Complaints about Strong Typing

- Types get in the way
  - Too obtrusive: too many type annotations
  - Too restrictive: types inhibit code re-use

```
val compose = fn (g : real->string) : ((int->real)->(int->string)) =>
  fn (f : int->real) : (int->string) =>
  fn (x:int):string => g(f(x))
```

Improving Matters

- Polymorphism: generic functions

```
val compose = fn (g : 'b->'c) : (('a->'b)->('a->'c)) =>
  fn (f : 'a->'b) : ('a->'c) =>
  fn (x:'a):'c => g(f(x))
```

- Implicit typing: automatically inferred annotations

```
val compose = fn g =>
  fn f =>
  fn x => g(f(x))
```

Brands of Polymorphism

- Parametric polymorphism (today's topic)
  - Generic code
  - Algorithm stays the same even when types differ.

- Ad-hoc polymorphism (overloading)
  - Different code runs depending on types
  - Choice may be compile-time (overloading) or run-time (dynamic dispatch/"polymorphism" in C++)

Polymorphism

November 27, 2001
CS 131: Programming Languages
An SML Puzzle

- Consider the definition

```sml
fun id x = x
```

- SML would say that

```sml
id : 'a -> 'a
```

meaning that, for any type 'a, if id is given a value of that type then the return value (if any) will also have the same type.

```sml
val x : int*real = (id 3, id 4.0)
```

An SML Puzzle

- Now consider the definition

```sml
fun apply(f:'a ->'a, x:'a) = f(x)
```

- SML is perfectly happy to then compile

```sml
fun increment x = x+1
val y = apply(increment, 3)
```

even though the first argument to apply clearly isn't polymorphic. What's going on?

An SML Puzzle

- Furthermore, although the code

```sml
let
  val id : 'a -> 'a = (fn x:'a => x)
  fun doit(f:'a ->'a) = (f 3, f 4.0)
in
  doit(id)
end
```

may look reasonable, SML rejects the definition of the function doit. Why?

Solution

- We need to distinguish between the two possible meanings of 'a -> 'a.
  - A function mapping values of type 'a to values of type 'a, for any type 'a.
  - A function mapping values of type 'a to values of type 'a, for some fixed but unspecified type 'a.
- Traditionally distinguished by writing

```sml
\forall 'a. 'a->'a
```

vs.

```sml
'a -> 'a
```

- Universally quantified types are called \textit{polymorphic} types.
Understanding SML Types

• When considering the types of variables previously-defined via val or fun, types are implicitly assumed to start with universal quantifiers for every type variable appearing
  - The type 'a->'a of id is "really" \( \forall 'a. 'a->'a \)
• For all other types, no universal quantifier is assumed.
  - The type 'a->'a of f in apply is "really" 'a->'a
  - The type of apply, however, is "really" \( \forall 'a. (('a->'a) * 'a) -> 'a \)
• Consequence: only prenex polymorphism

Views of Parametric Polymorphism

• Consider the code
  ```
  fun id x = x
  val x : int*real = (id 3, id 4.0)
  ```

How does this get implemented?

Boxed View

• The function id simultaneously has type int->int, and type real->real, and an infinite number of other types, all of which are summarized as \( \forall 'a. 'a->'a \).
• So the same piece of code is getting called twice, and it's doing the same thing.
  ```
  fun id x = x
  val x : int*real = (id 3, id 4.0)
  ```

Consequences?
Template View

• If we had one copy of the code for id for each use of the function, we wouldn't need polymorphism.

```plaintext
fun id1 (x:int):int = x
fun id2 (x:real):real = x
val x : int*real = (id1 3, id2 4.0)
```

• So, being able to write the function once is just a convenience, and the type ∀_'a. 'a→'a means that for any use of the function we can (in the compiler) generate a specific version of the code for that type.

```plaintext
fun id x = x
val x : int*real = (id 3, id 4.0)
```

Template View

• Note: in contrast to C++ templates, ML's polymorphic functions can always be typechecked in isolation.
  – Once we know its type, we can tell whether any use of the function is ok.
  – Don't have to first expand out the definition, plug in the arguments, and check that the resulting code compiles.
    • get error messages earlier.

Consequences?

Type Parameter View

• The type ∀_'a. 'a→'a for id means that the it needs a type '_a as an implicit (run-time) argument.

• So, what is really happening is something like

```plaintext
fun id ('a:TYPE) (x:'a) = x
val x : int*real = (id (int)(3), id (real)(4.0))
```

where in principle, the (single) implementation for id could look at the type and modify its behavior appropriately.
  – e.g., change the calling convention.
Consequences?

SML Polymorphism is...

• **Implicit**
  - All type functions and applications are automatically filled in during type inference.

• **Predicative**
  - Cannot apply a polymorphic function to a polymorphic type

• **Shallow/Prenex**
  - Universal quantifiers in a type must come first. (Hence cannot pass polymorphic functions as arguments.)

• **Nonrecursive**: Permits
  - polymorphic (recursive functions)
  - but not
  - recursive (polymorphic functions)

Polymorphism and Refs

• Should this code typecheck? Why or why not?

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

- "A variable definition may not be polymorphic unless the definition is a syntactic value."

\[
\begin{align*}
\text{val } x : \text{'a list } & = [ ] \quad \text{ok} \\
\text{val } y : \text{int list ref } & = \text{ref [ ] } \quad \text{ok} \\
\text{val } y : \text{'a list ref } & = \text{ref [ ] } \quad \text{not ok}
\end{align*}
\]

Which Polymorphism Implementation Does SML Use?

- No way to tell!
  - In part because of the value restriction.
  - In fact, there are ML compilers which use each of these strategies for compiling polymorphism.
  - For fancier versions of polymorphism (e.g., allowing polymorphic recursion) then the template-based approach wouldn't suffice, however.

Parametricity

- Consider SML without side-effects or infinite loops.

  - Suppose \( f \) has type \( \forall \text{'a}. \text{'a list } \rightarrow \text{'a list} \)
  - What can \( f \) be?
Parametricity

- If \( f \) has type \( \forall \alpha. \ 'a\ list \rightarrow 'a\ list \)
  - \( f \) could be an identity function
  - \( f \) could always return \text{nil}
  - \( f \) could return a fixed sub-list of its argument
  - \( f \) could return a permutation of its argument
  - The function \text{cannot} depend on the elements of the list, only on its structure.
  - The function \text{cannot} return elements in the output list that were not in the input list.
  - The function \text{cannot} work differently depending on the type of the list argument.

Parametricity

- Informally, a polymorphic function is said to be \textit{parametric} if its behavior is independent of its type argument.
  - i.e., same algorithm for all type instances.
- This can be elegantly formalized (but I won't)
  - “Related arguments yield related results”
  - Application: TAL and callee-save registers