Computer Science 131
Programming Languages

October 26, 2000
Parametric Polymorphism
Complaints about Strong Typing

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

```haskell
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

```plaintext
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

```plaintext
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
  - Different code runs depending on types
  - Choice may be compile-time or run-time
NQSML + Polymorphism

\[ v ::= n \mid tt \mid ff \mid \ldots \]
| \( fn \ (x:t) \Rightarrow e \)
| \( Fn \ \alpha \Rightarrow e \)

\[ e ::= v \mid e_1 + e_2 \mid e_1 < e_2 \]
| \( if \ e_1 \ then \ e_2 \ else \ e_3 \)
| \( x \mid let \ x \ be \ e_1 \ in \ e_2 \mid \ldots \)
| \( e_1(e_2) \mid e[t] \)

\[ t ::= \text{Int} \mid \text{Bool} \mid \ldots \]
| \( t_1 \rightarrow t_2 \)
| \( \alpha \mid \forall \alpha.t \)
Evaluation Rules

\[
\begin{align*}
e_1 & \rightarrow e_1' \\
(e_1 (e_2)) & \rightarrow (e_1')(e_2) \\

e_2 & \rightarrow e_2' \\
v_1 (e_2) & \rightarrow v_1 (e_2')
\end{align*}
\]

\[
(f n (x : t) \Rightarrow e_1) v_2 \rightarrow e_1 [x \rightarrow v_2]
\]

\[
\begin{align*}
e & \rightarrow e' \\
e(t) & \rightarrow e'(t)
\end{align*}
\]

\[
(F n \ x \Rightarrow e)(t) \rightarrow e[x \rightarrow t]
\]
Examples

- Polymorphic Identity
  
  Define \( \text{id} \) to be \( \text{Fn} \ \alpha \Rightarrow (\text{fn} \ (x:\alpha) \Rightarrow x) \)

- Then
  
  \[
  \begin{align*}
  \text{id} & : \forall \alpha. (\alpha \rightarrow \alpha) \\
  \text{id[Int]} & : \text{Int}\rightarrow\text{Int} \\
  \text{id[Int]}(3) & : \text{Int}
  \end{align*}
  \]
Examples

• Define \texttt{compose} by

\[
\begin{align*}
\text{Fn } \alpha & \Rightarrow \text{Fn } \beta \Rightarrow \text{Fn } \gamma \\
\text{fn } (g : \beta \rightarrow \gamma) & \Rightarrow \\
\text{fn } (f : \alpha \rightarrow \beta) & \Rightarrow \\
\text{fn } (x : \alpha) & \Rightarrow g(f(x))
\end{align*}
\]

• Then

\[
\begin{align*}
\text{compose} & : \forall \alpha . \forall \beta . \forall \gamma . (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \\
\text{compose[Int, Real, String]} & : \\
(\text{Real} \rightarrow \text{String}) & \rightarrow (\text{Int} \rightarrow \text{Real}) \rightarrow \\
(\text{Int} \rightarrow \text{String}) &
\end{align*}
\]
Typing Rules

• Need a notion of "well-formed types"
  - Requires no unbound type variables
• Let $T$ stand for set of active type variables.

\[
\begin{align*}
T \vdash \text{Int ok} & \quad T \vdash \text{Bool ok} & \quad \alpha \in T \\
T \vdash t_1 \text{ ok} & \quad T \vdash t_2 \text{ ok} & \quad T \vdash \alpha \in T \\
T \vdash t_1 \rightarrow t_2 \text{ ok} & \quad T \vdash \forall \alpha . t \text{ ok}
\end{align*}
\]
Typing Rules

• Determining whether an expression is well-typed now requires
  – A typing context $\Gamma$
  – Set of active type variables $T$
• Rules from before just carry $T$ around

\[
\Gamma, T \vdash n : \text{Int}
\]

\[
\Gamma, T \vdash e_1 : t_2 \rightarrow t \quad \Gamma, T \vdash e_2 : t_2
\]

\[
\Gamma, T \vdash e_1(e_2) : t
\]
New Typing Rules

\[
\begin{align*}
\Gamma, T \cup \{\alpha\} &\vdash e : t \quad \alpha \notin T \\
\Gamma, T &\vdash \text{Fn } \alpha \Rightarrow e : \forall \alpha . t
\end{align*}
\]

\[
\begin{align*}
\Gamma, T &\vdash e_1 : \forall \alpha . t_1 \quad \Gamma, T &\vdash t_2 \text{ ok} \\
\Gamma, T &\vdash e_1(t_2) : t_1[\alpha \rightarrow t_2]
\end{align*}
\]
Type Soundness

• Proofs of Type Preservation and Progress go through for this language
  – Even with refs, exceptions, continuations, ...
Polymorphism and Refs

• Suppose we have \texttt{ref} cells

• Consider the type $\forall \alpha. \left( (\alpha \rightarrow \alpha) \text{ Ref} \right)$
  - Values of this type are polymorphic functions
  - Given $\alpha$, return a ref cell (that can contain functions of type $\alpha \rightarrow \alpha$)

• For example, $\text{Fn} \ \alpha \rightarrow \text{ref} \ (\text{fn} \ (x: \alpha) \rightarrow x)$
  - This is a value
  - When it is applied to a type, allocates new ref
Polymorphism and Refs

• Consider the type $(\forall \alpha. (\alpha \rightarrow \alpha)) \text{ Ref}$
  - Values of this type are ref cells containing a polymorphic function
  - Not a prenex type, but ok in NQSML extension

• For example, $\text{ref (Fn } \alpha \Rightarrow \text{ fn (x: } \alpha \text{) } \Rightarrow \text{ x)$}
  - Single reference cell containing polymorphic identity function.
Comparison with SML

- 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)
SML Restrictions Formalized

\[ u ::= \text{Int} \mid \text{Bool} \mid \ldots \quad \text{(Monotypes)} \]
\[ \mid u_1 \rightarrow u_2 \]
\[ \mid \alpha \]
\[ t ::= u \mid \forall \alpha. t \quad \text{(Polytypes)} \]

- The type of terms are polytypes.
- Polymorphic functions can only be applied to monotypes.
Polymorphism and Refs

• Consider the following SML code

  let
  
  val succ = (fn n => n+1)
  val r : ('a->'a) ref = ref (fn x=>x)
  in

  r := succ;

  (!r) (true)

  end

• Accepted by early ML, but gets stuck
Polymorphism and Refs

- If you give SML either of the definitions

  ```ml
  val empty : ('a list) ref = ref []
  val r : ('a->'a) ref = ref (fn x=>x)
  ```

  the compiler will complain. Why?
The SML Value Restriction

• A variable definition may not be polymorphic unless the definition is a syntactic value.
  
  ```ml
  val x : 'a list = []  ok
  val y : int list ref = ref []  ok
  val y : 'a list ref = ref []  not ok
  ```

• Why?
  
  - This would make the type system unsound
Polymorphism and Refs

• Consider the code

\[
\text{let}
\begin{align*}
\text{val } \text{succ } & = (\text{fn } n \Rightarrow n+1) \\
\text{val } r : ('a\Rightarrow'a) \text{ ref } & = (\text{fn } x \Rightarrow x)
\end{align*}
\text{in}
\]
\[
\text{r } := \text{succ;}
\]
\[
(\text{!r}) (\text{true})
\end{align*}
\text{end}
\]
What's Going On?

• Code "looks ok" but would increment true
  - And didn't we say polymorphism + references was sound?
  - Confusing because polymorphic operations are all implicit.

• Let's try filling in the type functions and applications. There are two possibilities:
  - $r$ is a polymorphic function returning a ref cell
  - $r$ is a ref cell containing a polymorphic function
let
  val succ = (fn n => n+1)
  val r = Fn 'a => ref (fn (x:'a)=>x)
in
  r[Int] := succ;
  (! (r[Bool])) (true)
end

• Fresh ref cell for each use of r!
let

   val succ = (fn n => n+1)
   val r = ref (Fn 'a => fn (x:'a)=>x)

in

   r := succ;
   ((!r)[Bool]) (true)

end

• Now the assignment is type incorrect!
Polymorphism and Refs

• Summary of the problem:
  - Naive typechecking assumes Version 1
  - Naive running implements Version 2
  - Could typecheck and run Version 1, but it yields unexpected behavior.
  - Version 2 is outside the SML type system

• Value restriction:
  - Compiler will insert type functions only around values
  - Allow Version 1 only when it acts like Version 2.
Parametricity

• Consider NSQML without effects or recursion.

• Suppose $f$ has type $\forall \alpha . (\alpha \rightarrow \alpha)$
  - What function could it be?
Parametricity

• Suppose \( f \) has type \( \forall \alpha . (\alpha \rightarrow \alpha) \)

  - \( f \) must act like the polymorphic identity

    \[
    \text{Fn } \alpha \Rightarrow (\text{fn } (x: \alpha) \Rightarrow x)
    \]
Parametricity

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

• Suppose $f$ has type $\forall \alpha. (\alpha \text{ list} \to \alpha \text{ list})$
  – $f$ could be an identity function
  – $f$ could always return $\text{nil}[\alpha]$
  – $f$ could return a fixed sublist of its argument
  – $f$ could return a permutation of its argument
Parametricity

• Suppose \( \mathbf{f} \) has type \( \forall \alpha. (\alpha \text{ list} \rightarrow \alpha \text{ list}) \)
  
  - The function cannot depend on the elements of the list, only on its structure.
  
  - The function cannot return elements in the output list that were not in the input list.
  
  - The function cannot work differently depending on the type argument \( \alpha \).
Parametricity

• Informally, a polymorphic function is said to be parametric if its behavior is independent of its type argument.
  – i.e., same algorithm for all type instances.

• This can be elegantly formalized
  – "Related arguments yield related results"
  – But not in this class.
  – Application: TAL and callee-save registers