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 *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

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.

Structural style

- Use a "case" term to branch on the structure of types

eq : ∀α. (α′α) → bool
eq[α: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)
    (β′γ)   ) λ(x: β′γ, y: β′γ).
            eq[β](x.1,y.1) && eq[γ](x.2,y.2)
    (β → γ)   ) error "Can't compare functions"

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:Age, y:Age) =
    let xi = out[Age] x
    let yi = out[Age] y
    if xi > 30 && yi > 30
    then true else xi == yi
```

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

```
newtype Phone = int
eq(x:Phone,y:Phone) =
  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 = 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 operators*:

  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

- 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

$$\text{Age} \times \text{int} \quad \text{to} \quad \text{int} \times \text{int}$$

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*

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

  $\{e{:}\ \lambda\alpha{:}T.\alpha' int\}^{-}_{Age} : (int \,' \, int)$
- A bit more complicated for type operators

Operational Semantics

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

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

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

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

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

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

- 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

```
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 > 30 && yi > 30
                  then true else xi == yi
```

- eq[Age] (in[Age] 31, in[Age] 45) = true
- eq[int] (31, 45) = 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 labels.

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

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

- Can instantiate f only with types formed by the above constants.
  - $\text{eq}\ [(\text{int}'\text{bool})\ '\text{Age}]$ is ok
  - $\text{eq}\ [\text{Phone}\ '\ \text{int}]$ is not
  - $\text{eq}\ [\text{int} \rightarrow \text{bool}]$ is not

Restricted polymorphism

- Typecase must have a branch for every label 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 = {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    )
```

Universal set

- Set ⊤ is set of all labels
- f [α:T|⊤] …
  - f can be applied to any type
  - eq[α] 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.

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:
  - $\varnothing$ — empty map
  - $\{l)e\}$ — 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: $\text{int}\ '\ \text{int} \rightarrow \text{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \text{bool})\ \text{int}$
  - bool branch: $\text{bool}\ '\ \text{bool} \rightarrow \text{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \text{bool})\ \text{bool}$
  - Age branch: $\text{Age}\ '\ \text{Age} \rightarrow \text{bool}$
    $= (\lambda\alpha.\ \alpha'\alpha \rightarrow \text{bool})\ \text{Age}$
  - Product branch:
    $\forall\beta{:}T|L.\forall\gamma{:}T|L.\ (\beta'\gamma)\ '\ (\beta'\gamma) \rightarrow \text{bool}$
    $= \forall\beta{:}T|L.\forall\gamma{:}T|L.\ (\ (\lambda\alpha.\ \alpha'\alpha \rightarrow \text{bool})\ (\beta'\gamma)\ )$

Type Operators

- In general: type of branch for label l with kind k is τ’η l:k | Lι
  - (λα.α ′α → bool) η int : T | L ι = int ′int → bool
  - (λα.α ′α → bool) η ′ : T → T → T | L ι

    = ∀β:T|L.∀γ:T|L. (β′γ) ′ (β′γ) → bool
- Expand this type:

  τ’ητ:T | Lι = τ’ τ

  τ’ητ:k_1→ k_2 | Lι = ∀α:k_1|L. τ’ητ α:k_2|Lι

Type of typecase

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

    e_i has type $\tau' \eta l_i : k_i \mid L \iota$
- Type of first-class label map must include
  - What labels are in domain
  - What τ' and L are for the branches

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 τ'η l : k | L_2ι

Other map formers

- empty map ∅ has type �∅, 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 [α :T | { int}]
 (x : �{int}, τ', L �) =
  typecase α ({int) 2} ∪ x)

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

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}] { 1) … } [l] (in[l] 3, in[l] 6)

Open vs. closed polytypic ops

- Closed version of eq has type

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

  where $L = \{\ \text{int}, \text{bool}, ', \text{Age}\}$

  $\tau' = \lambda\alpha.\ (\alpha\ '\ \alpha) \rightarrow \text{bool}$

- Open version of eq has type

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

- 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 …
```

Fully-reflexive analysis

- New forms of types
  - $\forall\alpha{:}T|L.\ \alpha \to \alpha$
  - �L, τ', L' �
  - $\forall s{:}LS.\ \tau$
- A calculus is fully-reflexive if it can analyze all types.
  - Need kind-polymorphism for ∀
- Label set polymorphism also 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

- 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/