Type Dynamic - Presentation Notes

Stephen Chang

2/26/2010

Motivation

My presentation is about the type `Dynamic`. I'm going to tell you all about it in just a little bit, but first I'm going to motivate my theme with a few examples. Pretend we are using your favorite statically-typed functional language. What type should we assign to the `eval` function? Anyone? Oh and I should also say that we want to assign a type such that programs that use this function cannot go wrong, for some standard definition of wrong.

`eval` : `string` → ??

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

`tostring` : ?? → `string`

And here's one more:

`read` : `IO` → ??

where `IO` is a stream. It can be a file stream, or maybe a network socket.

As one final example, data whose type is unknown can also come from other programs, which is the case when you have multi-language programs. And this is the situation that Gray, Findler, and Flatt find themselves in when they try to make Scheme and Java interoperate with each other. I will talk about this paper in more detail at the end of the talk, time permitting.

The point is that in these examples, static typing is not enough. All the examples require type information at run time to ensure type safety. Type `Dynamic` is a way of embedding this run time type information into statically typed languages. This brings me to my first paper, Dynamic Typing in a Statically-Typed Language by Abadi, Cardelli, Pierce, and Plotkin. The paper originally appeared in POPL in 1989 and then an extended version appeared in the Transactions on Prog Lang and Systems (TOPLAS) journal in 1991. In the paper, Abadi and his co-authors introduce `Dynamic` values as pairs that contain some value along with a type tag for that value.

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

Here I've written a type rule and an evaluation rule for this `dynamic` constructor that is used to create `Dynamic` values. They are pretty straightforward. If some expression `e` has type `T`, then this expression where this `dynamic` constructor is applied to some `e` and `T`, has type `Dynamic`. For evaluation, if some expression `e` evaluates to a value `v`, then the result of evaluating the `dynamic` constructor applied to `e` and `T` is a pair containing `v` and its type tag `T`, which I'm going to denote using this angle bracket notation with a subscript. And obviously `Dynamic` values themselves also have type `Dynamic`. Any questions?

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

$$\frac{e \Rightarrow v}{(\texttt{dynamic}\ e : T) \Rightarrow \langle \texttt{v}, \tau \rangle_{\texttt{Dyn}}}$$

Ok, I've shown you how to create `Dynamic` values, and now I'm going to show you how to use them. This is done using a `typecase` construct. Here is the type rule and evaluation rule for `typecase`:

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

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

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

These may look slightly complicated but all you really need to know is that `typecase` is essentially a case statement that uses patterns and pattern matching to determine which branch to evaluate. The i subscript represents different branches of the case statement. Each branch will look something like this. The expression $x : T$ is a pattern and we say that a `Dynamic` value matches the pattern if this type pattern 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 if for any substitution of the declared type variables, the body of the branch has type T, and if all branches have type T, then the type of the entire `typecase` has type T.

The evaluation rule uses the function match which simply returns a substitution for the declared type variables, if there exists one, that allow the pattern type T to match the type tag inside the `Dynamic`. When a match is found, then the value inside the `Dynamic` gets bound to the variable in the pattern, like I mentioned before, and the body gets evaluated, with the appropriate substitutions made. And the result of evaluating the entire `typecase` is the result of evaluating the matching body.

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) v
      (n:Number) (num->str 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
```

There are a couple of things to note here. The first is that if you dont declare any type variables, you can leave out the first part of the pattern. The third and fourth cases are the interesting cases because they use patterns. The third case is matching any function and the fourth case is matching any pair. Notice that the tostring function is recursively called with the contents of the pair, which get repackaged as `Dynamic` values with the appropriate type tag.

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
```

Here is a function that consumes two curried `Dynamic` values and returns another `Dynamic` value. There is a second `typecase` nested inside the first one and you can see that the type variables declared in a pattern remain in scope for the entire body of that branch.

Background/History

I'd like to take a little detour and talk about the history of type `Dynamic`. As you can see from the examples, the type `Dynamic` is essentially an infinite disjoint union, or in htdp terms, an infinite data definition. But Abadi and his co-authors were not the first to come up with this idea.

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

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

So if Cardelli and his co-authors did not invent the `Dynamic` type, then what was their contribution. Well, they were the first to present a formal semantics for a language with `Dynamic`. There were some weak attempts previously.

1. 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

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

What Cardelli and his coauthors did was add `Dynamic` to the simply-typed lambda calculus. I've only shown you the relevant `dynamic` and `typecase` rules but they did present type checking rules and evaluation rules for the entire language. They also proved the soundness of their type system. The theorem as presented in the paper is:

Theorem (soundness): $\forall e, v, T$, if $\vdash e \Rightarrow v$ and $\vdash e : T$, then $v : T$

And like we saw in Paul's presentation, in the language, the authors have a wrong value which they do not assign a type. So a corollary to the above thm is that well typed terms are not wrong.‘

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. (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. So both the second and third paper listed here add `Dynamic` values to a polymorphic language, so I'm going to first present this one, and then this one, and then I'm going to compare the two papers. Here's an example of what you can do with polymorphic types.

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

With polymorphism, you can have types that are universally quantified over some type variable. If you want to write a polymorphic function, then you have to use a type abstraction, represented with a big lambda. Now that we have a polymorphic language, we need to abstract over types, that's what this big lambda is, it allows you to write polymorphic functions. Big lambda represents a function from types to terms. And the result is a polymorphic function that applies f to the argument and squares the result.

However, with polymorphism, the type variables from the previous paper are not expressive enough to capture some patterns. For example, if we try to implement the dynamic apply function from before, except we want it to match a polymorphic function, we run into problems.

```
\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
```

```
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? The example from before doesn't work because X and Y can depend on the quantified variable Z, so we need to capture this dependency somehow. 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 $\texttt{df} = \texttt{dynamic}(\ \Lambda\texttt{Z}.\lambda\texttt{x} : \texttt{Z} \times \texttt{Z}.\ \langle \texttt{snd}(\texttt{x}), \texttt{fst}(\texttt{x}) \rangle : \ldots\ )$
and $\texttt{da} = \texttt{dynamic}(\langle 3, 4 \rangle : \ldots)$
then $\texttt{F} = \Lambda\texttt{X}.\texttt{X} \times \texttt{X}$, $\texttt{G} = \Lambda\texttt{X}.\texttt{X} \times \texttt{X}$, and $\texttt{W} = \texttt{Int}$
but if $\texttt{df} = \texttt{dynamic}(\ \Lambda\texttt{Z}.\lambda\texttt{x} : \texttt{Z} \to \texttt{Z}.\texttt{x} : \ldots\ )$
and $\texttt{da} = \texttt{dynamic}(\lambda\texttt{x} : \texttt{Int}.\texttt{x} : \ldots)$
then $\texttt{F} = \Lambda\texttt{X}.\texttt{X} \to \texttt{X}$, $\texttt{G} = \Lambda\texttt{X}.\texttt{X} \to \texttt{X}$, and $\texttt{W} = \texttt{Int}$

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.

In all the examples given, we've had to write out all types and we had to use the big lambda construct to create polymorphic functions. But with polymorphic values, it can get annoying to have to write all the types out, especially polymorphic functions with the type abstractions. So some languages like ML use type inference, where the types are all implicit. So in the second half of the paper, the authors present add `Dynamic` to a language with implicit polymorphism. Interestingly, the authors do not give type rules or evaluation rules or prove soundness for this language (language with explicit 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.

Another difference I already mentioned, which is that pattern variables must be higher order, so that we can match polymorphic values.

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 $\texttt{lst} : \texttt{int list}$, I should be able to match a `Dynamic` that contains the empty list, which has type $\forall\alpha.\alpha\ \texttt{list}$. 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 $\forall\texttt{A}.(\texttt{A} \times \texttt{A}) \to \texttt{A}$ matches pattern $\{\texttt{F}\}(\texttt{f} : \forall\texttt{A}.\texttt{F}(\texttt{A}) \to \texttt{A})$ with $\texttt{F} = \Lambda\texttt{X}.\texttt{X} \times \texttt{X}$ but tag $\forall\texttt{A},\texttt{B}.(\texttt{A} \times \texttt{B}) \to \texttt{A}$ should also match the pattern because it is more general, but $\texttt{F}$ does not depend on $\texttt{B}$ ($\texttt{F} = \Lambda\texttt{X}.\texttt{X} \times ??$).

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

So the rules for `typecase` require changes to accomodate everything I just mentioned. For the sake of time, I'm not going to write it all out because the authors use quite a bit of type machinery to get it to work. In particular, you dont know how many variables this tuple represents at compile time so the authors need to add a new kind of tuple sort to their type system to make everything type check.

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` $\rightarrow$ `x)` is legal because the inferred type is $\forall\alpha.\alpha \rightarrow \alpha$ but `Dynamic` in `fn x` $\rightarrow$ `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` $\rightarrow \alpha$ `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 $\forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \rightarrow \beta)$, typechecker must assume that β ALWAYS depends on α, so β is actually type constructor parameterized by α – otherwise this example will typecheck:
   $\texttt{fn}\ \forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \rightarrow \beta) \rightarrow\ \texttt{f}(1) = \texttt{f}(\texttt{true})$
   even though applying the fn to $\texttt{dynamic}(\texttt{fn}\ \texttt{x} \rightarrow \texttt{x})$ produces a run time type error –
   With restriction, above example has type $\forall\alpha.\alpha \rightarrow S_\beta(\alpha)$ 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:
$\vdash v < p \Rightarrow m$ = "matching of value v against pattern p results in m"
$T : \tau \times$ 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

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

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

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
$(\mathtt{F},\mathtt{G})(\mathtt{v} : \forall \mathtt{A},\mathtt{B}.\mathtt{T}(\mathtt{A},\mathtt{F}(\mathtt{A}),\mathtt{B},\mathtt{G}(\mathtt{A},\mathtt{B})))$	$\forall\mathtt{A}.\exists\mathtt{F}.\forall\mathtt{B}.\exists\mathtt{G}.(\mathtt{v} : \mathtt{T}(\mathtt{A},\mathtt{F},\mathtt{B},\mathtt{G}))$
$(\mathtt{F},\mathtt{G})(\mathtt{v} : \forall \mathtt{A},\mathtt{B}.\mathtt{T}(\mathtt{A},\mathtt{F}(\mathtt{A}),\mathtt{B},\mathtt{G}(\mathtt{A},\mathtt{B})))$	$\forall\mathtt{A}.\exists\mathtt{F}.\forall\mathtt{B}.\exists\mathtt{G}.(\mathtt{v} : \mathtt{T}(\mathtt{A},\mathtt{S}_\mathtt{F}(\mathtt{A}),\mathtt{B},\mathtt{S}_\mathtt{G}(\mathtt{A},\mathtt{B})))$
$(\mathtt{F},\mathtt{G})(\mathtt{v} : \forall \mathtt{A},\mathtt{B}.\mathtt{T}(\mathtt{A},\mathtt{F}(\mathtt{A}),\mathtt{B},\mathtt{G}(\mathtt{B})))$	???
ad-hoc restrictions on pattern variables	simple interpretation in first order logic

Gray,Findler,Flatt - Fine-Grained Interoperability Through Mirrors and Contracts

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.