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

```
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
  - Different code runs depending on types
  - Choice may be compile-time or run-time

NQSML + Polymorphism

v ::= $\underline{n}$ | $\underline{tt}$ | $\underline{ff}$ | ...

  | fn (x:t) => e

  | Fn a => e

e ::= v | e_1 + e_2 | $e_1 < e_2$

  | if e_1 then e_2 else e_3

  | x | let x be e_1 in e_2 | ...

  | e_1(e_2) | e[t]

t ::= Int | Bool | ...

  | t_1 -> t_2

  | a | "a.t

Evaluation Rules

$$\frac{\mathtt{e_1} \to \mathtt{e_1'}}{\mathtt{e_1(e_2)} \to \mathtt{e_1'(e_2)}} \qquad \frac{\mathtt{e_2} \to \mathtt{e_2'}}{\mathtt{v_1(e_2)} \to \mathtt{v_1(e_2')}}$$

$$\frac{}{\mathtt{(fn\ (x:t)\ =>\ e_1)v_2} \to \mathtt{e_1}[\mathtt{x} \to \mathtt{v_2}]}$$

$$\frac{\mathtt{e} \to \mathtt{e'}}{\mathtt{e(t)} \to \mathtt{e'(t)}}$$

$$\frac{}{\mathtt{(Fn}\ a\ \mathtt{=>\ e)(t)} \to \mathtt{e}[a \to t]}$$

Examples

- Polymorphic Identity

  Define **`id`** to be **`Fn`** a **`=> (fn (x:`**a**`) => x)`**

- Then

```
id           : "a.(a ->a)
id[Int]      : Int->Int
id[Int](3)   : Int
```

Examples

- Define **compose** by

```
Fn a => Fn b => Fn g =>
   fn (g : b->g) =>
      fn (f : a->b) =>
         fn (x:a) => g(f(x))
```

- **Then**

```
compose : "a."b."g.(b->g)->(a->b)->(a->g)
compose[Int][Real][String] :
    (Real->String) -> (Int->Real) ->
                           (Int->String)
```

Typing Rules

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

$$\frac{}{\mathrm{T} \vdash \textbf{Int ok}} \qquad \frac{}{\mathrm{T} \vdash \textbf{Bool ok}} \qquad \frac{\mathrm{a}\ \hat{\mathrm{I}}\ \mathrm{T}}{\mathrm{T} \vdash \mathrm{a}\ \textbf{ok}}$$

$$\frac{\mathrm{T} \vdash \mathbf{t_1}\ \textbf{ok} \qquad \mathrm{T} \vdash \mathbf{t_2}\ \textbf{ok}}{\mathrm{T} \vdash \mathbf{t_1\text{->}t_2}\ \textbf{ok}}$$

$$\frac{\mathrm{T\grave{E}\{a\}} \vdash \mathbf{t}\ \textbf{ok} \qquad \mathrm{a}\ \ddot{\mathrm{I}}\ \mathrm{T}}{\mathrm{T} \vdash \text{"}\,\mathrm{a}\mathbf{.t}\ \textbf{ok}}$$

Typing Rules

- Determining whether an expression is well-typed now requires
  - A typing context Γ
  - Set of active type variables Θ
- Rules from before just carry Θ around

$$\frac{}{\Gamma, \Theta \vdash \mathbf{n} : \mathbf{Int}}$$

$$\frac{\Gamma, \Theta \vdash \mathbf{e_1} : \mathbf{t_2\text{->}t} \qquad \Gamma, \Theta \vdash \mathbf{e_2} : \mathbf{t_2}}{\Gamma, \Theta \vdash \mathbf{e_1(e_2)} : \mathbf{t}}$$

New Typing Rules

$$\frac{\mathrm{G, T\grave{E}\{a\}} \vdash \mathbf{e : t} \qquad \mathrm{a\ \ddot{I}\ T}}{\mathrm{G, T} \vdash \mathbf{Fn}\ \mathrm{a}\ \mathbf{=> e :}\ \mathrm{"\,a.}\mathbf{t}}$$

$$\frac{\mathrm{G, T} \vdash \mathbf{e_1 :}\ \mathrm{"\,a.}\mathbf{t_1} \qquad \mathrm{G, T} \vdash \mathbf{t_2\ ok}}{\mathrm{G, T} \vdash \mathbf{e_1(t_2) : t_1}[\mathrm{a} \rightarrow \mathbf{t_2}]}$$

Type Soundness

- Proofs of Type Preservation and Progress go through for this language
  - Even with refs, exceptions, continuations, …

Polymorphism and Refs

- Suppose we have **`ref`** cells
- Consider the type **`"a.((a->a) Ref)`**
  - Values of this type are polymorphic functions
  - Given a, return a ref cell (that can contain functions of type a**->**a)
- For example, **`Fn a => ref (fn (x:a) => x)`**
  - This is a value
  - When it is applied to a type, allocates *new* ref

Polymorphism and Refs

- Consider the type `("a.(a->a)) Ref`
  - Values of this type are ref cells containing a polymorphic function
  - Not a prenex type, but ok in NQSML extension
- For example, `ref (Fn a => fn (x:a) => 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 ::= Int | Bool | ...      (Monotypes)
    | u1 -> u2
    | a
t ::= u | "a.t              (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

```
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.

`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

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

Version 1

```
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!

Version 2

```
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 `"a.(a->a)`
  - What function could it be?

Parametricity

- Suppose **`f`** has type `"a.(a->a)`
  - **`f`** *must* act like the polymorphic identity

    `Fn a => (fn (x:a) => x)`

Parametricity

- Suppose **`f`** has type " a **`.(`**a **`list -> `** a **`list)`**
  - What can **`f`** be?

Parametricity

- Suppose **`f`** has type `"a.(a list -> a list)`
  - **`f`** could be an identity function
  - **`f`** could always return **`nil[a]`**
  - **`f`** could return a fixed sublist of its argument
  - **`f`** could return a permutation of its argument

Parametricity

- Suppose **f** has type " a.(a **list ->** a **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 a.

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