Dynamic Typing in a Statically Typed Language

[M. Abadi, L. Cardelli, B. Pierce, G. Plotkin]

Matthias Berg
Advisor: Andreas Rossberg

Saarland University, Programming Systems Lab / Software Engineering Chair
Seminar: Advanced Functional Programming

Abstract. Dynamic typing can be useful in statically typed languages. We extend the simply typed λ-calculus with dynamic typing and elaborate additional features like polymorphism and subtyping.

1 Introduction

There are situations, when even statically typed languages need to perform dynamic type checks. Examples are the handling of persistent storage or interprocess communication. If a process receives some data from another process, it cannot rely on this data to be of some expected type. The type has to be checked dynamically.

Another example are heterogeneous data structures. For instance if a language supports lists which can contain values of different types at the same time, then prior to the usage of an element of such a list, its type must be checked. This can only be done dynamically.

The function `eval` takes an expression as argument and evaluates it. The type of the result can only be determined dynamically. This is a further example were a statically typed language needs dynamic type checking.

Most of the work we present here is based on the papers [1] and [2], which propose to use a type called `Dynamic` to allow dynamic type checking. Values of this type are constructed by pairing a value with its type. Since such values contain a type, we can check this type dynamically. This inspection is done by a `typecase` construct.

Dynamic values are of type `Dynamic` and they can contain values of any type. Therefore it is easy to construct heterogeneous data structures in a language which supports `Dynamic`. For example a list, which can contain values of different types, is simply a list of dynamic values.

2 λ-Calculus with `dynamic` and `typecase`

We give now a formal definition of an extension of the simply typed λ-calculus, which knows the type `Dynamic` and a `typecase` construct. The syntax is rather simple:

$\tau \in \mathit{Typ} ::= X \mid \tau \to \tau \mid \texttt{Dynamic}$
$e \in \mathit{Exp} ::= x \mid \lambda x{:}\tau \;.\; e \mid e\ e \mid \texttt{dynamic}(e{:}\tau) \mid \texttt{typecase}\ e\ \texttt{of}\ x{:}P\ .\ e\ \texttt{else}\ e$

The differences to the simply typed λ-calculus are the type `Dynamic` and the new expressions with `dynamic` and `typecase`. Values of type `Dynamic` (upper case) are constructed by `dynamic` (lower case). It is used to pair an expression with its type.

The `typecase` construct takes an expression of type `Dynamic` as argument and checks whether its contained type matches the pattern P. So far a pattern is just a type and the check is a simple equality test. If the match succeeds, the expression following the pattern is evaluated, where the variable x is substituted with the value contained in the dynamic value. So `typecase` does not only dynamic type checking, it also gives us access to the value contained inside a dynamic value. If the type check does not succeed, the expression of the `else` branch is evaluated.

For example, the following function checks whether its argument contains a number (Assuming, the language supports numbers). If the check succeeds, the result is the number increased by one, otherwise the result is zero.

$$\lambda e{:}\texttt{Dynamic}\ .\ \texttt{typecase}\ e\ \texttt{of}\ x{:}\texttt{Nat}\ .\ x+1\ \texttt{else}\ 0$$

We now give reduction rules for the two new constructs. The others behave as in the simply typed λ-calculus. The rule for `dynamic` is rather simple. It states that the inner expression should be reduces to a value (Values are λ-expressions and `dynamic`-expressions, which contain a value):

$$\frac{e \Rightarrow v}{\texttt{dynamic}(e{:}\tau) \Rightarrow \texttt{dynamic}(v{:}\tau)}$$

The `typecase` construct requires two rules, since the pattern matching can succeed or fail. First, the expression following the `typecase` must reduce to a `dynamic`-value. Its contained type τ is matched with pattern P (This is the side condition). If the match succeeds ($\tau = P$), we reduce e_2, where the variable x is substituted with v_1, the value inside the `dynamic` expression. Otherwise we reduce e_3.

$$\frac{e_1 \Rightarrow \texttt{dynamic}(v_1{:}\tau) \qquad e_2[x := v_1] \Rightarrow v_2}{\texttt{typecase}\ e_1\ \texttt{of}\ x{:}P\ .\ e_2\ \texttt{else}\ e_3 \Rightarrow v_2}\ \tau = P$$

$$\frac{e_1 \Rightarrow \texttt{dynamic}(v_1{:}\tau) \qquad e_3 \Rightarrow v_3}{\texttt{typecase}\ e_1\ \texttt{of}\ x{:}P\ .\ e_2\ \texttt{else}\ e_3 \Rightarrow v_3}\ \tau \neq P$$

Now that we know the behaviour of `dynamic` and `typecase`, their typing rules should be straightforward. For `dynamic` we require that its inner expression really has the claimed type:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{dynamic}(e{:}\tau) : \texttt{Dynamic}}$$

The first expression in the `typecase` construct must have the type `Dynamic`. Furthermore the last two expression must have the same type τ, the type of the whole construct. Additionally the environment of e_2 is extended with the variable x of type P, since x is substituted with something of this type, if the match succeeds during the reduction.

$$\frac{\Gamma \vdash e_1 : \texttt{Dynamic} \qquad \Gamma, x{:}P \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{typecase}\ e_1\ \texttt{of}\ x{:}P\ .\ e_2\ \texttt{else}\ e_3 : \tau}$$

As an example we now write a function which, given two dynamic values, tries to apply the first to the second:

```
λdf:Dynamic . λdx:Dynamic .
  typecase df of
    f:Nat → Nat .
      typecase dx of
        x:Nat . f(x)
      else 0
  else 0
```

This function checks whether its first argument contains a function mapping numbers to numbers and whether its second argument contains a number. But how can we write such a function, which applies functions of arbitrary types to their arguments? This problem brings us to the subject of the following section.

3 Pattern Variables

The problem mentioned in the last section arises from the fact, that every pattern only matches a single type. If we allow the patterns to contain pattern variables, we will obtain a more expressive `typecase` construct. With pattern variables we can match parts of types. For example the pattern $U \rightarrow V$ with pattern variables U and V matches any functional type. A successful match binds U to the argument type and V to the result type of the function. We can now write a function which applies functions of arbitrary types to their arguments:

```
λdf:Dynamic . λdx:Dynamic .
  typecase df of
    {U, V} f:U → V .
      typecase dx of
        {} x:U . dynamic(f(x):V)
      else dynamic(...)
  else dynamic(...)
```

In front of every pattern we write its patters variables in braces. This is done in order to distinguish them from previously bound variables. For example the first pattern contains the pattern variables U and V. The pattern of the inner `typecase` contains no pattern variables. The U is the variable which was bound in the first pattern. This way we check whether the type of the second argument is equal to the argument type of the function.

Interestingly the result cannot be just $f(x)$. We need to pack it again with a `dynamic` expression. This is because of the requirement, that the `else` branches must have the same type as the matching branch and there is no way to construct something of type V, but we can easily build something of type `Dynamic`.

4 Polymorphism

In the following sections we will discuss some possible extensions of our calculus. It is easy to include polymorphism like in System F [3]. Here polymorphism is modelled by functions which take types as argument. Such functions are written $\lambda X\ .\ e$. For example the polymorphic identity function is written $\lambda X\ .\ \lambda x{:}X\ .\ x$ and has the type $\forall X\ .\ X \to X$. If this function is applied to some type, it reduces to the identity function of this type: $(\lambda X\ .\ \lambda x{:}X\ .\ x)\ [\mathtt{Nat}] \ \Rightarrow\ \lambda x{:}\mathtt{Nat}\ .\ x$

The integration of this scheme into our calculus is straightforward. The following example illustrates the use of a `typecase` which matches a polymorphic function f mapping lists to lists and returns a polymorphic function which, given a type and a list x of values of this type, applies f to the reverse of x:

```
λdf:Dynamic .
 typecase df of
   {} f:∀X . List X → List X .
      λY . λx:List Y . f [Y] (reverse [Y] x)
 else λY . λx:List Y . x
```

5 Higher-Order Pattern Variables

Interestingly our first-order pattern variables are not expressive enough in matching against polymorphic types. There is no pattern which matches any polymorphic function in a suitable way. For example the types $\forall X\ .\ X \to X$ and $\forall X\ .\ \mathtt{List}\ X \to \mathtt{List}\ X$ are incompatible, i. e. there is no non-trivial pattern which matches them both. One might think that the pattern $\forall X\ .\ U \to U$ with pattern variable U does the job, since a successful match can bind U to X or to $\mathtt{List}\ X$. But this causes a scoping problem, since U can be used outside the scope of type variable X. This introduces a new free type variable at runtime.

A solution to this problem are higher-order pattern variables. When such a variable is matched, it is not bound to some type, but to a function mapping types to types. The following pattern matches the two types from above by using a second-order pattern variable F: $\forall X\ .\ F\ X \to F\ X$. A successful match binds F to the identity on types, $\Lambda X\ .\ X$, or the following function: $\Lambda X\ .\ \mathtt{List}\ X$.

Now X is passed to F as an argument, making F independent of X. This solves the scoping problem from above.

6 Subtyping

If we include subtyping in our language, this has some implications on our pattern matching. A type T should match a pattern P, if $T \leq P$, i. e. if T is a subtype of P. For example, since $\texttt{Nat} \leq \texttt{Int}$, the following match should succeed:

```
typecase dynamic(5:Nat) of
  x:Int . ...
else ...
```

Unfortunately, the binding of pattern variables is not an easy task any more. In general there is no unique solution for this problem. For example it is clear, that the type $\texttt{Int} \to \texttt{Nat}$ matches the pattern $U \to U$. The types $\texttt{Int} \to \texttt{Int}$ and $\texttt{Nat} \to \texttt{Nat}$ are both supertypes of $\texttt{Int} \to \texttt{Nat}$ and valid instances of the pattern. But neither of them is a subtype of the other, so there is no reason to prefer one solution to the other.

This problem can be avoided by only allowing linear patterns, i. e. patterns where a pattern variable occurs at most once. But this approach is a bit too restrictive. Another solution is to add subtyping constraints and to perform exact matching. Here the matching works as in the previous sections, where we did not have subtyping, but additionally we check after the matching whether some specified subtyping constraints are met. So first we do an exact pattern matching and check the constraints afterwards. To match anything which is a subtype of `Int`, we write something like this:

```
typecase dynamic(5:Nat) of
  {U ≤ Int} x:U . ...
else ...
```

Here the match binds U to `Nat` and then it is checked that $\texttt{Nat} \leq \texttt{Int}$. The problem from above is avoided, since there is simply no way to exactly match the type $\texttt{Int} \to \texttt{Nat}$ with pattern $U \to U$.

7 Abstract Data Types

Another problem that arises with dynamic types is that the `typecase` construct destroys parametricity, i. e. now the reduction is not independent of types any more. Unfortunately type abstraction relies on parametricity to hide its representation. Dynamically, abstract types are just their representation types, hence `typecase` can be used to expose them. Solutions to this problem include the dynamic generation of new type names to restore type abstraction [4, 5].

8 Example Languages

There are programming languages which realise some of the presented ideas. For example the logic language Mercury knows the type `univ`, which corresponds to our type `Dynamic`. It also includes the predicates `type_to_univ` and `univ_to_type` to convert any value into something of type `univ` and back [6].

Haskell also includes dynamic typing. The GHC knows the type `Dynamic` which is realised via the type class `Typeable` of types with a known representation [7]. This representation is compared with the expected type's representation during the dynamic checks. This form of dynamic typing only works for monomorphic types.

The language Clean has a quite expressive dynamic typing. It includes pattern matching with first-order pattern variables and also deals with polymorphism [8].

The language Alice ML provides dynamic typing through the concept of packages. The constructs `dynamic` and `typecase` correspond to the operations to create and open packages, `pack` and `unpack` [9]. Alice ML supports subtyping which also applies to the type check performed by `unpack`.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.
2. Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.
3. Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analysis, et son application à l'élimination des coupures dans l'analysis et la théorie des types. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symposium*. North-Holland, 1971.
4. Andreas Rossberg. Generativity and dynamic opacity for abstract types. In Dale Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003. ACM Press.
5. Matthias Berg. Polymorphic lambda calculus with dynamic types. Bachelor's thesis, Programming Systems Lab, Saarland University, October 2004.
6. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. *The Mercury language reference manual*. University of Melbourne, http://www.cs.mu.oz.au/mercury, 1996.
7. Simon Marlow, Simon Peyton Jones, and Others. *The Glasgow Haskell Compiler*. University of Glasgow, http://www.haskell.org/ghc/, 2002.
8. Marco Pil. First class file i/o. In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 233–246, London, UK, 1997. Springer-Verlag.
9. Alice Team. *The Alice System*. Programming System Lab, Universität des Saarlandes, http://www.ps.uni-sb.de/alice/, 2005.
10. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.