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: $\alpha'\beta$, y: $\alpha'\beta$) =

  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

$eq : \forall\alpha.\ (\alpha'\alpha) \rightarrow bool$

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

typecase α of

int) λ(x:int, y:int). (x == y)

bool) λ(x:bool,y:bool).

if x then y else not(y)

($\beta'\gamma$)) λ(x: $\beta'\gamma$, y: $\beta'\gamma$).

eq[β](x.1,y.1) && eq[γ](x.2,y.2)

($\beta \rightarrow \gamma$)) error “Can’t compare functions”

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 $l{:}T = \tau$ 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 $l' : T \rightarrow T$ = list in e

in[l'] $: \forall\alpha.\ \text{list}\ \alpha \rightarrow l'\ \alpha$

out[l'] $: \forall\alpha.\ l'\ \alpha \rightarrow \text{list}\ \alpha$

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 $\{ e : \tau \}^{-}_{l}$ has type $\tau(\tau')$
  - If $e : \tau(\tau')$ then $\{ e : \tau \}^{+}_{l}$ has type $\tau(l)$
- *Example*

  $x : (\mathrm{Age}\ '\ \mathrm{int}) = (\lambda\alpha{:}T.\alpha\ '\mathrm{int})\ \mathrm{Age}$

  $\{e{:}\ \lambda\alpha{:}T.\alpha'\mathrm{int}\}^{-}_{\mathrm{Age}} : (\mathrm{int}\ '\ \mathrm{int})$
- 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

```
eq[α:T] = typecase α of
   int   ) λ(x:int,y:int). (x==y)
   Age ) λ(x:Age,y:Age).
              let xi = out[Age] x
              let yi = out[Age] y
                 if xi < 18 && yi < 18
                 then true else xi == yi
```

- eq[Age] (in[Age] 17, in[Age] 12) = true
- eq[int] (17, 12) = 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 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.

  $$eq : \forall\alpha{:}T|\{int,',bool,Age\}.\ \ldots$$

- Can instantiate f only with types formed by the above constants.
  - eq [(int′bool) ′Age] is ok
  - eq [Phone ′ int] is not
  - eq [int → bool] is not
- Kinding judgment approximates this set.

Restricted polymorphism

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

eq[α:T|{int, ′,bool,Age}]
 (x:α,y:α) =
    typecase α of
        int) …
        (β′γ)) λ(x: β′γ, y: β′γ).
                eq[β](x.1,y.1) && eq[γ](x.2,y.2)
        bool) …
        Age) …

- What about recursive calls for β and γ?

Product branch

- Use restricted polymorphism for those variables too.

let L be the set {Int, ′, Bool, Age}

eq[α:T|L] (x:α,y:α) =

    typecase α of

        Int)

        (β:T|L) ′(γ:T|L)) λ(x: β′γ, y: β′γ).

            eq[β](x.1,y.1) && eq[γ](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

eq[l] { l) λ(x:l,y:l). … } (in[l] 3, in[l] 6)

First-class maps

- New expression forms:
  - $\emptyset$ empty map
  - $\{l)e\}$ 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 eq follow a pattern:
  - int branch: $\mathrm{int} \,'\, \mathrm{int} \rightarrow \mathrm{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \mathrm{bool})\ \mathrm{int}$
  - bool branch: $\mathrm{bool} \,'\, \mathrm{bool} \rightarrow \mathrm{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \mathrm{bool})\ \mathrm{bool}$
  - Age branch: $\mathrm{Age} \,'\, \mathrm{Age} \rightarrow \mathrm{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \mathrm{bool})\ \mathrm{Age}$
  - Product branch:
    $\forall\beta{:}\mathrm{T}|\mathrm{L}.\forall\gamma{:}\mathrm{T}|\mathrm{L}.\ (\beta'\gamma) \,'\, (\beta'\gamma) \rightarrow \mathrm{bool}$
    $= \forall\beta{:}\mathrm{T}|\mathrm{L}.\forall\gamma{:}\mathrm{T}|\mathrm{L}.\ (\ (\lambda\alpha.\ \alpha'\alpha \rightarrow \mathrm{bool})\ (\beta'\gamma)\ )$

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:

  $\tau'\eta\tau{:}T \mid L\iota \;= \tau'\ \tau$

  $\tau'\eta\tau{:}k_1 \rightarrow k_2 \mid L\iota = \forall\alpha{:}k_1|L.\ \tau'\eta\tau\ \alpha{:}k_2|L\iota$
- Example:
  - $(\lambda\alpha.\alpha\ '\alpha \rightarrow \text{bool})\ \eta\ \text{int} : T \mid L\ \iota = \text{int}\ '\text{int} \rightarrow \text{bool}$
  - $(\lambda\alpha.\alpha\ '\alpha \rightarrow \text{bool})\ \eta\ ' : T \rightarrow T \rightarrow T \mid L\ \iota$

    $= \forall\beta{:}T|L.\forall\gamma{:}T|L.\ (\beta'\gamma)\ '\ (\beta'\gamma) \rightarrow \text{bool}$

Type of typecase

- typecase τ { l_1) e_1, …, l_n) e_n} has type $\tau' \tau$ when
  - τ has kind T using labels from L
  - for all l_i of kind k_i in L,

    e_i has type $\tau' \eta l_i{:}k_i \mid L\iota$

First-class maps

- Type of map is �L_1, τ', L_2 �
  - L_1 is the domain of the map
  - τ' and L_2 are for the type of each branch

- Singleton map $\{\, l \,)\, e \,\}$ has type �$\{l\}$, τ', L_2 �when
  - l is a label of kind k and
  - e has type $\tau' \eta\, l : k \mid L_2 \iota$

Not flexible enough

- Must specify the domain of the map.
  - eq: ∀α:T|L. �{int}, τ’, L�→ (α ′ α) → bool
- Can’t add branches for new labels

  new type l :T = int in

  eq [l] { l) λ(x:l,y:l). …} (in[l] 3, 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] [α:T | s ∪ {int,bool,}]

  (x : �s, τ’, s ∪ {int,bool}�) =

  typecase α

  x ∪ { int)…, bool) … }

call with:

  eq [{l}] { l) … } [l] (in[l] 3, in[l] 6)

Fully-reflexive analysis

- New forms of types
  - $\forall\alpha$:T|L. $\alpha \rightarrow \alpha$
  - �L, τ’, L’ �
  - $\forall$s:LS. τ
- A calculus is fully-reflexive if it can analyze all types.
  - Need kind-polymorphism for ∀
- Label set polymorphism lets us analyze types that contain label sets
- Branches are label-set polymorphic

typecase (�L, τ’, L’ �) { �s1,α, 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

$\{i{:}\lambda\alpha.\mathrm{int}\}^{+}_{l} \otimes i$

$\{(v_1,v_2){:}\lambda\alpha.\tau_1{}'\tau_2\}^{+}_{l} \otimes (\{v_1{:}\lambda\alpha.\tau_1\}^{+}_{l}, \{v_2{:}\lambda\alpha.\tau_2\}^{+}_{l})$

$\{(\lambda x{:}\tau.e){:}\lambda\alpha.\tau_1 \rightarrow \tau_2\}^{+}_{l}$

$\otimes\ \lambda x{:}\tau_1[l/\alpha].\ \{\ e[\{x{:}\lambda\alpha.\tau_1\}^{-}_{l}/x]{:}\ \lambda\alpha.\tau_2\}^{+}_{l}$

$\{v{:}\lambda\alpha.\alpha\}^{+}_{l} \otimes \mathrm{in}[l]\ v$

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
  - α cannot be analyzed, because no typecase can cover all branches.
  - No type containing α can be analyzed either.
  - Cheap way to add parametric polymorphism.

Other map formers

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

- $e_1 \cup e_2$ has type �$L_1 \cup L_2$, τ', L �when
  - e1 has type �L_1, τ', L �
  - e2 has type �L_2, τ', L �

Union is non-disjoint

$f[\,\alpha : T \mid \{\, int\}\,]$

$\quad (x : \text{�}\{int\}, \tau', \; L \,\text{�}) =$

$\qquad \text{typecase } \alpha \;\; (\{int\ )\ 2\} \cup x\ )$

- Can overwrite existing mappings:
  - $f\,[int]\ \{int\ )\ 4\} = 4$
- Reversing order prevents overwrite:

$\text{typecase } \alpha\ (\ x \cup \{int\ )\ 2\}\ )$

Open vs. closed polytypic ops

- Closed version of eq has type

  $\forall\alpha{:}T|L.\ \tau'\ \alpha$

  where $L = \{\ int, bool, ′, Age\}$

  $\tau' = \lambda\alpha.\ (\alpha\ ′\ \alpha) \rightarrow bool$

- Open version of eq has type

  $\forall s{:}LS.\ \forall\alpha{:}T|s \cup L.$?s, τ', $s \cup L$? $\rightarrow \tau'\ \alpha$

- What is the difference?

Open ops calling other ops

```
important : ∀s:LS. ∀α:T|s. �s, λβ.β → bool, s� → α → bool

print[s:LS][α:T|s]
  (mp : �s, (λβ. β → string), s�, mi : �s, λβ. β → bool,s�) =
    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 …
```