Unifying Nominal and Structural Ad-Hoc Polymorphism

Stephanie Weirich
University of Pennsylvania
Joint work with Geoff Washburn
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 \textit{polytypic}
Ad hoc polymorphism

Appears in many different forms:
- Overloading
- Haskell 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
- Nominal vs. structural
Nominal style

- Poster child: overloading

\[
\begin{align*}
\text{eq}(x:\text{int}, \ y:\text{int}) &= (x == y) \\
\text{eq}(x:\text{bool}, \ y:\text{bool}) &= \\
&\quad \text{if } x \text{ then } y \text{ else not}(y) \\
\text{eq}(x: \alpha'\beta, \ y: \alpha'\beta) &= \\
&\quad \text{eq}(x.1,y.1) \&\& \text{eq}(x.2,y.2)
\end{align*}
\]

- 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 \text{eq} as first-class function.
Use a “case” term to branch on the structure of types

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

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

\text{typecase } \alpha \text{ of }

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

- **Nominal style is “open”**
  - Can have as many or as few branches as we wish.
  - New branches can be added later (even other modules).

- **Structural style is “closed”**
  - Must have a case for all forms of types when operation is defined.
  - Use exceptions/error values for types that are not in the domain.

With **user-defined** (aka application-specific) types, these two forms are radically different.
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
Modeling user-defined types

- Define new label (a type isomorphism)
  new type Age = int

- Coercion functions
  in[Age] : int → Age
  out[Age] : Age → int

- Type checker enforces distinction.
  \[ (in[Age] 29) + 1 \]
With polytypism?

Nominal style -- add a new branch

\[
eq(x:\text{Age}, y:\text{Age}) =
\]

\[
\begin{align*}
\quad & \text{let } x_i = \text{out}[\text{Age}] x \\
\quad & \text{let } y_i = \text{out}[\text{Age}] y \\
\quad & \text{if } x_i > 30 \land y_i > 30 \\
\quad & \quad \text{then true else } x_i = y_i
\end{align*}
\]

... but, each new type must define new branches for all polytypic ops.

\[
\text{newtype Phone} = \text{int}
\]

\[
\text{eq}(x:\text{Phone}, y:\text{Phone}) =
\]

\[
\text{eq (out[Phone] x, out[Phone] y )}
\]
Structural Style

- Not extensible
- But, polytypic ops already available to all types
  - Language implicitly coerces
    - let x = in[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
  - Have a partial domain (static detection of wrong arguments)
  - Are first-class (based on typecase)
  - 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 humans!
  
  Explicit polymorphism.
  
  Writing polytypic operations is highly idiomatic.

- Next step is to design an appropriate source language/elaboration tool.
Type isomorphisms

**Syntax:** new type $l : T = \tau$ in $e$
- Scope of new label limited to $e$
- Inside $e$ use $\text{in}[l]$ and $\text{out}[l]$ to witness the isomorphism

**Also labels for type operators:**
- new type $l' : T \to T = \text{list}$ in $e$
- $\text{in}[l'] : \forall \alpha. \text{list } \alpha \to l' \alpha$
- $\text{out}[l'] : \forall \alpha. l' \alpha \to \text{list } \alpha$
User control of coercions

- Don’t automatically coerce types.
  - User may want to use a specialized branch.
- When specialized branch is unnecessary, make it easy to coerce types
  - And efficient too!
  - Especially when user-defined type is buried inside another data structure.
- 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 \( \tau' \)
  - If \( e : \tau(l) \) then \( \{ e : \tau \}^{-1}_1 \) has type \( \tau(\tau') \)
  - If \( e : \tau(\tau') \) then \( \{ e : \tau \}^{+}_1 \) has type \( \tau(l) \)
- Example

\[
x : (\text{Age}'\text{int}) = (\lambda \alpha : \text{T}.\alpha'\text{int}) \text{Age}
\]

\[
\{e : \lambda \alpha : \text{T}.\alpha'\text{int}\}^{-\text{Age}}_1 : (\text{int}'\text{int})
\]

- A bit more complicated for type operators
Operational Semantics

- Coercions don’t do anything at runtime, just change the types.
- Annotation determines execution.

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

- Reminiscent of colored brackets [GMZ00]
Special cases for new types

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

\[ \text{eq}[\alpha:T] = \text{typecase } \alpha \text{ of} \]
\[ \text{int } \lambda(x:\text{int}, y:\text{int}). (x==y) \]
\[ \text{Age } \lambda(x:\text{Age}, y:\text{Age}). \]
\[ \text{let } x_i = \text{out}[\text{Age}] x \]
\[ \text{let } y_i = \text{out}[\text{Age}] y \]
\[ \text{if } x_i > 30 \text{ && } y_i > 30 \]
\[ \text{then true else } x_i == y_i \]

\[ \text{eq}[\text{Age}] (\text{in}[\text{Age}] 31, \text{in}[\text{Age}] 45) = \text{true} \]

\[ \text{eq}[\text{int}] (31, 45) = \text{false} \]
What if there isn’t a branch?

new type l = int in
  eq[l] (in[l] 3, in[l] 6)
shouldn’t type check because no branch
for 1 in eq.

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

Polymorphic functions restricted by a set of labels.

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

\[ \text{eq } [\alpha : T | \{ \text{int,}', \text{bool, Age} \}] = \ldots \]

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

- eq [(int’bool) ’Age] is ok
- eq [Phone ’ int] is not
- eq [int \rightarrow bool] is not
Restricted polymorphism

# Typecase must have a branch for every label that could occur in its argument.

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

\[(x:\alpha,y:\alpha) = \]

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

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

# Use restricted polymorphism for those variables too.

let L = {Int, ', Bool, Age}

eq[α:T|L] (x:α,y:α) =

typecase α of

  Int

  (β:T|L) 'γ:T|L)

    λ(x: β', y: β').

    eq[β](x.1,y.1) && eq[γ](x.2,y.2)

  Bool

  Age

Universal set

Set \( \top \) is set of all labels

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

- \( f \) can be applied to any type
- \( eq[\alpha] \) 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.
How can we make a polytypic operation extensible to new types?

Make branches for typecase first-class

new type l = int in

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 branches to typecase.
Type of typecase branches

Branches in `eq` follow a pattern:

- **int branch**: `int' int → bool
  
  \( = (\lambda \alpha. \alpha' \alpha → bool) \text{ int} \)

- **bool branch**: `bool' bool → bool
  
  \( = (\lambda \alpha. \alpha' \alpha → bool) \text{ bool} \)

- **Age branch**: `Age' Age → bool
  
  \( = (\lambda \alpha. \alpha' \alpha → bool) \text{ Age} \)

- **Product branch**:
  
  \( \forall \beta: T|L. \forall \gamma: T|L. (\beta' \gamma)' (\beta' \gamma) → bool \)

  \( = \forall \beta: T|L. \forall \gamma: T|L. ( (\lambda \alpha. \alpha' \alpha → bool) (\beta' \gamma) ) \)
Type Operators

In general: type of branch for label 1 with kind \( k \) is \( \tau' \eta \, 1 : k \mid L \iota \)

\[
(\lambda \alpha. \alpha \, ' \alpha \to \text{bool}) \eta \text{ int} : T \mid L \iota = \text{int } ' \text{int} \to \text{bool}
\]

\[
(\lambda \alpha. \alpha \, ' \alpha \to \text{bool}) \eta ' : T \to T \to T \mid L \iota
\]

\[= \forall \beta : T \mid L. \forall \gamma : T \mid L. (\beta' \gamma) ' (\beta' \gamma) \to \text{bool} \]

Expand this type:

\[
\tau' \eta \tau : T \mid L \iota = \tau' \tau
\]

\[
\tau' \eta \tau : k_1 \to k_2 \mid L \iota = \forall \alpha : k_1 \mid L. \tau' \eta \tau \alpha : k_2 \mid L \iota
\]
Type of typecase

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

Type of first-class label map must include

- What labels are in domain
- What \( \tau' \) and \( L \) are for the branches
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 \{ l \} e \} has type $\{l\}, \tau', L_2$
  - \{ l \} e \} has type $\{l\}, \tau', L_2$
    - l is a label of kind k and
    - e has type $\tau' \eta l : k | L_2 \iota$
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

\[
\begin{align*}
    f \left[ \alpha : T \mid \{ \text{int} \} \right] \\
    (x : \{ \text{int} \}, \tau', L) = \\
    \text{typecase } \alpha \left( \{ \text{int} \} \ 2 \right) \cup x
\end{align*}
\]

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

- Reversing order prevents overwrite:
  \[ \text{typecase } \alpha \left( x \cup \{ \text{int} \} \ 2 \right) \)
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

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.

\[
eq [s:LS] [\alpha:T | s \cup \{\text{int}, \text{bool},\}] \\
(x : ?s, \tau', s \cup \{\text{int}, \text{bool}\}?) = \\
typecase \alpha \\
x \cup \{\text{int}\}...\text{, bool} ) ... }
\]

call with:

\[
eq [{\{1\}}] {1} ... } [1] (\text{in}[1] 3, \text{in}[1] 6)
\]
Open vs. closed polytypic ops

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

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

What is the difference?
Open ops calling other ops

important : \forall s:LS. \forall \alpha:T|s. \forall \beta. \beta \rightarrow \text{bool}, s\beta \rightarrow \alpha \rightarrow \text{bool}

print[s:LS][\alpha:T|s]
(mp : s, (\lambda \beta. \beta \rightarrow \text{string}), s\beta; mi : s, \lambda \beta. \beta \rightarrow \text{bool}, s\beta; =
typecase \alpha \text{ of}
(\beta:T|s \ ' \gamma:T|s ) )
\lambda(x:\beta \ ' \gamma).
write(“(“);
if important[s][\beta] mi (x.1) then print[s][\beta] (x.1) (mp,mi)
else write(“…”);
write(“,”);
if important[s][\gamma] mi (x.2) then …
Fully-reflexive analysis

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

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

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

- Branches are label-set polymorphic
  
  ```
  typecase (L, \tau', L')
  \{ s1, \alpha, s2 \}
  ```
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

- Default branch for typecase
  - Destroys parametricity
- Record/variant types
  - Label maps instead of label sets
- Type-level type analysis
  - First-class maps at the type level
- Combine with module system/distributed calculus
Key ideas (Summary)

- 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
  - Ensures typecase has a branch for each arg

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

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