Nominal and Structural Ad-Hoc Polymorphism

Stephanie Weirich
University of Pennsylvania
Joint work with Geoff Washburn
Ad-hoc polymorphism

Appears in many different forms:
- Overloading/type classes
- Instanceof/dynamic dispatch
- Run-time type analysis
- Generic/polytypic programming

Many distinctions between these forms:
- Compile-time vs. run-time resolution
- Types vs. type operators
- Type information vs. patterns/tags
- Nominal vs. structural
Nominal style

- Poster child: Overloading
  - `eq(x:int, y:int) = (x == y)`
  - `eq(x:bool, y:bool) = if x then y else not(y)`
  - `eq(x: α'β, y: α'β) = eq(x.1,y.1) & eq(x.2,y.2)`

- Don’t have to cover all types
  - Type checker uses `def` to ensure that there is an appropriate instance for each call site.
  - Can’t treat `eq` as first-class function (even with first-class polymorphism.)
Structural style

*Poster child: typecase*

\[ \text{eq : } \forall \alpha. (\alpha' \alpha) \rightarrow \text{bool} \]

\[ \text{eq}[\alpha:T] = \]

\[
\begin{align*}
\text{typecase } \alpha \text{ of} & \\
\text{int} & \lambda(x: \text{int}, y: \text{int}). (x == y) \\
\text{bool} & \lambda(x: \text{bool}, y: \text{bool}). \\
& \quad \text{if } x \text{ then } y \text{ else not}(y) \\
(\beta' \gamma) & \lambda(x: \beta' \gamma, y: \beta' \gamma). \\
& \quad \text{eq}[\beta](x.1, y.1) \& \& \text{eq}[\gamma](x.2, y.2) \\
(\beta \rightarrow \gamma) & \text{error “Can’t compare functions”}
\end{align*}
\]
Nominal vs. Structural

With *user-defined* (branded, generative) types, these two forms are very different.

Nominal style is “open”
- Can cover as many or as few forms of types as we wish.
- New branches can be added later (even in other modules).

Structural style is “closed”
- Must have a case for all forms of types when operation is defined.
- For types that are not in the domain:
  - Compile-time resolution: Compile-time error
  - Run-time resolution: Exceptions/error values
Nominal Style

- Add a new branch
- newtype Age = Age int
eq(x:Age, y:Age) =
  let (Age xi) = x
  let (Age yi) = y
  if xi <18 && yi < 18
  then true else xi == yi

... but, every new type must define new branches for all polytypic ops.
newtype Phone = Phone int
eq(x:Phone,y:Phone) =
  eq (unPhone x, unPhone y )
Structural Style

- Not extensible

- Sometimes language ignores distinction and implicitly coerces
  Polytypic ops available to all types
  
  let x = Age 53
  eq(x,21)
  
  Breaks distinction between Age and int
  Can’t have a special case for Age.

Which style is better?
Best of both worlds

- Idea: Combine both styles in one language, let the user choose.

- A language where we can write polytypic ops that
  - Are first-class (i.e. based on run-time analysis)
  - May have a partial domain (compile-time detection of invalid arguments)
  - May distinguish user-defined types from their definitions
  - May easily convert to underlying type
  - May be extensible (for flexibility)
  - May not be extensible (for closed-world reasoning)
Caveat

This language is not yet ready for programmers!

- Explicit polymorphism.
- Writing polytypic operations is highly idiomatic.

Next step is to design an appropriate source language/elaboration tool.
Key ideas

- **Expressive type isomorphisms**
  - User can easily convert between types
  - Distinction isn’t lost between them

- **Branches in typecase for new types**
  - Typecase does not need to be exhaustive
  - Restrict type polymorphism by a set of labels
  - Only instantiate with types formed from those labels

- **New branches at run time**
  - Label-set polymorphism makes polytypic ops extensible
Type isomorphisms

Syntax: new type l:T = τ in e
- Scope of new label limited to e
- Inside e use in[l] and out[l] to witness the isomorphism

Also labels for type constructors:
new type l’ : T → T = list in e
in[l’] : ∀α. list α → l’ α
out[l’] : ∀α. l’ α → list α
User control of coercions

- Not a type equality.
  - Users control type distinctions made at run-time.

- When specialized branch is unnecessary, make it easy to coerce types
  - When user-defined type is buried inside another data structure.
  - Should be efficient too—no run-time cost!

- Example: Coerce a value of type
  
  ```
  Age 'int to int' int
  ```

  without destructing/rebuilding product
Higher-order coercions

Coerce part of a type

If l is isomorphic to τ’

If \( e : \tau(l) \) then \( \{ e : \tau \}^- \) has type \( \tau(\tau’) \)

If \( e : \tau(\tau’) \) then \( \{ e : \tau \}^+ \) has type \( \tau(l) \)

Example

\( x : (\text{Age’} \ \text{int}) = (\lambda \alpha : T. \alpha’ \ \text{int}) \ \text{Age} \)

\( \{ e : \lambda \alpha : T. \alpha’ \ \text{int} \}^-_{\text{Age}} : (\text{int’} \ \text{int}) \)

A bit more complicated for type constructors.
Operational Semantics

Coercions don’t do anything at runtime, just change the types.

Annotation determines execution, but just pushes coercions around.

Could translate to untyped language w/o coercions.

Reminiscent of colored brackets [GMZ00].
Typecase and new types

If a new name is in scope, can add a branch for it in typecase

\[ \text{eq}[\alpha:T] = \text{typecase } \alpha \text{ of } \]
\[ \text{int } \lambda(x:\text{int}, y:\text{int}). (x == y) \]
\[ \text{Age } \lambda(x:\text{Age}, y:\text{Age}). \]
  \[ \text{let } x_i = \text{out}[\text{Age}] x \]
  \[ \text{let } y_i = \text{out}[\text{Age}] y \]
  \[ \text{if } x_i < 18 \text{ and } y_i < 18 \]
  \[ \text{then true else } x_i == y_i \]

\[ \text{eq}[\text{Age}] (\text{in}[\text{Age}] 17, \text{in}[\text{Age}] 12) = \text{true} \]

\[ \text{eq}[\text{int}] (17, 12) = \text{false} \]
What if there isn’t a branch?

new type l = int in


shouldn’t type check because no branch for l in eq.

Solution: Make type of polytypic functions describe what types they can handle.
Restricted polymorphism

Polymorphic functions restricted by a set of constants.

\[ \forall \alpha: T \mid \{ \text{int}, ', \text{bool}, \text{Age} \} \ldots \]

Can instantiate \( f \) only with types formed by the above constants.

- \( \text{eq } [(\text{int}' \text{bool}) ' \text{Age}] \text{ is ok} \)
- \( \text{eq } [\text{Phone } ' \text{int}] \text{ is not} \)
- \( \text{eq } [\text{int } \rightarrow \text{ bool}] \text{ is not} \)

Kinding judgment approximates this set.
Restricted polymorphism

- Typecase must have a branch for every name that could occur in its argument.

$$\text{eq}[\alpha:T\{\text{int, }\text{', bool, Age}\}]$$

$$\ (x:\alpha,y:\alpha) =$$

  typecase \(\alpha\) of
    int      ) …
    (\beta'\gamma) ) \lambda(x: \beta'\gamma, y: \beta'\gamma).
      eq[\beta](x.1,y.1) && eq[\gamma](x.2,y.2)
    bool     ) …
    Age      ) …

- What about recursive calls for \(\beta\) and \(\gamma\)?
Product branch

Use restricted polymorphism for those variables too.

let L be the set \{Int, ', Bool, Age\}

eq[\alpha : T|L] (x: \alpha, y: \alpha) =

typecase \alpha of

Int  

(\beta : T| L) \ '(\gamma : T| L) ) \ \lambda (x: \beta' \gamma, y: \beta' \gamma).

eq[\beta](x.1, y.1) \& \& eq[\gamma](x.2, y.2)

Bool  

Age  

Extensibility

How can we make a polytypic operation extensible to new types?

Make branches for typecase first-class

new type l = int in

\[\text{eq}[l] \{ 1 \} \lambda(x:l,y:l). \quad \ldots \} (\text{in}[l] 3, \text{in}[l] 6)\]
First-class maps

New expression forms:

- $\emptyset$: empty map
- $\{1\}e$: singleton map
- $e_1 \cup e_2$: map join

Type of map must describe
- the domain
- the type of each branch
Type of typecase branches

Branches in \( \text{eq} \) follow a pattern:

- **int branch**: \( \text{int} \to \text{bool} \)
  \[
  = (\lambda \alpha. \alpha' \to \text{bool}) \text{int}
  \]

- **bool branch**: \( \text{bool} \to \text{bool} \)
  \[
  = (\lambda \alpha. \alpha' \to \text{bool}) \text{bool}
  \]

- **Age branch**: \( \text{Age} \to \text{bool} \)
  \[
  = (\lambda \alpha. \alpha' \to \text{bool}) \text{Age}
  \]

- **Product branch**:
  \[
  \forall \beta:T|L. \forall \gamma:T|L. (\beta' \gamma)' (\beta' \gamma) \to \text{bool}
  \]
  \[
  = \forall \beta:T|L. \forall \gamma:T|L. ( (\lambda \alpha. \alpha' \to \text{bool}) (\beta' \gamma) )
  \]
Type Operators

In general: type of branch for label $l$ depends on $l$, the kind of $l$, a label set $L$ and some type constructor.

Write as $\tau'\eta \; l : k \mid L \iota$ expanded as:

$$\tau'\eta\tau : T \mid L \iota = \tau' \tau$$

$$\tau'\eta\tau : k_1 \rightarrow k_2 \mid L \iota = \forall \alpha : k_1 \mid L. \tau'\eta\tau \alpha : k_2 \mid L \iota$$

Example:

1. $(\lambda \alpha. \alpha \rightarrow \text{bool}) \eta \; \text{int} : T \mid L \iota = \text{int} \ ' \text{int} \rightarrow \text{bool}$
2. $(\lambda \alpha. \alpha \rightarrow \text{bool}) \eta ' : T \rightarrow T \rightarrow T \mid L \iota$
   $$= \forall \beta : T \mid L. \forall \gamma : T \mid L. (\beta' \gamma) ' (\beta' \gamma) \rightarrow \text{bool}$$
Type of typecase

\[
\text{typecase } \tau \{ \ l_1 \ e_1, \ldots, l_n \ e_n \} \text{ has type } \tau' \\
\tau \text{ when } \\
\tau \text{ has kind } T \text{ using labels from } L \\
\text{for all } l_i \text{ of kind } k_i \text{ in } L, \\
e_i \text{ has type } \tau' \eta_{l_i;k_i} \mid L \tau
\]
First-class maps

Type of map is \( L_1, \tau', L_2 \)
- \( L_1 \) is the domain of the map
- \( \tau' \) and \( L_2 \) are for the type of each branch

Singleton map \{\{1\}\} \ e \} has type \( \{\{1\} \}, \tau', L_2 \) when
- \( 1 \) is a label of kind \( k \) and
- \( e \) has type \( \tau' \eta \ 1 : k | L_2 \)
Not flexible enough

- Must specify the domain of the map.
  \[
  \text{eq: } \forall \alpha : T | L. \{ \text{int} \}, \tau', L \rightarrow (\alpha' \alpha) \rightarrow \text{bool}
  \]

- Can't add branches for new labels
  new type \( l : T = \text{int} \) in
  \[
  \text{eq } [l] \{ 1 \} \lambda(x:l,y:l). \ldots \} (\text{in}[l] 3, \text{in}[l] 6)
  \]

- Need to be able to abstract over maps with any domain --- label set polymorphism
Label-Set polymorphism

- Quantify over label set used in an expression.
- Use label-set variable in map type and type argument restriction.

\[
\begin{align*}
\text{eq } [s:LS] \ [\alpha:T \mid s \cup \{\text{int, bool,}\}] \\
(\text{x : } \hat{s}, \tau', s \cup \{\text{int, bool}\} \hat{s}) &= \\
\text{typecase } \alpha \\
\text{x } \cup \{\text{int }\ldots, \text{bool }\} \ldots \}
\end{align*}
\]

call with:

\[
\text{eq } [\{1\}] \{1\} \ldots \} [1] (\text{in}[1] 3, \text{in}[1] 6)
\]
Fully-reflexive analysis

- New forms of types
  - $\forall \alpha : T \mid L. \alpha \to \alpha$
  - $L, \tau', L'$
  - $\forall s : LS. \tau$

- A calculus is fully-reflexive if it can analyze all types.
  - Need kind-polymorphism for $\forall$

- Label set polymorphism lets us analyze types that contain label sets

- Branches are label-set polymorphic

  typecase ($L, \tau', L'$) {
    $s1, \alpha, s2$ e
  }
Analyzing label sets

- **setcase**
  - Analyzes structure of label sets
  - Determines if the normal form is empty, a single label, or the union of two sets.
  - Requires label and kind polymorphism

- **lindex**
  - Returns the "index" (an integer) of a particular label
  - Lets user distinguish between generated labels
Extensions

- Encode analysis of type constructors
- Default branch for typecase
  - Universal set of all labels
- Record/variant types
  - Label maps instead of label sets
- Type-level type analysis
  - First-class maps at the type level
- Combine with module system?
Conclusion

- Can combine features of nominal analysis and structural analysis in the same system.

- Gives us a new look at the trade-offs between the two systems.

- See paper at http://www.cis.upenn.edu/~sweirich/
Ad-hoc polymorphism

Define operations that can be used for many types of data

Different from
- Subtype polymorphism (Java)
- Parametric polymorphism (ML)

Behavior of operation depends on the type of the data

Example: polymorphic equality

\[ \text{eq : } \forall \alpha. (\alpha', \alpha) \rightarrow \text{bool} \]

Call those operations *polytypic*
User-defined types

- Application-specific types aid software development
  - A PhoneNumber is different than an Age even though both are integers.
  - Type checker distinguishes between them at compile time

Examples:
- class names in Java
- newtypes in Haskell
- generative datatypes in ML
User-defined types

Like Haskell newtypes, ML datatypes
Define new type name
new type Age = int

Type isomorphism not equality---
Coercion functions
  in[Age] : int → Age
  out[Age] : Age → int

Type checker enforces distinction.
  (in[Age] 29) + 1
Operational Semantics

Higher-order coercions

\{i: \lambda \alpha. \text{int}\}^+_1 \otimes i
\{(v_1,v_2): \lambda \alpha. \tau_1' \tau_2\}^+_1 \otimes
(\{v_1: \lambda \alpha. \tau_1\}^+_1, \{v_2: \lambda \alpha. \tau_2\}^+_1)
\{(\lambda x: \tau. e): \lambda \alpha. \tau_1 \rightarrow \tau_2\}^+_1
\otimes \lambda x: \tau_1[l/\alpha]. \{ e[\{x: \lambda \alpha. \tau_1\}^{-1}/x]: \lambda \alpha. \tau_2\}^+_1
\{v: \lambda \alpha. \alpha\}^+_1 \otimes \text{in}[l] \; v
Universal set

\( \textbf{Set} \, \top \) is set of all labels

\[ f[\alpha:T|\top] \ldots \]

- \( f \) can be applied to any type
- \( \text{eq}[\alpha] \) doesn't typecheck
- \( \alpha \) cannot be analyzed, because no typecase can cover all branches.
- No type containing \( \alpha \) can be analyzed either.
- Cheap way to add parametric polymorphism.
Other map formers

- Empty map $\emptyset$ has type $\emptyset$, $\tau'$, $L$
- For arbitrary $\tau'$, $L$

- $e_1 \cup e_2$ has type $L_1 \cup L_2$, $\tau'$, $L$
- When
- $e_1$ has type $L_1$, $\tau'$, $L$
- $e_2$ has type $L_2$, $\tau'$, $L$
Union is non-disjoint

\[
f[\alpha : T | \{\text{int}\}] \\
(x : \{\text{int}\}, \tau', L) = \text{typecase} \alpha (\{\text{int} \} 2 \cup x)
\]

- **Can overwrite existing mappings:**
  \[
f[\text{int}] \{\text{int} \} 4 = 4
\]

- **Reversing order prevents overwrite:**
  \[
  \text{typecase} \alpha (x \cup \{\text{int} \} 2)
  \]
Open vs. closed polytypic ops

Closed version of eq has type
\[ \forall \alpha : T|L. \, \tau' \alpha \]
where \( L = \{ \text{int}, \text{bool}, ', \text{Age} \} \)
\[ \tau' = \lambda \alpha. \, (\alpha' \alpha) \to \text{bool} \]

Open version of eq has type
\[ \forall s:LS. \, \forall \alpha : T|s \cup L. \, ?s, \tau', s \cup L ? \to \tau' \alpha \]

What is the difference?
Open ops calling other ops

important : ∀s:LS. ∀α:T|s. ![s], λβ.β → bool, s![α] → α → bool

print[s:LS][α:T|s]
(mp : ![s], (λβ. β → string), s![α] mi : ![s], λβ. β → bool,s![α] =
typecase α of
(β:T|s ' γ:T|s ) )
λ(x:β ' γ).
write("(";
if important[s][β] mi (x.1)
then print[s][β] (x.1) (mp,mi)
else write("…");
write(“,”);
if important[s][γ] mi (x.2) then …