Varieties of Polymorphism

Universal polymorphism is obtained when a function works uniformly on a range of types.
Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types and may behave in unrelated ways for each type.


Forms of Universal Polymorphism

- Parametric polymorphism (generics)
- ML-style, based on type parameters
- Focus of this lecture
- Inclusion polymorphism (subtyping)
- OO-style based on subtype inclusion
- Will study this Friday

Forms of Parametric Polymorphism

- Harper distinguishes between “first-class” and “second-class” forms of parametric polymorphism
- ML supports only “second-class” parametric polymorphism because of limitations in type-inference
- Supporting first-class versions is easy if you don’t care about type-inference

First vs Second Class

First-class
\[ \tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid t \mid \forall \tau(\tau) \]
Example: \( \forall \tau(t \rightarrow t) \rightarrow \forall \tau'(t' \rightarrow t') \)

Second-class
\[ \sigma ::= \tau \mid \forall \tau(\sigma) \]
\[ \tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid t \]
Example: \( \forall \tau(\forall \tau'((t \rightarrow t) \rightarrow (t' \rightarrow t'))) \)
Second Class Polymorphism

Second-class or “prenex” polymorphism the \( \forall \) are all at the front of the type

\[
\sigma ::= \tau \mid \forall t(\sigma) \quad \text{(polytypes)}
\]

\[
\tau ::= \text{int} \mid \tau_1 \to \tau_2 \mid t \quad \text{(monotypes)}
\]

Polytypes: \( \tau, \forall t(\tau), \forall t(\forall t'(\tau)) \), int

Monotypes: \( t, \text{int}, t \to \text{int} \ldots \)

PolyMinML

\[
\begin{align*}
\text{Polytypes} & \quad \sigma ::= \tau \mid \forall t(\sigma) \\
\text{Monotypes} & \quad \tau ::= \ldots \mid t \\
\text{Expressions} & \quad e ::= \ldots \mid \text{Fun} \text{in} e \mid \text{inst}(e, \tau) \\
\text{Values} & \quad v ::= \ldots \mid \text{Fun} \text{in} e
\end{align*}
\]

- Supports second-class parametric polymorphism
- Only instantiate polytypes with monotypes

Dynamic Semantics

\[
\text{inst}(\text{Fun} \text{in} e, \tau) \mapsto \{\tau/\{t\}e
\]

\[e \mapsto e' \]

\[
\text{inst}(e, \tau) \mapsto \text{inst}(e', \tau)
\]

Static Semantics

Must modify judgment forms to account for free type variables occurring in types during type checking

\[
\Delta \vdash \tau \text{ ok} \quad \tau \text{ is a well-formed type in } \Delta \\
\Gamma \vdash e : \sigma \quad e \text{ is a well-formed expression of type } \sigma \text{ in } \Gamma \text{ and } \Delta
\]

Static Semantics

Must modify judgment forms to account for free type variables occurring in types during type checking

\[
\Delta \vdash \tau \text{ ok} \quad \tau \text{ is a well-formed type in } \Delta \\
\Gamma \vdash \Delta : \sigma \quad \Delta \text{ is a well-formed type in } \Gamma \text{ and } \Delta \\
\Gamma \vdash e : \sigma \quad e \text{ is a well-formed expression of type } \sigma \text{ in } \Gamma \text{ and } \Delta
\]

Typo should be \( \sigma \)

Well-formed Type

\[
\begin{align*}
\Delta \vdash \text{int ok} & \quad \Delta \vdash \text{bool ok} & \quad t \in \Delta \\
\Delta \vdash t \text{ ok} & \quad \Delta \vdash \text{t ok} \\
\Delta \vdash \tau_1 \text{ ok} & \quad \Delta \vdash \tau_2 \text{ ok} \\
\Delta \vdash \tau_1 \to \tau_2 \text{ ok} & \quad \Delta \cup \{f\} \vdash \sigma \text{ ok} & \quad f \notin \Delta \\
\Delta \vdash \forall(\sigma) \text{ ok} & \quad \Delta \vdash \Gamma(x) \text{ ok} & \quad (\forall x \in \text{dom}(\Gamma)) \\
\Delta \vdash \Gamma \text{ ok}
\end{align*}
\]
Well-Formed Expressions

Well-formed expressions maintain the following invariant:
\[ \Gamma \vdash \Delta \vdash e : \sigma \quad t \notin \Delta \]
\[ \Gamma \vdash \Delta \vdash \text{Fun } t e : \forall \nu(\sigma) \]
\[ \Gamma \vdash \Delta e : \forall \nu(\sigma) \quad \Delta \vdash \tau \text{ ok} \]
\[ \Gamma \vdash \Delta \vdash \text{inst}(e, \tau) : \{\tau/\nu\} \sigma \]

Remaining rules maintain the following invariant:
\[ \Delta \vdash \text{ok}, \Delta \vdash \sigma \text{ ok}, \text{FV}(e) \subseteq \text{dom}(\Gamma), \text{and FTV}(e) \subseteq \Delta \]

Example: Polymorphic Compose

\[ \forall \nu \forall \nu(\nu : (u \mapsto \nu) \mapsto (t \mapsto \nu) \mapsto (t \mapsto \nu)) \]

Fun t in
Fun u in
Fun v in

\[ \text{fun } (f : u \mapsto v) : (t \mapsto u) \mapsto (t \mapsto v) \] is

\[ \text{fun } (g : t \mapsto u) : t \mapsto v \] is

\[ \text{fun } (x : t) : v \text{ is apply}(t, \text{apply}(g, x)) \]

Type Soundness

Similar to Substitution Lemma

Lemma 20.1 (Instantiation)
If \[ \Gamma \vdash \Delta \vdash e, \] where \( t \notin \Delta \), and \( \Delta \vdash \tau \text{ ok} \), then \( \{\tau/\nu\} \Gamma \vdash \{\tau/\nu\} e : \{\tau/\nu\} \nu \).

Lemma 20.2 (Canonical Forms)
If \( v : \forall \nu(\sigma) \), then \( v = \text{Fun } t e \) for some \( t \) and \( e \) such that \( \{t/\nu\} e : \sigma \).

Proofs are all similar to previous proofs just with more tedium!

First-Class Polymorphism

• Easy to obtain just relax distinction between polytypes and monotypes
• Zero changes to the proof or operational semantics
• Not supported for many reasons
  – Interacts badly with type-inference
  – Does not allow type-erasure in opsem

Effects/Erasure and Polymorphism

• The expressions \( \text{Fun} \) and \( \text{inst} \) similar to roll and unroll
  – No real need on a executing machine
  – Just to make type checking and type-soundness easier
• In second-order system with value restriction can be erased from program
• ML compilers use type-erasure during compilation

Value Restriction

Consider \( v : \sigma \)
\( v \) must be of the form
\[ \text{Fun } t_1 \text{ in } \text{Fun } t_2 \text{ in } ... \text{Fun } t_n \text{ in } e \]
Value restriction only allows type-abstraction over values so with the value restriction we know that \( "e" \) is also a value
If it is a value than it is safe to remove all the \text{Fun}
Example: Value Restriction

Consider program below

```ml
let
val r : All 'a (('a -> 'a) ref) =
  Fun 'a in ref (fn x:'a => x) end
in
r[int] := (fn x:int => x+1) ; (!r[bool]))(true)
end
```

the program is sound because there is a unique reference created with each instantiation of r.

Example: Value Restriction

Consider erased version below

```ml
let
val r : ('a -> 'a) ref = ref (fn x:'a => x)
in
r := (fn x:int => x+1) ; (!r)(true)
end
```

the program is unsound because there is just one reference created with each instantiation of r. SML rejects the above.

Polymorphism and Abstraction

- Parametric polymorphism can be used to enforce data abstraction
- Parametricity theorem lets us infer information about a value just by inspecting it's type
  - Similar to canonical forms lemmas
  - Gives us a semantic description of a value based on it's type

Example: $\forall t(t \rightarrow t)$

Consider function $f$ the type $\forall t(t \rightarrow t)$
assuming we only care about "interesting" functions (those that terminate) how can functions of that type behave?

All functions of that type must simply return their argument, i.e. they all behave as the identity function.

Semantics vs. Syntax

Note parametricity theorem states any function must "act like" the identity function it may "look" different

- $(\text{fn } x \Rightarrow x)$
- $(\text{fn } x \Rightarrow ((\text{fn } f \Rightarrow f) \ (\text{fn } y \Rightarrow y)) \ x))$
- $(\text{fn } x \Rightarrow (1+2:x))$

Compare to canonical forms lemmas which states values must be a specific piece of syntax

Other Theorems

$\forall t(t)$

No interesting values of this type

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

Interesting functions only examine the "spine" of the list
A Simple Abstraction Theorem

Consider N: \( \forall t (t \to (t \to t) \to t) \)

Interesting functions of this form can only apply their first and second arguments.

\[
\begin{align*}
\text{Fun} & \in \text{fn} \mathbb{Z} : \text{t \to t \to t} \equiv \mathbb{Z} \\
\text{Fun} & \in \text{fn} \mathbb{Z} : \text{t \to t \to t} \equiv \text{s}(\mathbb{Z}) \\
\text{Fun} & \in \text{fn} \mathbb{Z} : \text{t \to t \to t} \equiv \text{s}(\text{s}(\mathbb{Z}))
\end{align*}
\]

Formal Definition of Parametricity

Define a notion of equivalence of expressions indexed by type

\[
\begin{align*}
\sigma & ::= \tau | \text{all } t \in \sigma \\
\tau & ::= \text{bool} \mid \text{int} \mid \tau_1 \to \tau_2 \\
& \equiv_{\exp} v', \sigma' \iff (\exists v, v' : v \equiv v' \land v \equiv_{\exp} v', \sigma')
\end{align*}
\]

Definition of Parametricity (cont.)

Define a notion of equivalence of closed values indexed by type

\[
\begin{align*}
v & \equiv_{\exp} v' : \text{bool} \iff (v = v' = \text{true}) \lor (v = v' = \text{false}) \\
v & \equiv_{\exp} v' : \text{int} \iff (v = v') \\
v & \equiv_{\exp} v' : \tau_1 \to \tau_2 \iff (\forall v, v', (v \equiv_{\exp} v') \Rightarrow (\text{apply}(v, v') \equiv_{\exp} \text{apply}(v', v')) : \tau_2)
\end{align*}
\]

Definition of Parametricity (cont.)

Interesting case is definition of “all”

\[
\begin{align*}
v & \equiv_{\exp} v' : \text{all } t \in \sigma \iff \\
& \forall \tau, \tau', \text{R} : \tau \leftrightarrow \tau'. (\forall v_1, v_2, (v_1 \equiv_{\exp} v_2 : \tau) \Rightarrow (v_1 \text{ R} v_2) \Rightarrow \\
& v'[\tau] \equiv_{\exp} \text{apply}(v', v') : \sigma)
\end{align*}
\]

Almost arbitrary relation

Parametricity Theorems

Theorem 20.6 (Parametricity)

If \( \tau : \sigma \) is a closed expression, then \( v \equiv_{\exp} v : \tau \).

Theorem 20.7

If \( f : \forall t \equiv \text{id} : \forall t \equiv \text{id} \) is an interesting value, then \( f \equiv_{\exp} \text{id} : \forall t \equiv \text{id} \), where \( \text{id} \) is the polymorphic identity function.

Summary

- Many different forms of “polymorphism”
- Parametric polymorphism comes in both first-class and second-class forms
- Parametricity theorems basis for formal claims of data-abstraction