

A Critique of Statically Typed Languages

Statically typed languages are sometimes criticized on two grounds:

- Types are obtrusive: types overwhelm the code.
- Types inhibit code re-use: one version for each type.

1

Enhancing Concision: Type Inference

Implicitly-typed languages allow omission of type information.

```
val axpby =  
  fn (a, b) => fn (x, y) => a*x + b*y
```

Officially, the types are present, we just aren't required to specify them.

```
val axpby : int * int -> int * int -> int =  
  fn (a:int,b:int):int => fn (x:int,y:int):int => a*x + b*y
```

2

Enhancing Reuse: Polymorphism

Polymorphism supports **generic** programming:

```
val id : All 'a ('a -> 'a) =  
  Fn 'a => fn x:'a => x  
  
val compose :  
  All ('a,'b,'c) ('b -> 'c) * ('a -> 'b) -> 'a -> 'c =  
  Fn ('a,'b,'c) =>  
    fn (f:'b->'c, g:'a->'b):'a->'c =>  
      fn x:'a => f(g(x))
```

A **type abstraction** is a function that take types as arguments.

3

Enhancing Reuse: Polymorphism

Polymorphic functions are **instantiated** by applying them to types.

```
val n : int = id[int](3)  
val f : int -> string =  
  compose[int,char,string](to_string,to_char)
```

4

Example: Polymorphic Lists

An explicitly-typed, polymorphic **map** function:

```
val map : All ('a,'b) ('a -> 'b) -> 'a list -> 'b list =  
  Fn ('a, 'b) =>  
    fn f : 'a -> 'b =>  
      fn l : 'a list =>  
        case l  
        of Nil['a] => Nil['b]  
         | Cons['a](h:'a, t:'a list) =>  
           Cons['b](f h, map['a,'b](f)(t))
```

5

Polymorphic Type Inference

ML combines implicit typing and polymorphism, allowing us to write:

```
val id = fn x => x
val compose = fn (f,g) => fn x => f(g(x))
fun map f nil = nil | f (h::t) => map (f h, map f t)
```

Instantiation is performed automatically:

```
val n = id (3)
val f = compose(to_string,to_char)
val l = map succ [1,2,3]
```

6

Two Related Concepts

ML does two things for you:

- Infers missing type information in the most general way possible.
- Inserts implicit type abstractions and instantiations.

We'll consider polymorphism as a language mechanism; we may consider type inference later in the semester.

7

Formalizing Polymorphism

We investigate (Girard's) System F, or (Reynold's) polymorphic λ -calculus, which we will call $\mathcal{L}\{\rightarrow \forall\}$. Here is a grammar:

Category	Item	Abstract	Concrete
Type	τ	$::= t$	t
		$ \text{arr}(\tau_1; \tau_2)$	$\tau_1 \rightarrow \tau_2$
		$ \text{all}(t.\tau)$	$\forall(t.\tau)$
Expr	e	$::= x$	x
		$ \text{lam}[\tau](x.e)$	$\lambda(x:\tau.e)$
		$ \text{ap}(e_1; e_2)$	$e_1(e_2)$
		$ \text{Lam}(t.e)$	$\Lambda(t.e)$
		$ \text{App}[\tau](e)$	$e[\tau]$

Metavariable t ranges over **type variables** and x ranges over **expression variables**.

8

Formalizing Polymorphism

Type abstraction $\text{Lam}(t.e)$ defines a **generic** or **polymorphic** function with **type parameter** t .

Type application or **instantiation** $\text{App}[\tau](e)$ applies the polymorphic function to a specified type.

Polymorphic functions are classified by the **universal type** $\text{all}(t.\tau)$ that determines the type, τ of the result as a function of the argument t .

9

Polymorphic Types

Examples:

- $\forall(t.t \rightarrow t)$.
- $\forall(t.t \text{ list} \rightarrow t \text{ list})$.
- $(\forall(t.t \rightarrow t)) \rightarrow (\forall(t.t \rightarrow t))$.

10

Static Semantics

Static semantics for $\mathcal{L}\{\rightarrow \forall\}$ consists of two judgement forms, τ type, stating that τ is a well-formed type, and $e : \tau$, stating that e is a well-formed expression of type τ .

These two judgements are defined using parametric hypothetical judgements:

$$\mathcal{T} \mid \Delta \vdash \tau \text{ type}$$

and

$$\mathcal{T} \ \mathcal{X} \mid \Delta \ \Gamma \vdash e : \tau$$

11

Static Semantics

\mathcal{T} is a finite set of **type variables** and \mathcal{X} is a finite set of **expression variables**.

Δ is a finite set of **type hypotheses** of the form t type (i.e., t is a well-formed type) for some **type variable** (aka **type name**) t , such that $t \in \mathcal{T}$ and Γ is a finite set of **typing hypotheses** of the form $x : \tau$, where $x \in \mathcal{X}$ and $\Delta \vdash \tau$ type.

(As usual, \mathcal{T} and \mathcal{X} are not explicitly mentioned later since they can be recovered from Δ and Γ .)

12

Type Formation Rules

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \text{arr}(\tau_1; \tau_2) \text{ type}}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \text{all}(t, \tau) \text{ type}}$$

13

Typing Rules

$$\frac{}{\Delta \Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta \Gamma \vdash \text{lam}[\tau_1](x.e) : \text{arr}(\tau_1; \tau_2)}$$

$$\frac{\Delta \Gamma \vdash e_1 : \text{arr}(\tau_2; \tau) \quad \Delta \Gamma \vdash e_2 : \tau_2}{\Delta \Gamma \vdash \text{ap}(e_1; e_2) : \tau}$$

14

Typing Rules

Valid type abstractions:

$$\frac{\Delta, t \text{ type} \quad \Gamma \vdash e : \tau}{\Delta \Gamma \vdash \text{Lam}(t.e) : \text{all}(t, \tau)}$$

In words:

- Add t to the active set of type variables, ensuring that it is not already present.
- Type check body with t being active.
- Assign a polymorphic type to the type abstraction.

15

Typing Rules

Valid type instantiations:

$$\frac{\Delta \Gamma \vdash e : \text{all}(t, \tau') \quad \Delta \vdash \tau \text{ type}}{\Delta \Gamma \vdash \text{App}[\tau](e) : [\tau/t]\tau'}$$

In words:

- Ensure that e is polymorphic.
- Ensure that the type argument τ is valid.
- Instantiate the type of e by substitution.

16

Properties of Typing

Lemma 1 (Regularity)

If $\Delta \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i$ type for each hypothesis $x_i : \tau_i$ in Γ , then $\Delta \vdash \tau$ type

17

Properties of Typing

Lemma 2 (Substitution)

1. If $\Delta, t \text{ type} \vdash \tau' \text{ type}$ and $\Delta \vdash \tau \text{ type}$, then $\Delta \vdash [\tau/t]\tau' \text{ type}$
2. If $\Delta, t \text{ type} \Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau \text{ type}$, then $\Delta [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$
3. If $\Delta \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta \Gamma \vdash e : \tau$, then $\Delta \Gamma \vdash [e/x]e' : \tau'$

The second part of the lemma requires substitution into the context Γ as well as into the term and its type, because the type variable t may occur in any of these positions.

18

Typing Examples

For example,

$$\emptyset; \emptyset \vdash \Lambda(t.\lambda(x:t.x)) : \forall(t.t \rightarrow t),$$

because

$$t \text{ type}; \emptyset \vdash \lambda(x:t.x) : t \rightarrow t,$$

because

$$t \text{ type}; x:t \vdash x : t.$$

19

Typing Examples

Or, in abstract syntax,

$$\emptyset; \emptyset \vdash \text{Lam}(t.\text{lam}[t](x.x)) : \text{all}(t.\text{arr}(t; t)),$$

because

$$t \text{ type}; \emptyset \vdash \text{lam}[t](x.x) : \text{arr}(t; t),$$

because

$$t \text{ type}; x:t \vdash x : t.$$

20

Typing Examples

For example, $\text{int type}; I:\forall(t.t \rightarrow t) \vdash I[\text{int}] : \text{int} \rightarrow \text{int}$, since

- $\text{int type}; I:\forall(t.t \rightarrow t) \vdash I : \forall(t.t \rightarrow t)$, and
- $\text{int type} \vdash \text{int type}$, and
- $[\text{int}/t]t \rightarrow t = \text{int} \rightarrow \text{int}$.

21

Typing Examples

Or, in abstract syntax, $\text{int type}; I:\text{all}(t.\text{arr}(t; t)) \vdash \text{App}[\text{int}](I) : \text{arr}(\text{int}; \text{int})$, since

- $\text{int type}; I:\text{all}(t.\text{arr}(t; t)) \vdash I : \text{all}(t.\text{arr}(t; t))$, and
- $\text{int type} \vdash \text{int type}$, and
- $[\text{int}/t]\text{arr}(t; t) = \text{arr}(\text{int}; \text{int})$.

22

Dynamic Semantics

Main ideas:

- Type abstractions are values (just like ordinary abstractions).
- Type instantiation is an instruction step.

We'll use structural semantics to specify the dynamic semantics of polymorphism.

23

Dynamic Semantics for $\mathcal{L}\{\rightarrow \forall\}$

$$\frac{\overline{\text{lam}[\tau](x.e)} \text{ val}}{\text{ap}(\text{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}$$

$$\frac{e_1 \mapsto e'_1}{\text{ap}(e_1; e_2) \mapsto \text{ap}(e'_1; e_2)}$$

These rules impose a lazy (call-by-name) interpretation, but an eager (call-by-value) interpretation is also possible.

24

Safety

Lemma 3 (Canonical Forms)

Suppose that $e : \tau$ and e val, then

1. If $\tau = \text{arr}(\tau_1; \tau_2)$, then $e = \text{lam}[\tau_1](x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.
2. If $\tau = \text{all}(t.\tau')$, then $e = \text{Lam}(t.e')$ with t type $\vdash e' : \tau'$.

Theorem 4 (Safety)

1. If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.
2. If $e : \sigma$, then either e val or there exists e' such that $e \mapsto e'$.

26

“Deep” Polymorphism

Our model of polymorphism admits **deep** polymorphism:

- Can pass polymorphic functions as arguments and return them as results.
- Can build lists (or other aggregates) of polymorphic functions.
- Can store polymorphic functions in reference cells.

28

Dynamic Semantics for $\mathcal{L}\{\rightarrow \forall\}$

New value:

$$\overline{\text{Lam}(t.e)} \text{ val}$$

New instruction:

$$\overline{\text{App}[\tau](\text{Lam}(t.e))} \mapsto [\tau/t]e$$

New search rule:

$$\frac{e \mapsto e'}{\text{App}[\tau](e) \mapsto \text{App}[\tau](e')}$$

There is no by-name versus by-value distinction for type applications.

25

“Shallow” Polymorphism

ML provides only **prenex**, or **shallow**, polymorphism: $\forall(t_1, \dots, t_n.\tau)$, where τ is not a quantified type.

- A **polytype** σ is either a **monotype** τ , or a quantified polytype $\forall(t.\sigma)$.
- A **monotype** τ is an ML type, possibly involving type variables (e.g., $t \rightarrow t$).

This limitation is necessary to support “full” type inference (no types are ever required.)

The distinction can be formalized (details in Harper).

27

Representing Data Structures

A rich variety of types are **representable** using polymorphism.

- Product (tuple) types.
- Sum (disjoint union) types.
- Natural numbers.
- Lists, streams, trees.

29

Representing Data Structures

To be representable means that

- We can **define** the type in terms of polymorphic types.
- We can **define** the introduction and elimination forms for the types.

Key idea: **active**, rather than **passive**, data.

- Data structures respond to **messages**.
- Elimination forms send messages to the data.

30

Representing Products

Type:

$$\tau_1 \times \tau_2 := \forall (r. (\tau_1 \rightarrow \tau_2 \rightarrow r) \rightarrow r)$$

Pairs:

$$\langle e_1, e_2 \rangle := \Lambda (r. \lambda (x: \tau_1 \rightarrow \tau_2 \rightarrow r. x(e_1)(e_2)))$$

Fst:

$$\text{pr}_1(e) := e [\tau_1] (\lambda (x: \tau_1. \lambda (y: \tau_2. x)))$$

Snd:

$$\text{pr}_2(e) := e [\tau_2] (\lambda (x: \tau_1. \lambda (y: \tau_2. y)))$$

31

Representing Products

Idea: the pair take a **result type** and a **handler** as arguments, and passes the components of the pair to the handler to compute the result.

Check:

$$\begin{aligned} \text{pr}_1(\langle e_1, e_2 \rangle) &= \langle e_1, e_2 \rangle [\tau_1] (\lambda (x: \tau_1. \lambda (y: \tau_2. x))) \\ &\mapsto (\lambda (x: \tau_1. \lambda (y: \tau_2. x)))(e_1)(e_2) \\ &\mapsto [e_1, e_2/x, y]x \end{aligned}$$

This is the correct behavior!

32

Representing Sums

Type:

$$\tau_1 + \tau_2 := \forall (r. (\tau_1 \rightarrow r) \rightarrow (\tau_2 \rightarrow r) \rightarrow r)$$

Injections:

$$\begin{aligned} \text{in}[l](e) &:= \Lambda (r. \lambda (x: \tau_1 \rightarrow r. \lambda (y: \tau_2 \rightarrow r. x(e)))) \\ \text{in}[r](e) &:= \Lambda (r. \lambda (x: \tau_1 \rightarrow r. \lambda (y: \tau_2 \rightarrow r. y(e)))) \end{aligned}$$

Case analysis:

$$\begin{aligned} \text{case } e \{ \text{in}[l](x_1) \Rightarrow e_1 \mid \text{in}[r](x_2) \Rightarrow e_2 \} \\ &:= \\ &e [\rho] (\lambda (x_1: \tau_1. e_1)) (\lambda (x_2: \tau_2. e_2)) \end{aligned}$$

33

Summary

Polymorphism supports generic programming.

Universal types formalize polymorphism.

Polymorphism may be used to encode data structures (and natural numbers – see Harper for details).

34