Polytypic Programming and Intensional Type Analysis

Stephanie Weirich

Cornell University

Polytypic Programming

- Defines a function based on structural analysis of a type or type constructor.
- Examples: map, eq, zip, fold etc…
- Also called Generic Programming
- Look at version from Hinze's paper

"Polytypic values possess polykinded types"

Example: size⟨t⟩

- The size of a list is its length

  ⟦size⟨list⟩⟧ ≜

  sz_list [] = 0

  sz_list (hd::tl) = 1 + sz_list tl

- The size of a tree is the number of nodes

  ⟦size⟨tree⟩⟧ ≜

  sz_tree (Leaf) = 0

  sz_tree (Node(a,l,r)) =

  1 + sz_tree l + sz_tree r

Polytypic definition

size⟨1⟩ () = 0

size⟨int⟩ i = 0

size⟨+⟩ sza szb = λv. case v of

(Inl w) => sza w

(Inr w) => szb w

size⟨×⟩ sza szb = λv. sza (v.1) + szb (v.2)

type of rhs depends on kind of operator

size⟨t⟩ : Size ⟨k⟩ t if t : k

Size⟨★⟩ t = t → int

Size⟨k1 → k2⟩ t =

∀a. Size⟨k1⟩ a → Size⟨k2⟩ (t a)

Polytypic definition

$1 : \star$ so branch of type
() → int

size⟨1⟩ () = 0

size⟨int⟩ i = 0

size⟨+⟩ sza szb = λv. case v of

(Inl w) => sza w

(Inr w) => szb w

size⟨×⟩ sza szb = λv. sza (v.1) + szb (v.2)

type of rhs depends on kind of operator

size⟨t⟩ : Size ⟨k⟩ t if t : k

Size⟨★⟩ t = t → int

Size⟨k1 → k2⟩ t =

∀a. Size⟨k1⟩ a → Size⟨k2⟩ (t a)

Polytypic definition

```
size⟨1⟩ () = 0
size⟨int⟩ i = 0
size⟨+⟩ sza szb = λv. case v of
        (Inl w) => sza w
        (Inr w) => szb w
size⟨×⟩ sza szb = λv. sza (v.1) + szb (v.2)
```

int : ★ so branch of type int → int

type of rhs depends on kind of operator

```
size⟨t⟩ : Size ⟨k⟩ t              if  t : k
Size⟨★⟩ t          = t → int
Size⟨k1 → k2⟩ t =
        ∀a. Size⟨k1⟩ a → Size⟨k2⟩ (t a)
```

Polytypic definition

$+ : \star \to \star \to \star$ so branch is of type
$\forall a.\ (a \to int) \to \forall b.\ (b \to int) \to (a+b) \to int$

$size\langle 1 \rangle\ () = 0$

$size\langle int \rangle\ i = 0$

$size\langle + \rangle\ sza\ szb = \lambda v.$ case v of
(Inl w) => sza w
(Inr w) => szb w

$size\langle \times \rangle\ sza\ szb = \lambda v.\ sza\ (v.1) + szb\ (v.2)$

type of rhs depends on kind of operator

$size\langle t \rangle : Size\ \langle k \rangle\ t$ if $t : k$

$Size\langle \star \rangle\ t = t \to int$

$Size\langle k1 \to k2 \rangle\ t =$

$\forall a.\ Size\langle k1 \rangle\ a \to Size\langle k2 \rangle\ (t\ a)$

Polytypic definition

size⟨1⟩ () = 0

size⟨int⟩ i = 0

size⟨+⟩ sza szb = λv. case

(Inl w) => sz

(Inr w) => szb w

size⟨×⟩ sza szb = λv. sza (v.1) + szb (v.2)

× : ★ → ★ → ★ so branch is of type ∀a. (a → int) → ∀b. (b → int) → (a×b) → int

type of rhs depends on kind of operator

size⟨t⟩ : Size ⟨k⟩ t if t : k

Size⟨★⟩ t = t → int

Size⟨k1 → k2⟩ t =

∀a. Size⟨k1⟩ a → Size⟨k2⟩ (t a)

Definition (continued)

We've defined size for some types/type constructors, what about the rest?

size⟨λa:k.t⟩ =

Λa:k. λsza:Size⟨k⟩a. size⟨t⟩

size⟨a⟩ = sza

size⟨t1 t2⟩ = size⟨t1⟩[t2] size⟨t2⟩

size⟨μ⟩ = Λa:★→★. λsza. fix (sza [μ a])

Example

- So size specialized to list is a function from the size of a to the size of list a

size⟨list⟩ :

$$\forall a{:}\star.\ (a \to \mathrm{Int}) \to (\mathrm{list}\ a \to \mathrm{Int})$$

- Apply constant function to get sz_list

sz_list = Λa. size⟨list⟩[a] (λx:a. 1)

Example

$[\![\text{size}\langle\text{list}\rangle]\!] = [\![\text{size}\langle\lambda a.\ \mu\ (\lambda b.\ 1+\ a\times b)\rangle]\!]$

$= \Lambda a.\lambda xa.\ \text{fix}\ \lambda xb.\ [\![\text{size}\langle 1+a\times b\rangle]\!]$

$= \Lambda a.\lambda xa.\ \text{fix}\ \lambda xb.\ (\Lambda c.\ \lambda szc.\ \Lambda d.\ \lambda szd.$

(λv:c+d.case v of

(Inl w) -> szc w

(Inr w) -> szd w))

[1] $[\![\text{size}\langle 1\rangle]\!]$ [a×b] $[\![\text{size}\langle a\times b\rangle]\!]$

$= \Lambda a.\lambda xa.\ \text{fix}\ \lambda xb.\ \lambda v.$ case v of

(Inl w)-> (λw. 0) w

(Inr w)->(λw. $[\![\text{size}\langle a\rangle]\!]$ w.1+ $[\![\text{size}\langle b\rangle]\!]$w.2) w

$= \Lambda a.\ \lambda xa.\ \text{fix}\ \lambda xb.\ \lambda v.$ case v of

(Inl w) -> 0

(Inr w) -> xa (w.1) + xb (w.2)

Schema for polytypic functions

- A polytypic function is defined over the structure of a type.
- But a type is actually an equivalence classes of structures
  - $(\lambda a{:}\star.a + \mathrm{int})\ 1$ and $1+\mathrm{int}$ are the same type
- A polytypic function must give the same result for all equivalent types:
  - $\mathrm{size}\langle(\lambda a{:}\star.a + \mathrm{int})\ 1\rangle = \mathrm{size}\langle 1+\mathrm{int}\rangle$
- We are limited in the definition of $\mathrm{size}\langle\lambda a{:}k.t\rangle$

- For the type language elements, the definition is (almost) always the same:

$[\![\text{poly}\langle \lambda a{:}k.t\rangle]\!] =$

$\quad \Lambda a{:}k.\ \lambda sza{:}\text{Poly}\langle k\rangle a.\ [\![\text{poly}\langle t\rangle]\!]$

$[\![\text{poly}\langle a\rangle]\!] = sza$

$[\![\text{poly}\langle t1\ t2\rangle]\!] = [\![\text{poly}\langle t1\rangle]\!][t2][\![\text{poly}\langle t2\rangle]\!]$

$[\![\text{poly}\langle \mu\rangle]\!] = \Lambda a{:}\star\rightarrow\star.\ \lambda sza.\ \text{fix}\ (sza\ [\mu\ a])$

Schema for polytypic functions

Therefore, the type (almost) always has the same schema

some function of t

poly⟨t⟩ : Poly⟨k⟩ t

Poly⟨★⟩ t = …

Poly⟨k1 → k2⟩ t =

∀a:k1. Poly⟨k1⟩ a → Poly⟨k2⟩ (t a)

If we fill in the blank with (c t) let's notate the definition as: Poly⟨k⟩ c t

e.g. Size⟨k⟩ t = Poly⟨k⟩(λa. a → int) t

Operational semantics

- In some sense, intensional type analysis provides an operational semantics for polytypic programming.
- Instead of compiling away type examination, ita permits run-time type analysis (necessitating type passing in its implementation).
- But it is not defined for type constructors, only types.

Analysis of base types works

```
size =  typerec[λa.a → int] a of
    1 => λx:1. 0
    int => λx:int. 0
    a + b => λsizea. λsizeb. λx:a+b .
        case x of (Inl v) => sizea v
                  (Inr v) => sizeb v
    a×b =>λ sizea. λsizeb. λx:a×b.
        sizea (x.1) + sizeb (x.2)
```

Typechecking rule for typerec

$G \vdash a : \star$

$G \vdash c : \star \to \star$

$G \vdash e1 : c\ 1$

$G \vdash eint : c\ int$

$G \vdash e\to : \forall\ a.\ ca \to \forall\ b.\ cb \to c\ (a \to b)$

$G \vdash e\times : \forall\ a.\ ca \to \forall\ b.\ cb \to c\ (a \times b)$

$G \vdash e+ : \forall\ a.\ ca \to \forall\ b.\ cb \to c\ (a + b)$

$G \vdash typerec[c]\ a\ of\ opts : c\ a$

Note: For the rest of the talk read opts as
(e1, eint, e→, e×, e+)

Simplified typechecking rule

$G \vdash a : \star$

$G \vdash c : \star \to \star$

$G \vdash eop : Poly\langle kind\ op \rangle\ c\ op$

---------- (for op = 1, int, $\to$, $\times$, +)

$G \vdash typerec[c]\ a\ of\ opts : c\ a$

i.e. $e\times : Poly\langle \star \to \star \to \star \rangle c\ \times$

$= \forall a.\ Poly\langle \star \rangle c\ a \to Poly\langle \star \to \star \rangle c\ (\times a)$

$= \forall a.\ ca \to \forall b.\ cb \to c\ (a \times b)$

Operational Semantics

typerec[c] 1 opts → e1

typerec[c] int opts → eint

typerec[c] (a→b) opts →

e→ [a] (typerec[c] a opts) [b] (typerec[c] b opts)

typerec[c] (a+b) opts →

e+ [a] (typerec[c] a opts) [b] (typerec[c] b opts)

typerec[c] (a×b) opts →

e× [a] (typerec[c] a opts) [b] (typerec[c] b opts)

Extended typechecking rule

(First try)

$G \vdash a : k$

$G \vdash c : \star \rightarrow \star$

$G \vdash eop : Poly\langle kind\ op \rangle\ c\ op$

$G \vdash typerec[c]\ a\ of\ opts :\ Poly\langle k \rangle\ c\ a$

- We have to make the operational semantics of typerec match the "interpretation" from before

Operational Semantics

⟦poly⟨λa:k.t⟩⟧ =

  Λa:k. λxa:Poly⟨k⟩c a. ⟦poly⟨t⟩⟧

typerec[c] (λa:k. t) opts →

  Λa:k. λxa:Poly⟨k⟩c a. typerec[c] t opts

⟦poly⟨t1 t2⟩⟧= ⟦poly⟨t1⟩⟧[t2]⟦poly⟨t2⟩⟧

typerec[c] (t1 t2) opts →

  (typerec[c] t1 opts) [t2]

  (typerec[c] t2 opts)

Operational Semantics (cont)

⟦poly⟨a⟩⟧ = xa

typerec[c] a ops → ??

- Traditionally, transition rules are only defined for closed terms.
- In practice, evaluation is suspended (by an enclosing Λ) until a is provided.
- When this happens, a will be replaced with some other closed type, which will then be analyzed.
- We don't want this to happen!

Idea

- Use a placeholder, to "mark" the location of a

typerec[c] (λa:k. t) opts →

Λa:k. λxa:Poly⟨k⟩c a.

typerec t{Place xa/a} opts

typerec[c] (Place e) opts → e

- How do we know e is the right type?
- Strange to have terms in types… What term does (Place e) describe anyway?

R-types

- Side-step the issue: Separate types used for type-checking from those used for type analysis.
- Have terms represent types.
  - ex: special constant R1 represents type 1
- Then term representation includes placeholder (representing a type variable, even when that variable may be beta-replaced).

Term representations

- Special type constructor R indicates what type is represented

R1 : R(1)

Rint : R(int)

R+ : ∀ a. R a → ∀ b. R b → R(a+b)

- example: represent type (1+int) with term

  R+ [1] R1 [int] Rint

- Another simplification to typing rules:

  Rop : Poly⟨kind op⟩R op

Representations of constructors

- This even works for representations of type constructors
- Fans of λ_R may recognize the following definition:

$R(c:\star) = R(c)$

$R(c:k1 \rightarrow k2) =$

$\quad \forall a :k1.\ R(a:k1) \rightarrow R(ca : k2)$

- Exactly the same thing is going on!

$R(c:k) = \mathrm{Poly}\langle k \rangle R\ c$

- Representation of $\lambda a:\star.a$ is

  $\Lambda a:\star.\ \lambda xa:R(a:\star).xa$

Modified typing judgement

(second try)

$G \vdash a : k$

$G \vdash e : \text{Poly<k>R a}$

$G \vdash c : \star \rightarrow \star$

$G \vdash eop : \text{Poly}\langle \text{kind op} \rangle \text{c op}$

$G \vdash \text{typerec[c] e of opts} : \text{Poly}\langle k \rangle \text{c a}$

Place holder

typerec[c] (Λa:k.λxa:R(a:k). e) opts →

Λa:k. λxa:Poly⟨k⟩c a.

typerec e{Place xa/xa} opts

typerec[c] (Place e) opts → e

- How do we know e is the right type?
- if Place is of type: Poly⟨k⟩c a → R(a:k) it produces a representation that may be used for any iteration!

Solution: modify R

- For soundness must specialize R by return type of typerec
- R : (★→★) → ★ → ★

(final try)

G ⊢ a : k

G ⊢ e : Poly<k>(R c) a

G ⊢ c : ★→★

G ⊢ eop : Poly⟨kind op⟩c op

G ⊢ typerec[c] e of opts : Poly⟨k⟩c a

Place : Poly⟨k⟩ c a → Poly⟨k⟩(R c) a

Operational semantics

typerec[c] (Place e) opts → e

typerec[c] (Λa:k.λx:Poly⟨k⟩(Rc) a. e) opts
→
Λa:k.λxa:Poly⟨k⟩c a.
typerec[c] e{Place xa/x} opts

typerec[c] (e1 [c] e2) opts →
(typerec[c] e1 opts) [c]
(typerec[c] e2 opts)

Operational Semantics

typerec[c] (R1) opts → e1

typerec[c] (Rint) opts → eint

typerec[c] (R+) opts → e+

typerec[c] (R×) opts → e×

typerec[c] (R→) opts → e→

why not also:

typerec[c] (R∀) opts → e∀

typerec[c] (R∃) opts → e∃

Types with binding structure

- Use higher order abstract syntax

$\forall, \exists: (\star \to \star) \to \star$

$R\forall$: $Poly\langle(\star \to \star) \to \star\rangle$ (Rc) $\forall$

$= \forall a{:}\star \to \star.(\forall b{:}\star.\ Rcb \to Rc(ab)) \to Rc(\forall a)$

i.e. create a representation of a polymorphic type with a function from a representation of the arg (b) to the representation of the result (ab).

- Kind polymorphism required for $F\omega$

$\forall : \forall\chi.\ \chi \to \star$

More schema for polytypism

- A polytypic function like map requires a different schema to describe its type:

$\mathsf{Poly}\langle\star\rangle\ t1\ t2 = c\ t1\ t2$

$\mathsf{Poly}\langle k1 \to k2\rangle\ t1\ t2 =$

$\quad \forall a1\ a2.\ \mathsf{Poly}\langle k1\rangle\ a1\ a2 \to$

$\qquad\qquad \mathsf{Poly}\langle k2\rangle\ (t1\ a1)\ (t2\ a2)$

- example - $\mathsf{map}\langle f{:}\star\to\star\rangle$:

$\qquad \forall a1\ a2.\ a1 \to a2 \to (f\ a1) \to (f\ a2)$

$\mathsf{map}\langle f\rangle\ :\ \mathsf{Poly}\langle\star\to\star\rangle\ (\lambda t1\ t2.\ t1 \to t2)$

Multiplace relation

- In fact we can generalize the polytypic schema to n arguments:

$\mathrm{Poly}_n\langle\star\rangle\ t_1 \ldots t_n = c\ t_1 \ldots t_n$

$\mathrm{Poly}_n\langle k1 \to k2\rangle\ t_1 \ldots t_n =$

$\forall\ a_1{:}k1 \ldots a_n{:}k1.\ \mathrm{Poly}_n\langle k1\rangle\ c\ a_1 \ldots a_n \to \mathrm{Poly}_n\langle k2\rangle\ c\ (t_1\ a_1) \ldots (t_n\ a_n)$

- How do we implement this schema with typerec?

Easy answer

- $\mathrm{Poly}_n\langle k\rangle$ c $t_1 \ldots t_n$
- typerec, representations, place all keep track of n

R_n: $(\star^n \to \star) \to \star^n \to \star$

Rop_n : $\mathrm{Poly}_n\langle k\rangle$ c op … op

Place_n : $\mathrm{Poly}_n\langle k\rangle$ c $t_1 \ldots t_n \to$ $\mathrm{Poly}_n\langle k\rangle$ $(R_n c)$ $t_1 \ldots t_n$

$\mathrm{typerec}_n(\Lambda a_1 \ldots a_n.\lambda x.e) \to$

$\Lambda a_1 \ldots a_n{:}k.\ \lambda y{:}\mathrm{Poly}_n\langle k\rangle$ c $a_1 \ldots a_n$.

$\mathrm{typerec}_n$ e{Place_n y/x}

Example

- Even the "0"th version is useful…

print_type = Λa. λx:(R_0 string a).
 typerec$_0$ x of
  1 => "1"
  int => "int"
  a+b => λpa pb. pa ^ "+" ^ pb
  ∀ f => λpf: string → string.
   let x = gensym() in
   "all" ^ x ^ "." ^ (pf x)

Example

- Even the "0"th version is useful…

print_type = Λa. λx

typerec$_0$ x of

1 => "1"

int => "int"

a+b => λpa pb. pa ^ "+" ^ pb

∀ f => λpf: string → string.

let x = gensym() in

"all" ^ x ^ "." ^ (pf x)

With typerec$_1$ this would be of type ∀a:★. string → string

Example

- Even the "0"th version is useful…

print_type = Λa. λx

typerec$_0$ x of

1 => "1"

int => "int"

a+b => λpa pb. pa ^ "+

∀ f => λpf: string → string.

let x = gensym() in

"all" ^ x ^ "." ^ (pf x)

With typerec$_1$ this would be of type ∀a:★. string → string

And we'd have to add a "dummy" type arg here

Harder answer

- With the easy answer we lose an important property – that all polytypic functions can be created from the same type representation.
  - Representations of int + int
  - $R+_0$ $Rint_0$ $Rint_0$ for print_type
  - $R+_1$ [int] $Rint_1$ [int] $Rint_1$ for size
  - $R+_2$ [int][int] $Rint_2$ [int][int] $Rint_2$ for map
- This is really annoying considering all of these representations have the same type erasure.
- Maybe some sort of kind or arity polymorphism will do the trick…

Recursive types?

- equi-recursive types
  - type is equivalent to its unrolling
- iso-recursive types (LMLI, etc)
  - type must be coerced to its unrolling
- Design choice of language

equi-recursive types

- μ t = t(μ t)
- μ shouldn't be visible to analysis
- typerec μ hardwired to fix.

⟦poly⟨μ⟩⟧ =

Λa:★→★. λpa:Poly(c)⟨★→★⟩ a.

fix (pa [μ a])

typerec[c] μ opts →

Λa:★→★. λpa:Poly(c)⟨★→★⟩ a. fix (pa [μ a])

iso-recursive types

- requires explicit coercion
  - roll : t(μt) → μt
  - unroll : μt → t(μt)
- mu should be visible to analysis
  - typerec (μ) = eμ
- sometimes makes a difference

  print_type : all a. string

  eμ == λr : string → string

  “mu” ++ var ++ “.” ++ r (var)

Summary

- A nifty way to write polytypic functions with typerec
- Handles types with binding structure nicely:
  - iso-recursive, polymorphic, and existential types

Future work

- Type level type analysis (Typerec) interacts badly. (Equivalence classes of type syntax becomes more complicated)
- Haven’t worked out multi-place representations
- Can’t write every function that I want to write : second version of cast from ICFP 00