Type Dynamic - Presentation Notes

Stephen Chang

2/26/2010

Motivation

My presentation is about the type `Dynamic`. Values with type `Dynamic` are pairs that contain some value along with a type tag for that value. Here we have a type rule and an evaluation rule for the creation of `Dynamic` values. They are pretty straightforward. For brevity, I'm not going to write out the entire language, but these `Dynamic` values have been added to the simply typed lambda calculus. If some expression `e` has type τ, then this expression where this `dynamic` constructor is applied to some `e` and τ, has type `Dynamic`. For evaluation, if this expression `e` evaluates to v, then the result of this expression is the pair of v and its type tag τ, and obviously this value also has type `Dynamic`. Again, this is the constructor that I'm going to use to create dynamic values, and this is the notation I'm going to use to represent actual `Dynamic` values. Any questions?

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

$$\frac{e \rightarrow^* v}{(\texttt{dynamic}\ e : \tau) \rightarrow^* \langle \texttt{v}, \tau \rangle_{\texttt{Dyn}}}$$

So, what is this good for? Well, in statically typed languages, there are some programs that cannot be assigned a type at compile time. For example, what type would you assign to this function:

`eval` : `string` $\rightarrow$??

You can't give it a static type, right? How about this one:

`print` : ?? $\rightarrow$ `string`

These programs all need type information at run time and type `Dynamic` is a way of embedding this run time type information in statically typed languages. Here is another example:

`read` : `IO` $\rightarrow$??

where this `IO` is a source of untyped data and can represent a disk, another process, or maybe another computer over the internet.

Untyped data can also come from other languages, for example, when you are dealing with multi-language programs. Another example of an application of type `Dynamic` comes from a paper by Gray, Findler, Flatt, where they use `Dynamic` in an object oriented setting. The contribution of the paper is that it demonstrates fine-grained interoperability between Scheme and Java. The paper presents an example where you have a Scheme server that interacts with Java Servlets. In Java, method invocation is always tied to the static type of an object, so you can't call a method unless it was statically known. In order to work with Scheme, you need dynamic method calls like you have in Smalltalk or Python. So to get this behavior, the authors add a `dynamic` type and objects with this type are not inspected until runtime to see if they implement a method that is called. I'm going to skip the web server example since I am not super familiar with web programming concepts, so here's a silly example:

```
Food getFavoriteFood(dynamic fish, Food[] kinds) {
  if (fish.hasFavorite())
    return new Food(fish.getFavorite());
  else
    return fish.chooseFavorite(kinds);
}
```

In the example, the existence of the fish methods is not checked until run time. The fish object does not even have to have all three methods implemented, and it could be a Scheme object.

Ok, back to the functional world. I've shown you how to create `Dynamic` values, but I have not shown you how to eliminate them. This is done using a `typecase` construct. Here are the type rules and evaluation rules for `typecase`:

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{Dynamic} \\ \forall i, \forall \sigma \in Subst_{\vec{X}_i}\ \Gamma[x_i \leftarrow T_i\sigma] \vdash e_i\sigma : T \\ \Gamma \vdash e_{else} : T\end{array}}{\begin{array}{c}\Gamma \vdash (\texttt{typecase}\ e\ \texttt{of} \\ \ldots(\vec{X}_i)(x_i : T_i)\ e_i \ldots \\ \texttt{else}\ e_{else} \\ \texttt{end}) : T\end{array}}$$

$$\frac{\begin{array}{c}\vdash\ e \Rightarrow (\texttt{dynamic}\ w : T) \\ \forall j < k. match(T, T_j)\text{fails} \\ match(T, T_k) = \sigma \\ \vdash e_k\sigma[x_k \leftarrow w] \Rightarrow v\end{array}}{\begin{array}{c}\vdash (\texttt{typecase}\ e\ \texttt{of} \\ \ldots(\vec{X}_i)(x_i : T_i)\ e_i \ldots \\ \texttt{else}\ e_{else} \\ \texttt{end}) \Rightarrow v\end{array}}$$

$$\frac{\begin{array}{c}\vdash\ e \Rightarrow (\texttt{dynamic}\ w : T)\\ \forall k.match(T, T_k)\text{fails}\\ \vdash e_{else} \Rightarrow v\end{array}}{\begin{array}{c}\vdash (\texttt{typecase}\ e\ \texttt{of}\\ \ldots (\vec{X}_i)(x_i : T_i)\ e_i \ldots\\ \texttt{else}\ e_{else}\\ \texttt{end}) \Rightarrow v\end{array}}$$

`typecase` works by using patterns and pattern matching. The i represents different branches of the case statement. The expression $x : T$ is a pattern and we say that a `Dynamic` value matches the pattern if this type T matches the type inside the `Dynamic` value. When we have a match, this x gets bound to the value inside the `Dynamic`. T does not have to be an exact type. This X with an arrow over it represents type variables that can occur in T, so what this rule is saying is that An example might be more useful here, so here is an example of the print function from before:

```
tostring: Dynamic -> String =
  \dv:Dynamic.
    typecase dv of
      (v:String) (string-append '"' v '"')
      (n:Nat) (natToStr n)
      (X,Y)(f:X -> Y) "<function>"
      (X,Y)(p:X x Y)
        (string-append "<" (tostring (dynamic (fst p):X) ","
                             (tostring (dynamic (snd p):Y) ">")
      (d:Dynamic) (string-append "dynamic" (tostring d)
      else "unknown"
    end
```

Background/History

1. As you can tell from the print example, the type `Dynamic` is essentially an infinite disjoint union, or in terms of HTDP, it's an infinite data definition that can be any possible value.

2. languages with (finite) disjoint union: Algol-68, Pascal (tagged variant record)

3. languages infinite disjoint union: Simula-67's subclass structure (INSPECT statement allows program to determine subclass of a value at run time)

4. languages with dynamic typing in static context: CLU (`any` type/force), Cedar/Mesa (REFANY/TYPECASE), Modula-2+, Modula-3 (from Cedar/Mesa) – these languages wanted to support programming idioms from LISP

5. formalization of language with `Dynamic`: Schaffert and Scheifler (1978) gave formal description and denotational semantics for CLU but did not give a soundness theorem – also required every value to carry type tag at run time

6. ML had some proposals for adding `Dynamic` but ultimately unpublished: Gordon (1980, personal communication), Mycroft (1983, draft)

7. languages with mechanisms for handling persistent data: Amber, Modula-2+ (pickling)

Here is another example, that illustrates that the type variables declared in a pattern are in scope for the entire body of the branch:

```
\df:Dynamic.\de:Dynamic.
  typecase df of
    (X,Y)(f:X -> Y)
      typecase de of
        (e:X) (dynamic (f e):Y)
        else (dynamic "Error":String)
      end
    else (dynamic "Error":String)
  end
```

Theorem (soundness): $\forall e, v, T$, if $\vdash e \Rightarrow v$ and $\vdash e : T$, then $v : T$
Corollary: $\forall e, v, T$ if $\vdash e \Rightarrow v$ and $\vdash e : T$, then $v \neq \texttt{wrong}$ (because `wrong` is not well-typed)
Authors also give denotational semantics – difficulty is assigning meaning to `Dynamic` values – use ideal model of types and Banach Fixed Point theorem from MacQueen, Plotkin, Sethi (1986)
Theorems: Typechecking is sound: If e is well typed, then $[\![e]\!]_\rho \neq \texttt{wrong}$ (for well-behaved ρ) proved via: $\forall \Gamma, e, \rho, T$ (ρ consistent with Γ on e), if $\Gamma \vdash e : T$ then $[\![e]\!]_\rho \in [\![T]\!]$
Evaluation is sound: If $\vdash e \Rightarrow v$, then $[\![e]\!] = [\![v]\!]$

Abadi, et al. (1989/1991) - Dynamic Typing in a Statically Typed Language

Abadi, et al. (1995) - Dynamic Typing in Polymorphic Languages

Explicit Polymorphism

Most statically typed functional languages allow polymorphic types. So in their second paper, Abadi et al. add `Dynamic` values to a language with polymorphism. Here's an example of what you can do with polymorphic types. Here we are trying to match a polymorphic function that can consume any type and produces an integer. Now that we have a polymorphic language, we can abstract over types, that's what this big lambda is. And the result is a polymorphic function that applies f to the argument and squares the result.

```
squarePolyFun =
  λdf:Dynamic
    typecase df of
      (f:∀Z.Z→Int)
        ΛW.λx:W.f[W](x)*f[W](x)
      else ΛW.λx:W.0
```

However, with polymorphism, our type variables are not expressive enough to capture some patterns. For example, if we try to implement the dynamic apply function from before:

```
dynamicApply =
  λdf:Dynamic.λda:Dynamic.
    typecase df of
      {} (f:∀Z.??→??)
        typecase da of
          {W} (a:W)
            dynamic( f[W](a):?? )
```

What types do we give the input and output of f? Remember, we need to be able to match all functions, so something like this: $\Lambda\tau.\lambda\mathtt{x} : \tau \times \tau. \ldots$, and $\Lambda\tau.\lambda\mathtt{x} : \tau \to \tau. \ldots$

We need second-order pattern variables that can be type operators in `typecase`, example:

```
dynamicApply =
  λdf:Dynamic.λda:Dynamic.
    typecase df of
      {F,G} (f:∀Z.F(Z)→G(Z))
        typecase da of
          {W} (a:F(W))
            dynamic( f[W](a):G(W) )
```

if $\mathtt{df} = \mathtt{dynamic(}\ \Lambda\mathtt{Z}.\lambda\mathtt{x} : \mathtt{Z} \times \mathtt{Z}.\, \langle \mathtt{snd}(\mathtt{x}), \mathtt{fst}(\mathtt{x})\rangle : \ldots\ )$
and $\mathtt{da} = \mathtt{dynamic(}\langle 3, 4\rangle : \ldots)$
then $\mathtt{F} = \Lambda\mathtt{X}.\mathtt{X} \times \mathtt{X}$, $\mathtt{G} = \Lambda\mathtt{X}.\mathtt{X} \times \mathtt{X}$, and $\mathtt{W} = \mathtt{Int}$
but if $\mathtt{df} = \mathtt{dynamic(}\ \Lambda\mathtt{Z}.\lambda\mathtt{x} : \mathtt{Z} \to \mathtt{Z}.\mathtt{x} : \ldots\ )$
and $\mathtt{da} = \mathtt{dynamic(}\lambda\mathtt{x} : \mathtt{Int}.\mathtt{x} : \ldots)$
then $\mathtt{F} = \Lambda\mathtt{X}.\mathtt{X} \to \mathtt{X}$, $\mathtt{G} = \Lambda\mathtt{X}.\mathtt{X} \to \mathtt{X}$, and $\mathtt{W} = \mathtt{Int}$

But adding high-order pattern variables causes a problem where matching may not be unique. Authors fix this problem by restricting pattern variables to be second order and this allows them to devise a unique matching alg.

Authors do not give type rules or evaluation rules or prove soundness for language with explicit polymorphism.

In the language just presented, you construct `Dynamic` values in the same way as in the previous paper, by giving the constructor a value and an explicit type tag. But some languages with static type systems don't require any type tags, so the authors also add `Dynamic` to a language with implicit polymorphism.

So what are the differences in a language with implicit polymorphism. The first obvious difference is that the constructor only takes one parameter now, the expression itself. The `Dynamic` value still contains a type tag, but this type is now inferred from the given expression.

This requires a change to our matching algorithm because instead of matching an exact type we must be able to match a more general type to a less general pattern. For example, if I have the pattern `lst` : `int list`, I should be able to match a `Dynamic` that contains the empty list, which has type $\forall\alpha.\alpha$ `list`. And this makes intuitive sense because in the explicit language, I could have created a `Dynamic` with the empty list and given it the type `int list` but in the implicit language, the most general type is always inferred. The authors call this property of matching more general types the tag instantiation property.

But there is a problem when you have tag instantiation and second order pattern variables, because second order pattern variables can depend on universal variables, but tag instantiation requires matching with more general types, so you dont know how many universal variables there will be. Example: Tag $\forall \mathtt{A}.(\mathtt{A} \times \mathtt{A}) \to \mathtt{A}$ matches pattern $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}) \to \mathtt{A})$ with $\mathtt{F} = \Lambda \mathtt{X}.\mathtt{X} \times \mathtt{X}$ but tag $\forall \mathtt{A}, \mathtt{B}.(\mathtt{A} \times \mathtt{B}) \to \mathtt{A}$ should also match the pattern because it is more general, but `F` does not depend on `B` ($\mathtt{F} = \Lambda \mathtt{X}.\mathtt{X} \times ??$).

Solution is to capture variables that appear in tag but not in a pattern in a tuple `P` and have all pattern variables depend on `P`. So pattern $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}) \to \mathtt{A})$ is actually $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}; \mathtt{P}) \to \mathtt{A})$. `P` gets instantiated at run time. For the previously mentioned tag $\forall \mathtt{A}, \mathtt{B}.(\mathtt{A} \times \mathtt{B}) \to \mathtt{A}$, `P` gets instantiated to (`B`). Since arity of `P` is not known at compile type, a special tuple sort must be introduced. (Show type rules?)

Authors give typechecking and evaluation rules for implicit polymorphic language with `Dynamic` but do not prove any theorems.

Implicit Polymorphism

Leroy and Mauny (JFP 1993) Dynamics in ML

1. Contribution is adding `Dynamic` values to ML – two extensions: closed type `Dynamic` values (fully implemented in CAML) and non-closed type (prototyped in CAML).

2. `dynamic` construct takes one parameter – type is inferred

3. no explicit `typecase` construct, instead `Dynamic` elimination is integrated into ML pattern matching – pattern written `dynamic(p : T)`, where `p` = pattern and `T` = type

Closed-type `Dynamic` values in ML

1. only allowed to create `Dynamic` values with closed types

2. so `dynamic(fn x` $\to$ `x)` is legal because the inferred type is $\forall\alpha.\alpha \to \alpha$ but `Dynamic` in `fn x` $\to$ `dynamic x` is illegal because `x` has type α which is free – type of value put into `Dynamic` cannot be determined at compile time (would require run time type information to be passed to all functions, even those that dont create `Dynamic` values because you dont know if nested functions do)

3. allowing `Dynamic` objects to have unclosed types would also break ML parametricity properties – polymorphic fns need to operate uniformly over all input types – ie `map g (f l)` = `f (map g l)` for `f` : $\forall\alpha.\alpha$ `list` $\to$ α `list` but the following example does not have this property:

```
let f = fn l ->
  match (dynamic l) with
    dynamic(m:int list) -> reverse l
  | d -> l
```

4. type tag in `Dynamic` value can match less general pattern – so polymorphic pattern actually matches less things than specific pattern – more general patterns need to appear first in case statements

5. typing rules:
   type scheme $\sigma ::= \forall \alpha_1 \ldots \alpha_n . \tau$
   type env $E : Var \rightarrow \sigma$
   $E \vdash a : \tau \Rightarrow b$ = "expression a has type τ in type env E", b is type-annotated version of a
   $Clos(\tau, V)$ = closure of type τ wrt type vars not in $V = \forall \alpha_1 \ldots \alpha_n . \tau$, where $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau) \backslash V$
   $Clos(\tau, \emptyset)$ = type scheme obtained by generalizing free vars in $FV(\tau)$
   $\vdash p : \tau \Rightarrow E$ = "pattern p has type τ and enriches type environment by E
   $Clos(E, \emptyset)$ = type env obtained by creating type schemes from $\tau \in Rng(E)$
   $$\frac{E \vdash a : \tau \Rightarrow b \qquad FV(\tau) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ a : \texttt{Dynamic} \Rightarrow \texttt{dynamic}(b, Clos(\tau, \emptyset))}$$
   $$\frac{\vdash p : \tau \Rightarrow E}{\vdash \texttt{dynamic}(p : \tau) : \texttt{Dynamic} \Rightarrow Clos(E, \emptyset)}$$

6. evaluation rules:
   $\vdash v < p \rightarrow^* m$ = "matching of value v against pattern p results in m"
   $\tau \leq \sigma$ = type τ is instance of type scheme σ (σ is more general)
   $$\frac{e \vdash b \rightarrow^* v}{e \vdash \texttt{dynamic}(b : \sigma) \rightarrow^* \texttt{dynamic}(v : \sigma)}$$
   $$\frac{\vdash v < p \rightarrow^* e \qquad \tau \leq \sigma}{\vdash \texttt{dynamic}(v : \sigma) < \texttt{dynamic}(p : \tau) \rightarrow^* e}$$

7. Soundness: if $[] \vdash a_0 : \tau_0 \Rightarrow b_0$ for some type τ, then we cannot derive $[] \vdash b_0 \rightarrow^* \texttt{wrong}$

8. authors also show how to modify unification algorithm and discuss other implementation issues

Non-closed-type `Dynamic` values in ML

1. closed-type `Dynamic` values are not enough to match certain cases – `print` fn might want to match `Dynamic` that contains any pair, but a pattern like $\texttt{dynamic}((\texttt{x}, \texttt{y}) : \alpha \times \beta)$ only matches `Dynamic` values whose internal type tag is at least as general as $\forall \alpha \forall \beta . \alpha \times \beta$ and will not match `Dynamic` values where internal type is a pair of specific types.

2. need existentially quantified pattern variables – can have patterns like:
   $\exists \alpha . \exists \beta . \texttt{dynamic}((\texttt{x}, \texttt{y}) : \alpha \times \beta)$ – matches `Dynamic` that contains any pair
   $\exists \alpha . \texttt{dynamic}(\texttt{x} :: \texttt{l} : \alpha\ \texttt{list})$ – matches `Dynamic` that contains any list
   $\exists \alpha . \exists \beta . \texttt{dynamic}(\texttt{f} : \alpha \rightarrow \beta)$ – matches `Dynamic` that contains any fn

3. existential and universal quantifiers can be mixed – semantics depends on order of quantification
   $\forall \alpha . \exists \beta . \texttt{dynamic}(\texttt{f} : \alpha \rightarrow \beta)$ – matches `Dynamic` that contains fn that operates uniformly on input – β depends on α – example would be $\texttt{f} : \forall \alpha . \alpha \rightarrow \alpha\ \texttt{list}$
   $\exists \alpha . \forall \beta . \texttt{dynamic}(\texttt{f} : \alpha \rightarrow \beta)$ – matches `Dynamic` that contains fn that returns β for any β – no such fn!

4. when type variable β is allowed to depend on α, like in example $\forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta)$, typechecker must assume that β ALWAYS depends on α, so β is actually type constructor parameterized by α – otherwise this example will typecheck:
   $\texttt{fn}\ \forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta) \to\ \texttt{f}(1) = \texttt{f}(\texttt{true})$
   even though applying the fn to $\texttt{dynamic}(\texttt{fn x} \to \texttt{x})$ produces a run time type error –
   With restriction, above example has type $\forall\alpha.\alpha \to S_\beta(\alpha)$ so you cannot apply the fn to both 1 and `true` because you will get types $\texttt{S}_\beta(\texttt{int})$ and $\texttt{S}_\beta(\texttt{bool})$ as operands to =

5. typing rules:
   type scheme $\sigma ::= \forall\alpha_1 \ldots \alpha_n.\tau$
   type env $E : Var \to \sigma$
   $E \vdash a : \tau \Rightarrow b$ = "expression a has type τ in type env E", b is type-annotated version of a $Clos(\tau, V)$ = closure of type τ wrt type vars not in $V = \forall\alpha_1 \ldots \alpha_n.\tau$, where $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau) \backslash V$
   $Clos(\tau, \emptyset)$ = type scheme obtained by generalizing free vars in $FV(\tau)$
   quantifier prefixes: $Q ::= \epsilon \mid \forall\alpha.Q \mid \exists\alpha.Q$ (assume vars renamed so same var is not bound twice)
   $BV(Q)$ = set of variables found by prefix Q
   $\bar{\tau}$ = types that do not contain previously mentioned type constructors
   $\theta :$ TypeVar $\to \tau$ = type substitution
   $S :$ TypeVar $\ldots \times Q \to \theta$ is defined as follows:

$$\begin{aligned} S(\alpha_1 \ldots \alpha_n, \epsilon) &= id \\ S(\alpha_1 \ldots \alpha_n, \forall\alpha.Q) &= S(\alpha_1 \ldots \alpha_n \alpha, Q) \\ S(\alpha_1 \ldots \alpha_n, \exists\alpha.Q) &= \{\alpha \mapsto S_\alpha(\alpha_1 \ldots \alpha_n)\} \circ S(\alpha_1 \ldots \alpha_n, Q) \end{aligned}$$

   $\vdash p : \tau \Rightarrow E$ = "pattern p has type τ and enriches type environment by E
   $Clos(E, \emptyset)$ = type env obtained by creating type schemes from $\tau \in Rng(E)$

$$\frac{E \vdash a : \tau \Rightarrow b \qquad FV(\tau) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ a : \texttt{Dynamic} \Rightarrow \texttt{dynamic}(b, Clos(\tau, \emptyset))}$$

$$\frac{FV(\bar{\tau}) \subseteq BV(Q) \qquad Q \vdash p : \bar{\tau} \Rightarrow E \qquad \theta = S(\epsilon, Q)}{Q \vdash \texttt{dynamic}(p : \bar{\tau}) : \texttt{Dynamic} \Rightarrow Clos(\theta(E), \emptyset)}$$

6. evaluation rules:
   $\vdash v < p \to^* m$ = "matching of value v against pattern p results in m"
   $T : \tau \times \text{Env} \to \bar{\tau}$ – instantiates type constructors in τ – defined as:

$$\begin{aligned} T(S_\alpha(\tau_1 \ldots \tau_n), e) &= \bar{\tau}[\alpha_1 \leftarrow T(\tau_1, e), \ldots, \alpha_n \leftarrow T(\tau_n, e)] \text{ if } e(\alpha) = \lambda\alpha_1 \ldots \alpha_n.\bar{\tau} \\ T((\forall\alpha_1 \ldots \alpha_n.\tau), e) &= \forall\alpha_1 \ldots \alpha_n.T(\tau, e) \text{ if } \{\alpha_1 \ldots \alpha_n\} \cap Dom(e) = \emptyset \end{aligned}$$

   e – evaluation environment (can also map type vars to type constructors)
   Γ – set of type equations to be solved

$$\frac{e \vdash b \to^* v}{e \vdash \texttt{dynamic}(b : \sigma) \to^* \texttt{dynamic}(v : T(\sigma, e))}$$

$$\frac{Q \vdash v < p \to^* (e, \Gamma) \qquad \bar{\sigma} = \forall \alpha_1 \ldots \alpha_n . \bar{\tau}' \qquad \{\alpha_1 \ldots \alpha_n\} \cap BV(Q) = \emptyset}{Q \vdash \texttt{dynamic}(v : \bar{\sigma}) < \texttt{dynamic}(p : \bar{\tau}) \to^* (e, \Gamma \cup \{\bar{\tau}' = \bar{\tau}\})}$$

7. no soundness theorem for second extension

8. authors also show how to modify unification algorithm and discuss other implementation issues

Abadi, et al. vs Leroy and Mauny's `Dynamic` language with implicit polymorphism

Abadi, et al.	Leroy and Mauny
explicit `typecase` construct	integrate into ML pattern matching
higher order pattern variables	existential pattern variables
$\forall \alpha . \alpha \to \mathtt{F}[\alpha]$	$\forall \alpha . \exists \beta . \alpha \to \beta$
arbitrary dependencies between pattern variables (more expressive)	mixed quantification only allows linear dependencies between pattern variables
ad-hoc restrictions on pattern variables	simple interpretation in first order logic