Polytypic Programming and Intensional Type Analysis

AKA: Run-Time Type Constructor Analysis

Stephanie Weirich

Polytypic programming

- Defines operations over ***static*** type information
  - Examples: Charity, FML, PolyP, Generic Haskell
- Many wonderful applications:
  - Structural equality, cloning, hashing, checkpointing, marshalling, iterators, visitor patterns, pattern matching, unification
  - Automatically generates these functions for every type.
- Based on category theory (definable functors)

Not universal

- However, sometimes type information isn't known statically
  - Type variables in polymorphic functions
  - Polymorphic recursion
  - Separate compilation
  - Dynamic linking

Run-time type analysis

- Language propagates type information at run time.
- Dynamic type information leads to new applications
  - Garbage collection, dynamic loading, stub generation, proxies, extensible systems, implementation of polymorphism

Intensional Type Analysis

- Harper & Morrisett - λ_i^{ML}
- Type system guarantees that type information is used correctly.
- The term *typecase* analyzes type variables through pattern matching.

Λ α. typecase α of

int ⇒ …

β × γ ⇒ …

Operation

- At run time, typecase pattern matches its type argument

typecase [int]

int ⇒ "int type"

β × γ ⇒ "product type"

β → γ ⇒ "function type"

evaluates to **"int type"**

Example: serialization

valtostring = Λα. typecase α of

int ⇒ int2string

bool ⇒ λx:bool. if x then "true" else "false"

β × γ ⇒ λx:(β × γ).

"(" + valtostring[β] (x.1) + ","

+ valtostring[γ] (x.2) + ")"

β → γ ⇒ λx:(β → γ).

"<function>"

Branches reflect type information

valtostring = Λα. typecase α of

int ⇒ …// of type int → string

bool ⇒ … // of type bool → string

β × γ ⇒ … // of type (β × γ) → string

β → γ ⇒ … // of type (β → γ) → string

So valtostring is of type

∀α. α → string.

Not as expressive…

The function *valtostring* was defined in terms of a type.

However, many motivating examples of polytypic programming are defined by ***type constructors*** such as list, option, and tree.

Polytypic Programming

$\mathbf{map}[\tau] : \forall \alpha\beta.\ (\alpha \to \beta) \to (\tau\ \alpha) \to (\tau\ \beta)$

$\mathbf{size}\ [\tau] : \forall \alpha.\ (\alpha \to \mathbf{int}) \to (\tau\ \alpha) \to \mathbf{int}$

$\mathbf{zip}\ [\tau] : \forall \alpha\beta.\ (\tau\ \mathbf{a}) \to (\tau\ \beta) \to (\tau\ (\alpha \times \beta))$

$\mathbf{map[list]} : \forall \alpha\beta.(\alpha \to \beta) \to (\mathbf{list}\ \alpha) \to (\mathbf{list}\ \beta)$

$\mathbf{size[option]} : \forall \alpha.\ (\alpha \to \mathbf{int}) \to (\mathbf{option}\ \alpha) \to \mathbf{int}$

$\mathbf{zip\ [tree]} : \forall \alpha\beta.\ (\mathbf{tree\ a}) \to (\mathbf{tree}\ \beta) \to (\mathbf{tree}\ (\alpha \times \beta))$

Problem

Because typecase analyzes types, not type constructors, we can't write **map**, **size**, or **zip** with typecase.

Why not?

- Functions in lmli *iterate* over the structure of types with typecase.
- For a type constructor **τ**, only recourse is to apply **τ** to something else to produce a type.

map [τ] = Λτ. Λα. λf :(α → β).

typecase [τ α] of

…

This can't work

opt1 a = unit + a

opt2 a = unit + unit

map[opt1] f = λx.

case x of (inl y⇒ inl () | inr y ⇒ f y)

map[opt2] f = λx.x

What if we want to map **(λx:unit. 3)** over the two different datastructures?

We get the same result

map[opt1][unit][int] f

= typecase [opt1 unit] of …

= typecase [unit+unit] of …

map[opt2][unit][int] f

= typecase [opt2 unit] of …

= typecase [unit+unit] of …

A new approach

- A new term for analyzing types: ***tinterp***
  - typecase desconstructs types
  - tinterp *interprets* the type language as terms

- Inspired by previous work on polytypism
  - A polytypic function should be an interpretation of the type language [Hinze].
  - This talk incorporates that idea into a type passing framework instead of a meta-level operation.
  - Analogy: templates in C++ vs. polymorphism in ML.

Type language

$\kappa ::= T \mid \kappa_1 \rightarrow \kappa_2$

$\tau ::= \alpha$

$\mid \lambda\alpha{:}\kappa.\ \tau$

$\mid \tau_1\ \tau_2$

$\mid$ **int** $\mid$ **string**

$\mid \rightarrow \mid \times \mid \forall \mid \exists \mid \mu \mid \ldots$

- kinds
- variables
- functions
- applications
- constants

- **The type int $\times$ int is the constant $\times$ applied to int twice.**
- **The type $\forall\alpha\ .\ \alpha \rightarrow\alpha$ is the constant $\forall$ applied to the type constructor $(\lambda\alpha{:}T.\ \alpha \rightarrow\alpha\ )$.**

Type constructors

- Type constructors are functions in this language.

$$\textbf{option} = \lambda\alpha{:}T.\ \textbf{unit} + \alpha$$
$$\textbf{list} = \lambda\alpha{:}T.\ \mu(\lambda\beta.\ \textbf{unit} + (\alpha \times \beta))$$
$$\textbf{tree} = \lambda\alpha{:}T.\ \mu(\lambda\beta.\ \textbf{unit} + (\alpha \times \alpha \times \beta))$$

Interpreter

Define analysis term: **tinterp[η] τ**

- Operational semantics of tinterp is an interpreter of the type language.
  - Invariant: if $\tau = \tau'$ then tinterp[η] τ is equivalent to tinterp[η] τ'
- η is an environment
  - mappings for all constants (like branches of typecase)
  - mappings for all type variables.

Operational semantics of tinterp

- Type constants are retrieved from the environment

  tinterp[η] int ➔ η(int)

  tinterp[η] bool ➔ η(bool)

  tinterp[η] → ➔ η(→)

  tinterp[η] × ➔ η(×)

  tinterp[η] ∀ ➔ η(∀)

- Type variables are retrieved from the environment

  tinterp[η] α ➔ η(α)

Type functions – Version 1

- Type functions are mapped to term functions.

  tinterp[η] (λα:κ.τ) →

  λx. tinterp[η+{α↦x}] (τ)

  – Applying this function will analyze the open type **τ** with an additional branch for **α.**

- Type application steps to term application.

  tinterp[η] (τ_1 τ_2)

  → (tinterp[η] τ_1) (tinterp[η] τ_2)

Why does this make sense?

- Type β-equivalence $(\lambda\alpha.\ \tau)\tau' = \tau\{\tau'/\alpha\}$ preserved by term β-equivalence.

tinterp[η] $(\lambda\alpha.\ \tau)\tau'$

→ (tinterp[η] $(\lambda\alpha.\tau)$)(tinterp[η] τ')

→ $(\lambda x.$ tinterp[η+{$\alpha \mapsto x$}] τ) (tinterp[η] τ')

→ tinterp[η+{$\alpha \mapsto$tinterp[η] τ'}] τ

= tinterp[η] ($\tau\{\tau'/\alpha\}$)

Example: typetostring

```
typetostring[τ] = tinterp [η] τ
    where η =  {
          int      ⇒ "int"
          string   ⇒ "string"
          ×        ⇒  λr:string. λs:string.
                          "(" + r + "*" + s + ")"
          →        ⇒  λx:string. λy:string.
                          "(" + r + "->" + s + ")"
          ∀        ⇒  λr:(string → string).
                          let v = gensym () in
                          "(all " + v + "." + (r v) + ")"
    }
```

Example execution

typetostring[int×int]

- **➔ (tinterp[η] ×) (tinterp[η] int) (tinterp[η] int)**
- **➔ (λx:string. λy:string. "("+ x +"*"+ y +")")**
  (tinterp[η] int) (tinterp[η] int)
- **➔ (λx:string. λy:string. "("+ x +"*"+ y +")")**
  "int" "int"
- **➔ "(" + "int" + "*" + "int" + ")"**
- **➔ "(int*int)"**

How do we typecheck tinterp?

- The result type of tinterp depends on the kind of its argument.

$$\sigma\langle T\rangle = \sigma$$

$$\sigma\langle \kappa_1 \to \kappa_2\rangle = \sigma\langle \kappa_1\rangle \to \sigma\langle \kappa_2\rangle$$

Example:

typetostring[int] : string

typetostring[λα:T. unit + α] : string → string

***Terminology*:**

Hinze calls $\sigma\langle\kappa\rangle$ a *polykinded type*.

What about the environment?

- If α has kind κ then $\eta(\alpha)$ must have type $\sigma\langle\kappa\rangle$

Example:

int:T and $\times$ **:T→T→T**

int $\Rightarrow$ **"int"** **: string**

$\times$ $\Rightarrow$ **λr:string. λs:string.**

"(" + r + "*" + s + ")"

: string → string → string

What about valtostring?

valtostring = tinterp[η] α

where η = {

int ⇒ int2string : int → string

bool ⇒ λx:bool. if x then "true" else "false"

: bool → string

… ⇒…

}

- Doesn't typecheck!
- **int** and **bool** have the same kind so **η(int)** and **η(bool)** should be of the same type.

Not descriptive enough

valtostring = tinterp[η] α

where η = {

int ⇒ int2string : (λα.α → string) int

bool ⇒ λx:bool. if x then "true" else "false"

: (λα.α → string) bool

… ⇒…

}

- Doesn't typecheck!
- **int** and **bool** have the same kind so **η(int)** and **η(bool)** should be of the same type.

Polykinded types – Version 2

- Parameterize the return type
  - Before, **σ** was of kind **T.**

$$\sigma\langle T\rangle = \sigma$$

$$\sigma\langle \kappa_1 \to \kappa_2\rangle = \sigma\langle \kappa_1\rangle \to \sigma\langle \kappa_2\rangle$$

  - This time, **σ** is a function of kind **T → T.**

$$\sigma\langle \tau : T\rangle = \sigma\,\tau$$

$$\sigma\langle \tau : \kappa_1 \to \kappa_2\rangle =$$

$$\forall \alpha{:}\kappa_1.\ \sigma\langle \alpha : \kappa_1\rangle \to \sigma\langle \tau\,\alpha : \kappa_2\rangle$$

Interpret a type constructor as a polymorphic function

New operational semantics

- Interpret **(λα:κ. τ)** as a *polymorphic* function.

tinterp[η,ρ] (λα:κ.τ) ➔

Λβ:κ. λx: σ⟨β:κ⟩ .

tinterp[η+{α↦x}, ρ+{α↦β}] (τ)

- Interpret **(τ_1 τ_2)** as a type and term application.

tinterp[η,ρ] (τ_1 τ_2) ➔

(tinterp[η,ρ] τ_1) [ρ(τ_2)] (tinterp[η,ρ] τ_2)

New operational semantics

- Interpret **(λα:κ. τ)** as a *polymorphic* function.

tinterp[η,ρ] (λα:κ.τ) ➔

Λβ:κ. λx: σ⟨β:κ⟩ .

tinterp[η+{α↦x}, ρ+{α↦β}] (τ)

- Interpret **(τ_1 τ_2)** as a type and term application.

tinterp[η,ρ] (τ_1 τ_2) ➔

(tinterp[η,ρ] τ_1) [ρ(τ_2)] (tinterp[η,ρ] τ_2)

Example: valtostring

```
valtostring = Λα. tinterp [η,ρ] α  where η = {
    int  ⇒  int2string
    ×    ⇒ Λβ:T. λr:(β → string). Λγ:T. λs:(γ → string).
                λx:(β × γ).
                    "(" + r(x.1) + "," + s(x.2) + ")"
    ∃    ⇒ Λβ:T→T.
                λr:(∀γ. (γ →string) →(βγ →string) ).
                    λ x: ∃β.  let <γ, y> = unpack x in
                    r [γ] (valtostring[γ]) y
}
```

More flexibility

let x = pack < 2,3 > as ∃ α.int × α

What should be the result of

tostring[∃ α. int × α] x

- Before: **<2,3>**
- What if we want the abstract type to stay hidden ?

  < #hidden, 3>

Preserving abstraction

```
valtostring =  Λα. tinterp[η,ρ] α   where η = {
      int      ⇒ int2string
      string ⇒ λ x:string. "\"" + x + "\""
      β × γ  ⇒ λr:(β →string). λs:(γ → string).
                    λx:( β × γ ).
                       "<" + r(x.1) + "," + s(x.2)  + ">"
      ∃β       ⇒ λr:(∀α.(α →string) → (β α) → string).
                    λ x. let <y , γ > = unpack x in
                    r [γ] (λx:γ. "#hidden") y
   }
```

Generalized Polykinded Types

- Seen instances of 0 and 1, now generalize to n.
- **σ** is of kind **T→…→T→T**

$$\sigma \langle \tau_1 \ldots \tau_n : T \rangle = \sigma\, \tau_1 \ldots \tau_n$$

$$\sigma \langle \tau_1 \ldots \tau_n : \kappa_1 \to \kappa_2 \rangle =$$

$$\forall \alpha_1 \ldots \alpha_n .$$

$$\sigma \langle \alpha_1 \ldots \alpha_n : \kappa_1 \rangle \to \sigma \langle (\tau_1\, \alpha_1) \ldots (\tau_n\, \alpha_n) : \kappa_2 \rangle$$

Example:

A polykinded type, parameterized twice by α

- What about map? n=2.

$$\text{map} : \forall\alpha{:}T\to T.\ (\to)\ \langle \alpha\ \ \alpha : T\to T\ \rangle$$

$$= \forall\alpha{:}T\to T.$$

$$\forall\beta_1.\forall\beta_2.\ (\to)\ \langle\ \beta_1\ \beta_2 : T\ \rangle \to \langle\ (\alpha\ \beta_1)\ (\alpha\ \beta_2)\ : T\to T\ \rangle$$

$$= \forall\alpha{:}T\to T.$$

$$\forall\beta_1.\forall\beta_2.\ (\beta_1 \to \beta_2) \to \alpha\ \beta_1 \to \alpha\ \beta_2$$

Example: map

$$\text{map} = \Lambda\alpha{:}T \to T.\ \text{tinterp}\ [\eta,\rho_1,\rho_2]\ \alpha \ \text{ where } \eta = \{$$

$$\text{int} \ \Rightarrow\ \lambda x{:}\text{int}.\ x$$

$$\text{unit} \Rightarrow\ \lambda x{:}\text{unit}.\ x$$

$$\times \ \Rightarrow \Lambda\alpha_1.\Lambda\alpha_2.\ \lambda r{:}(\alpha_1 \to \alpha_2).\ \ \Lambda\beta_1.\Lambda\beta_2.\ \lambda s{:}(\beta_1 \to \beta_2).$$

$$\lambda x{:}\ \alpha_1 \times \beta_1.\ \ < r\ (x.1),\ s\ (x.2) >$$

$$+ \ \Rightarrow \Lambda\alpha_1.\Lambda\alpha_2.\ \lambda r{:}(\alpha_1 \to \alpha_2).\ \Lambda\beta_1.\Lambda\beta_2.\ \lambda s{:}(\beta_1 \to \beta_2).$$

$$\lambda x{:}(\alpha_1 + \beta_1).\ \ \text{case } x \text{ of}$$

$$\text{inl } y \Rightarrow \text{inl}(r\ y) \mid \text{inr } z \Rightarrow \text{inr}(s\ z)$$

$$\}$$

Extensions

- Kind polymorphism extends applicability of polytypic definitions.
- Type-erasure semantics for tinterp
  - Show how to represent types with terms, yet retain the type checking properties of ITA.
  - Makes phase distinction clear.
  - Control over what types are analyzable
- Details in *Programming with Types*, available from www.cis.upenn.edu/~sweirich/