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.

Enhancing Concision: Type Inference

Implicitly-typed languages allow omission of type information.

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

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

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

Enhancing Reuse: Polymorphism

Polymorphism supports generic programming:

```ml
val id : All 'a u'a -> 'av = 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.

Example: Polymorphic Lists

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

```ml
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 => Nil | Cons(a:b:'a, t:'a list) => Cons(b) fn h, map(a:'b,f(t))
```
Polymorphic Type Inference

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

```latex
val id = fn x => x
val compose = fn uf|gv => fn x => fuguxvv
fun map f nil = nil | f (h::tv) => map (f h, map f t)
```

Instantiation is performed automatically:

```latex
val n = id u3v
val f = composeuto
val l = map succ [1|2|3]
```

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.

Formalizing Polymorphism

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$: := t$</td>
<td>$t$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$: := x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{arr}(\tau_1;\tau_2)$</td>
<td>$\tau_1 \to \tau_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{all}(t,\tau)$</td>
<td>$\forall \tau$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{lam}(\tau)[x.e]$</td>
<td>$\lambda x : \tau . e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{app}(e_1;e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{lam}(e)$</td>
<td>$\Lambda e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{app}(e)$</td>
<td>$e \tau$</td>
</tr>
</tbody>
</table>

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

Polymorphic Types

Examples:

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

Static Semantics

Static semantics for $L(\to \forall)$ consists of two judgement forms, $\tau$ type, stating that $\tau$ is a well-formed type, and $e : \tau$, stating that $e$ is a well-formed expression of type $\tau$.

These two judgements are defined using parametric hypothetical judgements:

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

and

$$
T \ A \mid \Delta \Gamma \vdash e : \tau
$$
Static Semantics

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

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

Type Formation Rules

\[ \Delta, t \vdash t : \tau \]
\[ \Delta \vdash \text{arr}(t_1, t_2) : \tau \]
\[ \Delta \vdash \text{all}(\tau, \tau) : \tau \]

Typing Rules

**Valid type abstractions:**
\[ \Delta, t \vdash e : \tau \]
\[ \Delta \vdash \text{Lam}(e) : \text{all}(\tau, \tau) \]

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.

Typing Rules

Valid type instantiations:
\[ \Delta, \tau \vdash e : \tau \]
\[ \Delta \vdash \text{App}(e, t) : \tau \]

In words:
- Ensure that \( e \) is polymorphic.
- Ensure that the type argument \( \tau \) is valid.
- Instantiate the type of \( e \) by substitution.

Lemma 1 (Regularity)

If \( \Delta \vdash e : \tau \), and if \( \Delta \vdash \tau_i \) type for each hypothesis \( x_i : \tau_i \) in \( \Gamma \), then \( \Delta \vdash \tau \) type.
Properties of Typing

Lemma 2 (Substitution)
1. If $\Delta, t \vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t] \tau'$ type
2. If $\Delta, t \vdash \sigma' : \tau'$ and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t] \sigma' : \tau/t \tau'$
3. If $\Delta, x : \tau \vdash \sigma' : \tau'$ and $\Delta \vdash \tau$ type, then $\Delta \vdash [x/x] \sigma' : \tau$

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

Typing Examples

Or, in abstract syntax,

$\emptyset, \emptyset \vdash \lambda \mathit{t}(\lambda \mathit{x} \mathit{t})(\mathit{x} : \mathit{t}) : \forall \mathit{U} \mathit{t}. \mathit{arr}(\mathit{t} : \mathit{t})$,

because

$t \vdash \lambda \mathit{t}(\lambda \mathit{x} \mathit{t})(\mathit{x} : \mathit{t}) : \mathit{arr}(\mathit{t} : \mathit{t})$,

because

$t \vdash x : \mathit{t}$.

Typing Examples

Or, in abstract syntax, $\mathit{iat} \vdash \forall \mathit{U} t. \mathit{arr}(\mathit{t} : \mathit{t}) \vdash \forall \mathit{U} t. \mathit{arr}(\mathit{t} : \mathit{t})$:

- $\mathit{iat} \vdash \forall \mathit{U} t. \mathit{arr}(\mathit{t} : \mathit{t}) \vdash \forall \mathit{U} t. \mathit{arr}(\mathit{t} : \mathit{t})$, and
- $\mathit{iat} \vdash \forall \mathit{U} t. \mathit{arr}(\mathit{t} : \mathit{t})$,
- $\mathit{iat} \vdash \mathit{arr}(\mathit{t} : \mathit{t}) = \mathit{arr}(\mathit{t} : \mathit{t})$.

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.
Dynamic Semantics for $L^{\rightarrow\forall}$

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

Safety

Lemma 3 (Canonical Forms)
Suppose that $e : \tau$ and $e \text{ val}$, then

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

Theorem 4 (Safety)
1. If $e : \sigma$ and $e \Rightarrow e'$, then $e' : \sigma$.
2. If $e : \sigma$, then either $e \text{ val}$ or there exists $e'$ such that $e \Rightarrow e'$.

“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.

”Shallow” Polymorphism

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

- A polytype $\sigma$ is either a monotype $\tau$, or a quantified polytype $\forall(t. \sigma)$.
- A monotype $\tau$ 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).

Representing Data Structures

A rich variety of types are representable using polymorphism.

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

---

**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:

\[ \text{pr}_1(e_1,e_2) = \langle e_1,e_2 \rangle \Rightarrow e_1 \\text{and } \text{pr}_2(e_1,e_2) = \langle e_1,e_2 \rangle \Rightarrow e_2 \]

This is the correct behavior!

---

**Representing Sums**

Type:

\[ r_1 \cdot r_2 : = \forall (r : r_1 \rightarrow r_2 \rightarrow r) \]

Injections:

\[ \text{in}[l] : = \lambda x.\lambda y.\lambda z.\lambda t.\lambda r.\lambda (x,y).r(x,y) \]

\[ \text{in}[r] : = \lambda x.\lambda y.\lambda z.\lambda t.\lambda r.\lambda (x,y).r(y,x) \]

Case analysis:

\[ \text{case} e \{ \text{in}[l] \Rightarrow e_1 \mid \text{in}[r] \Rightarrow e_2 \} : = e \{ [t] \Rightarrow e_1 \mid [t] \Rightarrow e_2 \} \]

---

**Summary**

Polymorphism supports generic programming.

Universal types formalize polymorphism.

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