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 reuse: one version for each type.

Enhancing Concision: Type Inference

Implicitly-typed languages allow omission of type information.

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

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

```
val axby : 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:

```
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.

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))
```
**Polymorphic Type Inference**

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

\[
\begin{align*}
\text{val } \text{id} &= \text{fn } x \Rightarrow x \\
\text{val } \text{compose} &= \text{fn } (f,g) \Rightarrow \text{fn } x \Rightarrow f(g(x)) \\
\text{fun } \text{map } f \text{ nil} &= \text{nil} | \text{f} (\text{h};t) \Rightarrow \text{map} (f, \text{h}; \text{map} f t)
\end{align*}
\]

Instantiation is performed automatically:

\[
\begin{align*}
\text{val } n &= \text{id} (3) \\
\text{val } f &= \text{compose} (\text{to_string},\text{to_char}) \\
\text{val } l &= \text{map succ} \ [1,2,3]
\end{align*}
\]

**Polymorphic MinML**

Additions to the abstract syntax:

\[
\begin{align*}
\text{Types} & \quad \tau ::= t \mid \forall t \Gamma \tau \\
\text{Expressions} & \quad e ::= \text{Fun } t \end{align*}
\]

\[
\begin{align*}
\text{Values} & \quad v ::= \text{Fun } t \\
\end{align*}
\]

The type variable \( t \) is bound in the body \( \tau \) of the polymorphic type \( \forall (\tau) \).

Notation in the literature:

\[
\begin{align*}
\forall t.e & \quad \text{for Fun } t \end{align*}
\]

\[
\begin{align*}
\lambda x:\tau.e & \quad \text{for fn } x:\tau \end{align*}
\]

**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 will not have time to consider type inference.

**Polymorphic Types**

Examples:

\[
\begin{align*}
\forall t \rightarrow t & \\
\forall t \rightarrow \text{list} & \\
(\forall t \rightarrow t) \rightarrow (\forall t \rightarrow t)
\end{align*}
\]

**Type Formation**

Valid type variables and base types:

\[
\begin{align*}
\frac{t \in \Delta}{\Delta \vdash t \text{ type}} \\
\frac{}{\Delta \vdash \text{int type}} \\
\frac{}{\Delta \vdash \text{bool type}}
\end{align*}
\]

Valid function types:

\[
\begin{align*}
\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}}
\end{align*}
\]

**Context Formation**

Valid quantified types:

\[
\begin{align*}
\frac{\Delta \vdash \tau \text{ type} \quad \forall t \not\in \Delta}{\Delta \vdash \forall (t) \text{ type}}
\end{align*}
\]

Valid typing contexts:

\[
\begin{align*}
\frac{\Delta \vdash \Gamma (x) \text{ ok} \quad (\forall x \in \text{dom}(\Gamma))}{\Delta \vdash \Gamma \text{ ok}}
\end{align*}
\]
Typing Examples

For example, \( \forall t \vdash t : \text{int} \rightarrow \text{int} \), since

- \( \forall t \vdash t \rightarrow t \rightarrow t \), and
- \( \emptyset \vdash \text{int} \rightarrow \text{int} \), and
- \( \{ \text{int}/t \} t \rightarrow t = \text{int} \rightarrow \text{int} \).

Typing Rules

Valid type instantiations:

\[
\frac{\Delta \vdash \tau \text{ type} \quad \Gamma \vdash \Delta \vdash \epsilon : \forall \tau}{\Gamma \vdash \Delta \vdash \epsilon[\tau] : \forall \tau \rightarrow \tau'}
\]

In words:

- Ensure that the type argument \( \tau \) is valid.
- Ensure that \( \epsilon \) is polymorphic.
- Instantiate the type of \( \epsilon \) by substitution.

Typing Examples

For example,\
\[
\emptyset \vdash \text{fun } (x : t) \rightarrow \text{fun } x \text{ end : } \forall (t \rightarrow t),
\]
because\
\[
\emptyset \vdash t \quad \text{fun } f (x : t) \rightarrow \text{fun } x \text{ end : } t \rightarrow t,
\]
because\
\[
x : t \vdash (x : t) \rightarrow (x : t).
\]

An Example

Here's \text{map} written in PolyMinML extended with lists:

Fun \( t \) in
Fun u in
fun _ (f : t \rightarrow u) : t list \rightarrow u list is
fun loop (l : t list) : u list is
listcase l
of nil => nil
| cons(x,l) => cons (f x, loop l) end
end
end
end
end
Dynamic Semantics

Main ideas:

• Type abstractions are values (just like ordinary abstractions).

• Type instantiation is an instruction step.

We'll use SOS to specify the dynamic semantics of polymorphic MinML.

Type Safety

Theorem 1
If $e : \tau$, then either $e$ is a value or there exists $e' : \tau$ such that $e \mapsto e'$.

Proof sketch:

• Prove preservation by induction on evaluation.

• Prove progress by induction on typing.

• A closed value of type $\forall(\tau)$ is a type abstraction: $\text{Fun } in e \text{ end}$.

“Deep” Polymorphism

PolyMinML 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.

Dynamic Semantics

New instruction:

$$\text{Fun } in e \text{ end } [\tau] \mapsto (\tau/t)e$$

New search rule:

$$e [\tau] \mapsto e'[\tau]$$

“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 $\forall(\sigma)$.

• A monotype $\tau$ is a MinML type, possibly involving type variables (e.g., $t \mapsto t$).

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

Polymorphic References

Consider the type $T = \forall((t \mapsto t)\text{ref})$.

• Elements are polymorphic functions $f$.

• Instantiation of $f$ to $\tau$ yields a value of type $\tau \mapsto \tau\text{ref}$.

Type of polymorphic functions yielding reference cells.
Polymorphic References

Example function \( f \) of this type:

\[
\text{Fun } t \in \text{ref} \ (\text{fun } x : t) : t \text{ is } x \text{ end} \end{equation}

This is a value.

Each time we instantiate \( f \) we get a new reference cell.

- \( f [\text{int}] : (\text{int} \rightarrow \text{int}) \text{ ref} \)
- \( f [\text{bool}] : (\text{bool} \rightarrow \text{bool}) \text{ ref} \)

Polymorphic References

Now consider the type \( U = (\forall t \rightarrow t) \text{ ref} \).

- Elements are reference cells holding a polymorphic function.
- This is not a prenex polymorphic type.

Type of reference cells containing polymorphic functions.

Polymorphic References

Example value \( x \) of this type:

- \( \text{ref} (\text{Fun } t \in \text{ref} \ (\text{fun } x : t) : t \text{ is } x \text{ end}) \)
- A single reference cell containing the polymorphic identity.

Polymorphic References

Consider the following example:

\[
\text{let val } r : (\forall t \rightarrow t) \text{ ref } = \text{ref} \ (\text{fn } x \Rightarrow x) \\text{ in} \r[\text{int}] := \text{succ} ; (!r)(\text{true}) \text{ end}
\]

In early versions of ML this was type correct, yet it’s clearly not safe!

Polymorphic References

With explicit polymorphism the problem is revealed.

What is \( r \)?

- A polymorphic function yielding a reference cell? Or ...
- A reference cell containing a polymorphic function?

Version 1: a polymorphic function yielding reference cells.

\[
\text{let val } r = \text{fn t in ref} (\text{fn x : t} \Rightarrow x) \\text{ in} \ r[\text{int}] := \text{succ} ; (! (r[\text{bool}]))(\text{true}) \text{ end}
\]

Each instance of \( r \) yields a fresh reference cell.

- The set and the get are performed on distinct cells.
- Sound, but not particularly useful.
Polymorphic References

Version 2: a reference cell containing a polymorphic function.

let val r = ref (Fn t in Fn x:t ⇒ x)
in r := succ ; ![r] [bool](true) end

This is type incorrect.

- succ has type int⇒int.
- Assignment requires value of type ∀tTτe→ τf→ tU→ tU.

Unsound, and duly rejected by the type checker.

Representing Data Structures

A rich variety of types are representable in PolyMinML.

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

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:

split (v1,v2) as (x,y) in e' end = (v1,v2) ![v'] ![λx:τ1,λy:τ2.e']
⇒ (λx:τ1,λy:τ2.e')(v1)(v2)
⇒ (v1,v2/x,y)e'

This is the correct behavior!

Representing Sums

Type:

τ1*τ2 ≔ ∀(τ1→τ2→t)→t

Pairs:

(v1,v2) ≔ λt.λh:τ1→τ2→t.h(v1)(v2)

Split:

split e as (x,y) in e' end = e ![v'] ![λx:τ1,λy:τ2.e']
Representing Natural Numbers

Type:

\[ \text{nat} := \forall ((t \rightarrow t) \rightarrow (t \rightarrow t)) \]

Numbers:

\[ \text{zero} := \lambda t.\lambda s.t \rightarrow \lambda b.t.b \]
\[ \text{succ}(n) := \lambda t.\lambda s.t \rightarrow \lambda b.t.s(n \, t \, (s \, (b))) \]

Some arithmetic:

\[ m + n := n \, \text{nat} \,(\lambda p.\text{nat} \, \text{succ}(p)) \, (m) \]
\[ m \times n := n \, \text{nat} \,(\lambda p.\text{nat} \, p \times m) \, (\text{zero}) \]
\[ m^n := n \, \text{nat} \,(\lambda p.\text{nat} \, p \times m) \, (\text{succ}(\text{zero})) \]

Summary

Polymorphism supports generic programming.

Universal types formalize polymorphism.

Polymorphism may be used to encode data structures.