (g :@  g)
  where g = Lam ((Var (S Z)) :@ ((Var Z) :@ (Var Z)))
```

Compiling to LangSnl

The definition and evaluator for *LangSnl* worked great (except for the y-combinator) so I was hopeful about using it as a target language for *Lang*. However this did not happen. Starting from scratch I attempted to implement a compiler with the signature

```
compile :: NameEnv -> DL.Exp -> LangSnl l t
```

NameEnv is a map from variable names to integers representing their De Brujin indices. Originally, I left out the *TypeRep t* argument for simplicity but without it the compiler raises a " Couldn't match type 't5' with 't', 't5' is a rigid type variable bound by a pattern with constructor...". It sort of makes sense, but I'm not exactly sure why adding the *TypeRep t* and a call to *checkTr* fixes the problem.

The function mapping integers to their De Brujin indices also failed to work. The first version type checks fine, but the recursive "prime" version gives us an "infinite type" error. Every additional case adds a type variable *a* to the type. In the recursive case I assume Haskell has no way to type such an expression (Is this term typeable in dependently typed languages? If so, what is the type? otherwise, why not?).

```
intToDeBrujin :: Int -> Index (a : a : a : a : b) a
intToDeBrujin 0 = Z
intToDeBrujin 1 = S Z
intToDeBrujin 2 = S (S Z)
intToDeBrujin 3 = S (S (S Z))

-- error:
intToDeBrujin 0 = Z
intToDeBrujin n = S (intToDeBrujin (n - 1))
```

Even after overcoming those problems I was unable to get the lambda case for the compiler working. It would not type through the type checker and I ran out of time. While the fancy type level programming brought many benefits: it's definitely super cool and fun to play with, I don't fully understand the errors being generated or the type checker limitations making it difficult to debug.

Conclusion and future work

I had a lot of fun programming with many GHC extensions I have never used before. I gained a better understanding of what they do and why they are useful. I learned why Haskell is considered a good language to implement languages in, and parsing was a joy. For future work the obvious is to actually get the compiler from *Lang* to *LangSnl* working. Once I'm happy with the omitting as many unnecessary casts as possible, it would be fun to write some CPU intensive programs and benchmark what kind of cost this runtime overhead has.

References

[1] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. *A Reflection on Types.* In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, WadlerFest 2016: A list of successes that can change the world, LNCS, pages 292–317. Springer, 2016.

[2] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. *Giving Haskell a promotion.* In Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI '12). ACM, New York, NY, USA, 53-66. DOI=http://dx.doi.org/10.1145/2103786.2103795