Type Dynamic - Presentation Notes

Stephen Chang

2/26/2010

Motivation

1. Static typecheckers have many advantages but sometimes dynamic typechecking is unavoidable. Type `Dynamic` is about embedding dynamic typechecking within statically typed languages.

2. What static type to give to `eval` : `string` → ??

3. What static type to give to `print` : ?? → `string` Need to know type at runtime to dispatch proper type-specific print fn

4. What static type to give to `read` : `IO` → ?? Can be from disk, another process, another computer over the network/internet

5. Multi-language programs (use Gray,Findler,Flatt?)

Background/History

1. `Dynamic` is essentially an infinite disjoint union

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 ML had some proposals for adding `Dynamic` but ultimately unpublished: Gordon (1980, personal communication), Mycroft (1983, draft)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

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

1. Introduce new datatype `Dynamic` whose values pairs of value `v` and type tag `T` (so `Dynamic` values carry type information at runtime) – like an infinite disjoint union (sum type) – allow values of different types to be manipulated uniformly

2. (`dynamic e : T`) construct creates `Dynamic` values and `typecase` construct eliminates

3. `typecase` example:

```
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
```

   nested `typecase` example:

```
\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
```

4. Contribution: formal description of language with `Dynamic` values – cbv simply-typed lambda calculus + `dynamic` and `typecase`

5. typecheck and eval rules for `dynamic`:

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

6. typecheck and eval rules for `typecase`:

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

$$\vdash\ e \Rightarrow (\texttt{dynamic}\ w : T)$$

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

7. Theorem: $\forall e, v, T$, 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 $v \neq$ `wrong` (because `wrong` is not well-typed)

8. Authors 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)

9. Typechecking is sound: If e is well typed, then $[\![e]\!]_\rho \neq$ `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

1. Contribution: add explicit polymorphism to language with `Dynamic`– get type abstractions (Λ)

2. example:

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

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

```
dynamicApply =
  λdf:Dynamic.λda:Dynamic.
```

```
typecase fg of
  {F,G} (f:∀Z.F(Z)→G(Z))
    typecase da of
      {W} (a:F(W))
        dynamic( f[W](a):G(W) )
```

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

4. Formally, `Dynamic` values added to System F_ω (simply-typed lambda calculus + type abstractions + type operators)
5. Having higher order pattern variables in `typecase` allows ambiguous matches - example: pattern `F(Int)` matches tag `Int` with either `F` = $\Lambda X.X$ or `F` = $\Lambda X.$`Int`.
6. So patterns must be restricted so that they match any tag uniquely (this property is called definiteness) - restrict language to only second-order polymorphism
7. Authors go not give typing or evaluation rules for explicit polymorphism

Implicit Polymorphism

1. In a language with implicit polymorphism (like ML), some changes need to be made to matching algorithm
2. `dynamic` construct only takes one parameter – no type tag – typechecker infers most general type
3. in `typecase`, matched value must be able to be instantiated differently for different uses. example:

```
foo = λdf.
  typecase df of
    (f:∀A.(A→A)→(A→A)⟨f add1, f not⟩
  else ...
```

4. languages with type inference always infer the most general type, but if there were explicit type tags in `Dynamic` values, the programmer could have given a less general type so patterns should always match type tags that are more general than the pattern – called tag instantiation
5. first order pattern variables not expressive enough to match some examples so we add second order pattern variables like before that can be instantiated to type operators. Example where second order pattern variables are needed:

```
applyTwice = λdf.λdxy.
  typecase df of
    {F,F'} (f:F(P)->F'(P))
      typecase dxy of
        {G,H} (⟨x,y⟩:F(G(Q)) × F(H(Q)))
          ⟨f x,f y⟩
        else ...
   else ...
```

6. tag instantiation and second-order pattern variables cause some problems when used together. 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 $\forall \mathtt{A}.(\mathtt{A} \times \mathtt{A}) \rightarrow \mathtt{A}$ matches pattern $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}) \rightarrow \mathtt{A})$ with $\mathtt{F} = \Lambda \mathtt{X}.\mathtt{X} \times \mathtt{X}$ but tag $\forall \mathtt{A}, \mathtt{B}.(\mathtt{A} \times \mathtt{B}) \rightarrow \mathtt{A}$ should also match the pattern because it is more general, but F does not depend on B ($\mathtt{F} = \Lambda \mathtt{X}.\mathtt{X} \times ??$).

7. 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 $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}) \rightarrow \mathtt{A})$ is actually $\{\mathtt{F}\}(\mathtt{f} : \forall \mathtt{A}.\mathtt{F}(\mathtt{A}; \mathtt{P}) \rightarrow \mathtt{A})$. P gets instantiated at run time. For the previously mentioned tag $\forall \mathtt{A}, \mathtt{B}.(\mathtt{A} \times \mathtt{B}) \rightarrow \mathtt{A}$, 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?)

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

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 → x) is legal because the inferred type is $\forall \alpha.\alpha \rightarrow \alpha$ but Dynamic in fn x → 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 $\mathtt{f} : \forall \alpha.\alpha$ list → 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

$\exists\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta)$ – matches `Dynamic` that contains any fn

3. existential and universal quantifiers can be mixed – semantics depends on order of quantification
   $\forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta)$ – matches `Dynamic` that contains fn that operates uniformly on input – β depends on α – example would be $\texttt{f} : \forall\alpha.\alpha \to \alpha\ \texttt{list}$
   $\exists\alpha.\forall\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta)$ – 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:
   $\texttt{fn}\ \forall\alpha.\exists\beta.\texttt{dynamic}(\texttt{f} : \alpha \to \beta) \to\ \texttt{f(1)} = \texttt{f(true)}$
   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:
   $\vdash v < p \rightarrow^* m =$ "matching of value v against pattern p results in m"
   $T : \tau \times \text{Env} \rightarrow \bar{\tau}$ – instantiates type constructors in τ – defined as:

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

   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
$\forall\alpha.\alpha \rightarrow \texttt{F}[\alpha]$	$\forall\alpha.\exists\beta.\alpha \rightarrow \beta$
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

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

References