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 \alpha of
     int
           ) \lambda(x:int, y:int). (x == y)
                  ) \lambda(x:bool,y:bool).
     bool
                       if x then y else not(y)
    (\beta'\gamma) ) \lambda(x; \beta'\gamma, y; \beta'\gamma).
                      eq[\beta](x.1,y.1) \&\& eq[\gamma](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, 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

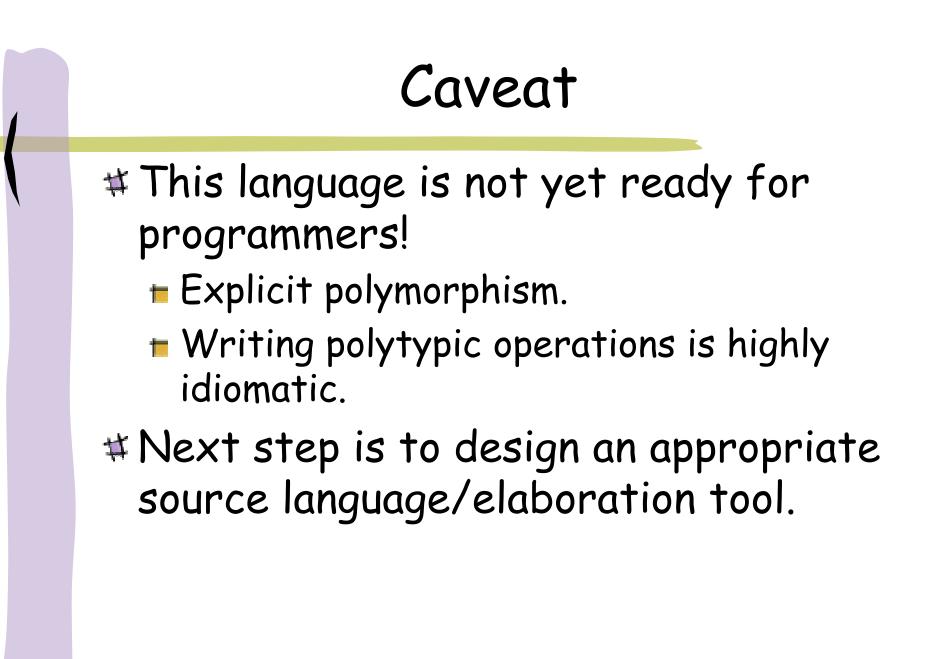
let x = Age 53

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)



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 $1:T = \tau$ in e
 - Scope of new label limited to e
 - Inside e use in[1] and out[1] 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 runtime.
- 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 1 is isomorphic to τ ' **If** $e : \tau(1)$ then $\{e : \tau\}^{-1}$ has type $\tau(\tau')$ **If** $e: \tau(\tau')$ then $\{e:\tau\}^+$ has type $\tau(1)$ # Example x : (Age ' int) = ($\lambda \alpha$:T. α 'int) Age $\{e: \lambda \alpha: T. \alpha' int\}_{Age} : (int ' int)$ # 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[\alpha:T] = typecase \alpha of
    int ) \lambda(x:int,y:int). (x==y)
    Age ) \lambda(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 1 = int in eq[1] (in[1] 3, in[1] 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 constants. eq : $\forall \alpha$: T|{int, ', bool, Age}.... # Can instantiate f only with types formed by the above constants. teq [(int'bool) 'Age] is ok eq [Phone ' int] is not $= eq [int \rightarrow bool] is not$ # 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:\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 firstclass
new type 1 = int in
eq[1] { 1) λ(x:1,y:1). ... } (in[1] 3, in[1] 6)

First-class maps

#New expression forms:

- $\begin{array}{ll} \blacksquare & \varnothing & empty map \\ \blacksquare & \{1)e\} & singleton map \\ \blacksquare & e_1 \cup e_2 & map join \end{array}$
- Type of map must describe
 the domain
 the type of each branch

Type of typecase branches

Branches in eq follow a pattern: **int branch**: int ' int \rightarrow bool = $(\lambda \alpha. \alpha' \alpha \rightarrow bool)$ int **bool branch:** bool ' bool \rightarrow bool = $(\lambda \alpha. \alpha' \alpha \rightarrow bool) bool$ **The Age branch:** Age ' Age \rightarrow bool = $(\lambda \alpha. \alpha' \alpha \rightarrow bool)$ Age Product branch: $\forall \beta: T | L. \forall \gamma: T | L. (\beta' \gamma) ' (\beta' \gamma) \rightarrow bool$ = $\forall \beta: T | L. \forall \gamma: T | L. ((\lambda \alpha, \alpha' \alpha \rightarrow bool) (\beta' \gamma))$

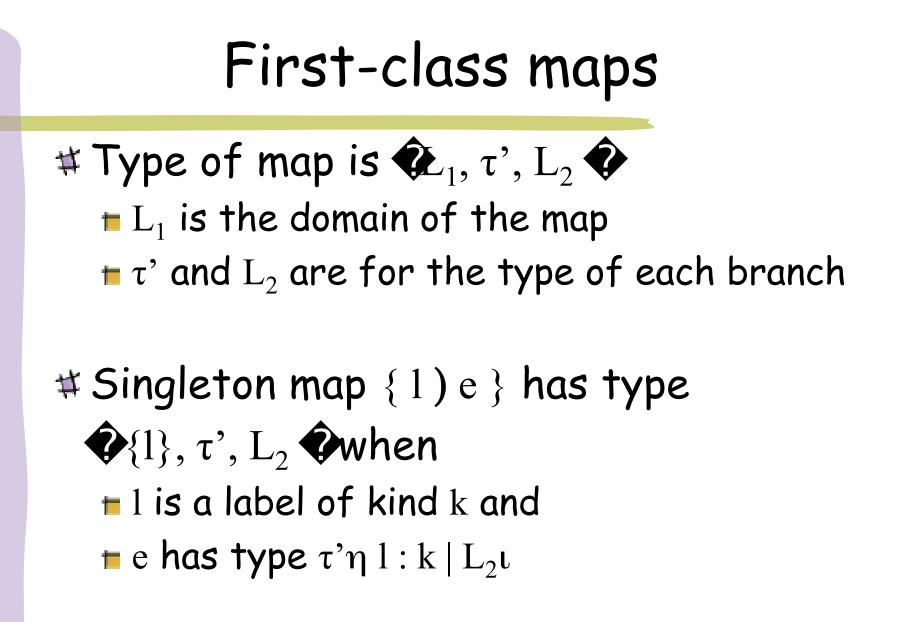
Type Operators

In general: type of branch for label 1 depends on 1, the kind of 1, a label set L and some type constructor. # Write as $\tau'\eta$ 1:k | Li 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: t (λα.α 'α → bool) η int : T | L ι = int 'int → bool **t** (λα.α 'α → bool) η ' : T →T →T | L ι $= \forall \beta: T | L. \forall \gamma: T | L. (\beta' \gamma) ' (\beta' \gamma) \rightarrow bool$

Type of typecase

typecase $\tau \{ l_1 \} e_1, ..., l_n \}$ 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 τ'ηl_i:k_i | Lι



Not flexible enough

★ Must specify the domain of the map.
eq: ∀α:T|L. $f(int), \tau', L f(a' a) \rightarrow bool$ ★ Can't add branches for new labels
new type 1 :T = int in
eq [1] { 1) λ(x:1,y:1). ... } (in[1] 3, in[1] 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 \cup {int,bool,}]

$$(x: \textcircled{k}, \tau', s \cup {int, bool}) =$$

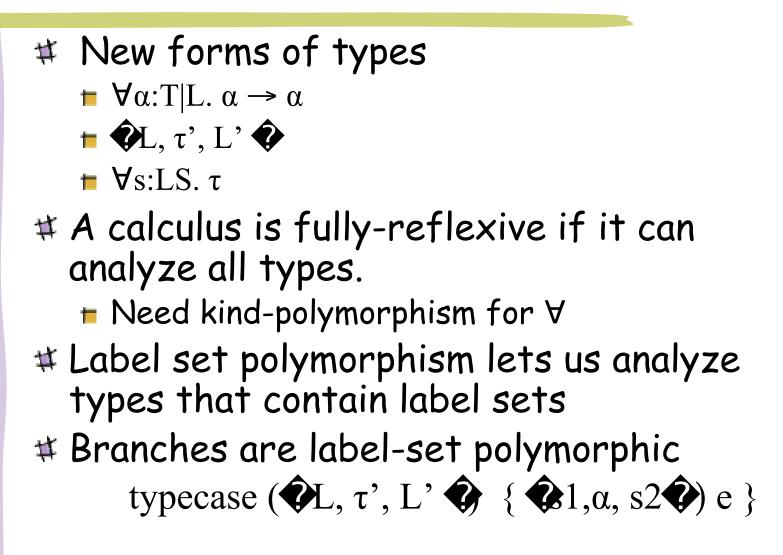
typecase α

 $x \cup \{ \text{ int })..., \text{bool }) ... \}$

call with:

eq [{1}] { 1) ... } [1] (in[1] 3, in[1] 6)

Fully-reflexive analysis



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

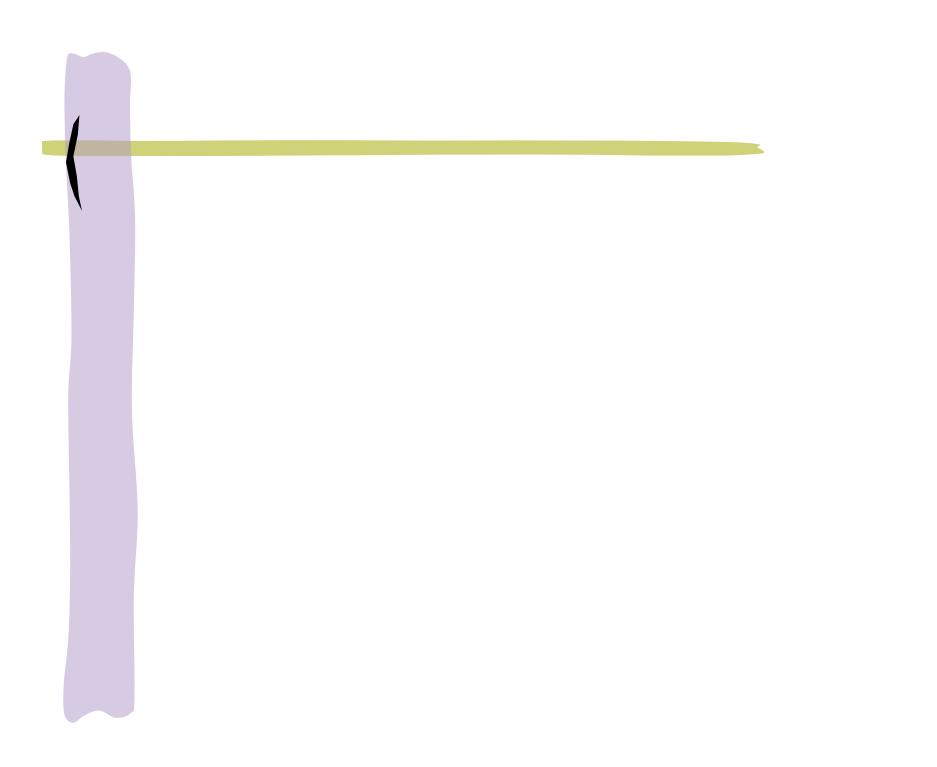
Extensions

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 eq : $\forall \alpha. (\alpha' \alpha) \rightarrow bool$ Call those operations polytypic

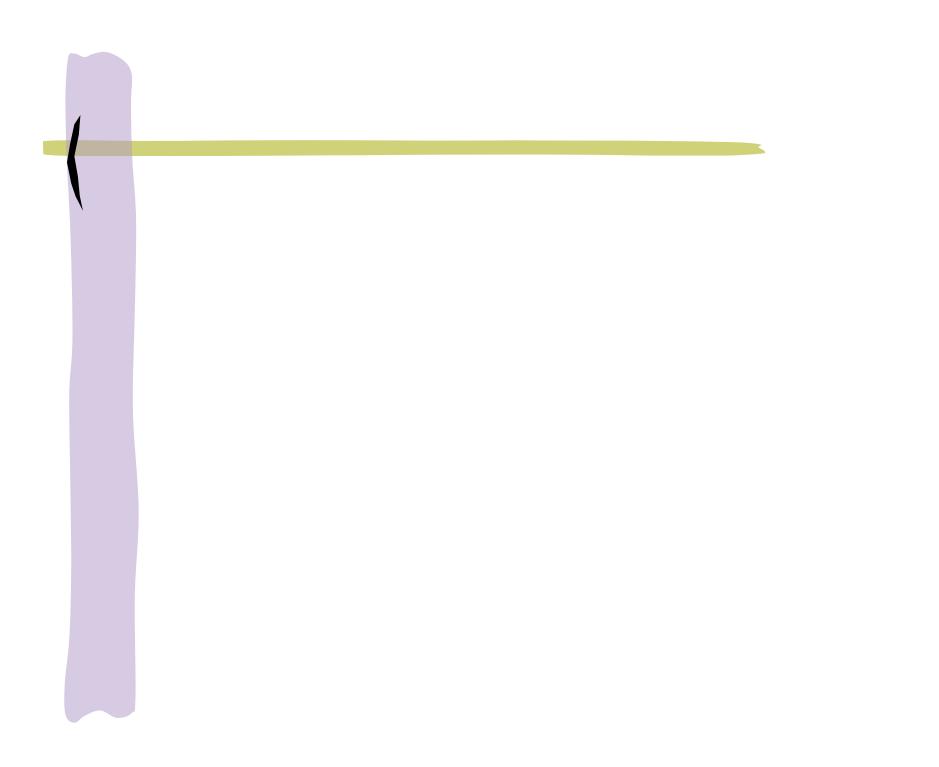
User-defined types

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 \rightarrow int **#** Type checker enforces distinction. × (in[Age] 29) + 1



Operational Semantics
Higher-order coercions

$$\{i:\lambda\alpha.int\}^+_1 \otimes i$$

 $\{(v_1,v_2):\lambda\alpha.\tau_1'\tau_2\}^+_1 \otimes$
 $(\{v_1:\lambda\alpha.\tau_1\}^+_1,\{v_2:\lambda\alpha.\tau_2\}^+_1)$
 $\{(\lambda x:\tau.e):\lambda\alpha.\tau_1 \rightarrow \tau_2\}^+_1$
 $\otimes \lambda x:\tau_1[l/\alpha]. \{e[\{x:\lambda\alpha.\tau_1\}^-_l/x]:$
 $\lambda\alpha.\tau_2\}^+_1$
 $\{v:\lambda\alpha.\alpha\}^+_1 \otimes in[l] v$

Universal set

- * Set T is set of all labels
- $\# f [\alpha:T|^T] \dots$
 - **f** can be applied to any type
 - eq[α] doesn't typecheck
 - a 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 \varnothing has type $\diamondsuit \varnothing, \tau', L \diamondsuit$ For arbitrary τ', L

$e_1 \cup e_2$ has type $\mathcal{O}L_1 \cup L_2$, τ ', L \mathcal{O} when = e1 has type $\mathcal{O}L_1$, τ ', L \mathcal{O} = e2 has type $\mathcal{O}L_2$, τ ', L \mathcal{O}

Union is non-disjoint

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

Can overwrite existing mappings: f [int] {int) 4} = 4 # Reversing order prevents overwrite: typecase α (x ∪ {int) 2})

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 $\forall s:LS. \forall \alpha:T|s \cup L. \clubsuit s, \tau', s \cup L \clubsuit \tau' \alpha$

What is the difference?

Open ops calling other ops

important : \forall s:LS. $\forall \alpha$:T|s. \diamondsuit , $\lambda\beta$. $\beta \rightarrow$ bool, s $\diamondsuit \rightarrow \alpha \rightarrow$ bool

```
print[s:LS][a:T|s]
  (mp : \mathbf{Q}, (\lambda\beta. \beta \rightarrow string), s \mathbf{Q} mi : \mathbf{Q}, \lambda\beta. \beta \rightarrow bool, s \mathbf{Q} =
     typecase \alpha 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 ...
```