Programs as data
Higher-order functions, polymorphic types, and type inference

Peter Sestoft

Monday 2012-09-24

Plan for today

- Higher-order functions in F#
- A higher-order functional language
- F# mutable references
- Polymorphic types
  - Informal procedure
  - Type rules
  - Unification
  - The union-find data structure
  - Type inference algorithm
- Variant generic types in Java and C#
  - Java use-side variance
  - C# 4.0 declaration-side variance

Higher-order functions and anonymous functions in F#

- A higher-order function takes another function as argument

('a->'b) -> ('a list -> 'b list)

```
let rec map f xs =
    match xs with
      | []     -> []
      | x::xr -> f x :: map f xr
```

```
let mul2 x = 2.0 * x;;
map mul2 [4.0; 5.0; 89.0];;
```

[8.0; 10.0; 178.0]

- Anonymous functions

```
map (fun x -> 2.0 * x) [4.0; 5.0; 89.0]
```

[false; false; true]

```
map (fun x -> x > 10.0) [4.0; 5.0; 89.0]
```

Higher-order functions in C#

- Delegate types

```
delegate R Func<R>()
delegate R Func<A1,R>(A1 x1)
delegate R Func<A1,A2,R>(A1 x1, A2 x2)
```

unit -> R

A1 -> R

A1 * A2 -> R

```
delegate void Action<A1>(A1 x1)
delegate void Action<A1,A2>(A1 x1, A2 x2)
```

A1 -> unit

A1*A2 -> unit

- Anonymous method expressions

```
delegate(int x) { return x>10; }

delegate(int x) { return x*x; }
```

Func<int,bool>

Func<int,int>

```
(int x) => x>10
x => x>10
x => x*x
```

C# 3.0

```
fun (x:int) -> x>10
fun x -> x>10
fun x -> x*x
```

F#

Uniform iteration over a list

```
let rec sum xs =
    match xs with
      | []     -> 0
      | x::xr -> x + sum xr

let rec prod xs =
    match xs with
      | []     -> 1
      | x::xr -> x * prod xr
```

int list -> int

- Generalizing 0/1 to e, and +/* to f:

```
let rec foldr f xs e =
    match xs with
      | []     -> e
      | x::xr -> f x (foldr f xr e)
```

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

List.foldBack in F#

The foldr function replaces :: by f, and [] by e:

$$\text{foldr} \diamond (x_1::x_2::\ldots::x_n::[])\ e = x_1 \diamond (x_2 \diamond (\ldots \diamond (x_n \diamond e) \ldots))$$

Many functions definable using foldr

```
len xs    = foldr (fun _ res -> 1+res) xs 0

sum xs    = foldr (fun x res -> x+res) xs 0

prod xs   = foldr (fun x res -> x*res) xs 1

map g xs = foldr (fun x res -> g x :: res) xs []

listconcat xss = foldr (fun xs res -> xs @ res) xss []

strconcat ss = foldr (fun s res -> s ^ res) ss ""
```

`filter p xs =` list of those x in xs for which p x is true

`forall p xs =` p x is true for all x in xs

`exists p xs =` p x is true for some x in xs

Joint exercises

- Define these F# functions in terms of foldr
  - filter p xs
  - forall p xs
  - exists p xs

Composing functions, “pipe”

- Given list xs, throw away small numbers, square the remaining numbers, and compute their sum:

```
sum (map (fun x -> x*x) (filter (fun x -> x>10) xs))
```

- Somewhat difficult to read: inside-out
- Idea: Define infix higher-order function |>

```
x |> f = f x
```

- Now the list operations combine naturally:

```
xs |> filter (fun x -> x>10) |> map (fun x -> x*x) |> sum
```

F# mutable references

- A reference is a cell that can be updated

```
let r = ref 177
!r
(r := !r+1; !r)
!r
```

Create int reference

Dereference

Assign to reference

- Useful for generation of new names etc:

```
let nextlab = ref -1;;
let newLabel () = (nextlab := 1 + !nextlab;
                   "L" + string (!nextlab));;
newLabel();;
newLabel();;
newLabel();;
```

Higher-order micro-ML/micro-F#

- Higher-order functional language
  - A function may be given as argument:

```
let twice g x = g(g x)
```

  - A function may be returned as result

```
let add x = let f y = x+y in f
let addtwo = add 2
let x = 77
addtwo 5
```

add has two arguments!

- Closures needed:
  - The function returned must enclose the value of f's parameter x – has nothing to do with later x
- Same micro-ML syntax: Fun/Absyn.fs

Interpretation of a higher-order language

- The closure machinery is already in place
- Just redefine function application:

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    | ...
    | Call(eFun, eArg) ->
      let fClosure = eval eFun env
      in match fClosure with
         | Closure (f, x, fBody, fDeclEnv) ->
           let xVal = eval eArg env
           let fBodyEnv =
               (x, xVal) :: (f, fClosure) :: fDeclEnv
           in eval fBody fBodyEnv
         | _ -> failwith "eval Call: not a function"
```

ML/F#-style parametric polymorphism

```
let f x = 1
in f 2 + f true
```

Type for f is
'a -> int

int -> int

bool -> int

- Each expression has a statically known type
- The type may be polymorphic ('many forms') and have multiple type instances

Type generalization and specialization

- If f has type ($\alpha \rightarrow$ int) and α appears nowhere else, the type gets generalized to a *type scheme* $\forall\alpha.(\alpha \rightarrow \text{int})$:

```
let f x = 1
```

$\forall\alpha.(\alpha \rightarrow \text{int})$

- If f has type scheme $\forall\alpha.(\alpha \rightarrow \text{int})$ then α may be instantiated by/specialized to any type:

```
f 42
f false
f [22]
f (3,4)
```

f : int → int

f : bool → int

f : int list → int

f : int*int → int

Polymorphic type inference

- F# and ML have polymorphic type *inference*
- Static types, but not explicit types on functions

α β

```
let twice g y = g (g y)
```

$\alpha = \beta \to \delta$

$\alpha = \delta \to \varepsilon$

$\beta = \delta$ and $\delta = \varepsilon$
so
$\alpha = \beta \to \beta$

$(\beta \to \beta) \to (\beta \to \beta)$

- We *generalize* β, so **`twice`** gets the polymorphic type $\forall\beta.\ (\beta \to \beta) \to (\beta \to \beta)$, hence "$\beta$ may be any type"

```
let mul2 y = 2 * y
```

mul: int -> int

```
twice mul2 11
```

twice : (int->int)->(int->int)

Basic elements of type inference

- "Guess" types using type variables α, β, …
- Build and solve "type equations" $\alpha = \beta \rightarrow \delta$ …
- *Generalize* types of let-bound variables/funs. to obtain type schemes $\forall\beta.\ (\beta\rightarrow\beta) \rightarrow (\beta\rightarrow\beta)$
- *Specialize* type schemes at variable use

- This is called
  - ML-polymorphism
  - let-polymorphism
  - Hindley-Milner polymorphism (1969 & 1978)

Restrictions on ML polymorphism, 1

- Only let-bound variables and functions can have a polymorphic type
- A *parameter*'s type is never polymorphic:

```
let f g = g 7 + g false
```

Ill-typed: parameter g never polymorphic

- A function is not polymorphic in its own body:

```
let rec h x =
    if true then 22
    else h 7 + h false
```

Ill-typed: h not polymorphic in its own body

Restrictions on ML polymorphism, 2

- Types must be finite and non-circular

f not polymorphic in its own body

```
let rec f x = f f
```

- Guess x has type α
- Then `f` must have type $\alpha \to \beta$ for some β
- But because we apply `f` to itself in `(f f)`, we must have $\alpha = \alpha \to \beta$
- But then $\alpha = (\alpha \to \beta) \to \beta = ((\alpha \to \beta) \to \beta) \to \beta = \ldots$ is not a finite type
- So the example is ill-typed

Restrictions on ML polymorphism, 3

- A type parameter that is used in an enclosing scope cannot be generalized

g : β→int

α = β

α

β

```
let f x =
    let g y = if x=y then 11 else 22
    in g false
in f 42
```

α bound in outer scope, cannot generalize β

Ill-typed: function g not polymorphic

- Reason: If this were well-typed, we would compare x (42) with y (false), not good…

Joint exercises

- Which of these are well-typed, and why/not?

```
let f x = 1
in f f
```

```
let f g = g g
```

```
let f x =
    let g y = y
    in g false
in f 42
```

```
let f x =
    let g y = if true then y else x
    in g false
in f 42
```

Type rules for ML-polymorphism

$$\frac{}{\rho \vdash i : \mathtt{int}}$$

$$\frac{}{\rho \vdash b : \mathtt{bool}}$$

$$\frac{\rho(x) = \forall \alpha_1, \ldots, \alpha_n . t}{\rho \vdash x : [t_1/\alpha_1, \ldots, t_n/\alpha_n]t}$$

Specialize from typescheme

$$\frac{\rho \vdash e_1 : \mathtt{int} \qquad \rho \vdash e_2 : \mathtt{int}}{\rho \vdash e_1 + e_2 : \mathtt{int}}$$

$$\frac{\rho \vdash e_1 : \mathtt{int} \qquad \rho \vdash e_2 : \mathtt{int}}{\rho \vdash e_1 < e_2 : \mathtt{bool}}$$

Generalize to typescheme

$$\frac{\rho \vdash e_r : t_r \qquad \rho[x \mapsto \forall \alpha_1, \ldots, \alpha_n . t_r] \vdash e_b : t \qquad \alpha_1, \ldots, \alpha_n \text{ not free i}}{\rho \vdash \mathtt{let}\ x = e_r\ \mathtt{in}\ e_b\ \mathtt{end}\ : t}$$

$$\frac{\rho \vdash e_1 : \mathtt{bool} \qquad \rho \vdash e_2 : t \qquad \rho \vdash e_3 : t}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : t}$$

$$\frac{\rho[x \mapsto t_x, f \mapsto t_x \to t_r] \vdash e_r : t_r \qquad \rho[f \mapsto \forall \alpha_1, \ldots, \alpha_n . t_x \to t_r] \vdash e_b : t \qquad \alpha_1, \ldots, \alpha_n \text{ not free in } \rho}{\rho \vdash \mathtt{let}\ f\ x = e_r\ \mathtt{in}\ e_b\ \mathtt{end}\ : t}$$

$$\frac{\rho \vdash e_1 : t_x \to t_r \qquad \rho \vdash e_2 : t_x}{\rho \vdash e_1\ e_2 : t_r}$$

Joint exercises

- Draw the type trees for some of these

```
let x = 1
in x < 2
```

```
let f x = 1
in f 2 + f false
```

```
let f x = 1
in f f
```

Programming type inference

- Algorithm W (Damas & Milner 1982) with many later improvements
- Symbolic type equation solving by
  - Unification
  - The union-find data structure
- “Not free in ρ” formalized by binding levels:

$\alpha = \beta$

α:0

β:1 β:0

```
0 let f x =
  1 let g y = if x=y then 11 else 22
    in g false
  in f 42
```

- Since β-level < g-level, do not generalize β

Unification of two types, unify(t_1,t_2)

Type t_1	**Type t_2**	**Action**
int	int	No action
bool	bool	No action
$t_{1x} \rightarrow t_{1r}$	$t_{2x} \rightarrow t_{2r}$	unify(t_{1x},t_{2x}) and unify(t_{1r},t_{2r})
α	α	No action
α	β	Make $\alpha=\beta$
α	t_2	Make $\alpha=t_2$ unless t_2 contains α
t_1	β	Make $\beta=t_1$ unless t_1 contains β
All other cases		Failure, type error!

The union-find data structure

- A graph of nodes (type variables) divided into disjoint classes
- Each class has a representative node
- Operations:
  - New: create new node (type variable)
  - Find(n): find representative of node n's class
  - Union(n1,n2): join the classes of n1 and n2

Type inference for micro-ML, 1

```
let rec typ (lvl : int) (env : tenv) (e : expr) : typ =
    match e with
    | CstI i -> TypI
    | CstB b -> TypB
    | Var x  -> specialize lvl (lookup env x)
    | ...
```

$$\frac{}{\rho \vdash i : \texttt{int}}$$

$$\frac{}{\rho \vdash b : \texttt{bool}}$$

$$\frac{\rho(x) = \forall \alpha_1, \ldots, \alpha_n . t}{\rho \vdash x : [t_1/\alpha_1, \ldots, t_n/\alpha_n] t}$$

typ ρ e = t

if and only if

$$\rho \vdash e : t$$

Type inference for micro-ML, 2

```
let rec typ (lvl : int) (env : tenv) (e : expr) : typ =
    match e with
    | Prim(ope, e1, e2) ->
      let t1 = typ lvl env e1
      let t2 = typ lvl env e2
      match ope with
        | "*" -> (unify TypI t1; unify TypI t2; TypI)
        | "+" -> (unify TypI t1; unify TypI t2; TypI)
        | "=" -> (unify t1 t2; TypB)
        | "<" -> (unify TypI t1; unify TypI t2; TypB)
        | "&" -> (unify TypB t1; unify TypB t2; TypB)
        | _   -> failwith ("unknown primitive " ^ ope)
```

$$\frac{\rho \vdash e_1 : \texttt{int} \qquad \rho \vdash e_2 : \texttt{int}}{\rho \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\rho \vdash e_1 : \texttt{int} \qquad \rho \vdash e_2 : \texttt{int}}{\rho \vdash e_1 < e_2 : \texttt{bool}}$$

Type inference for micro-ML, 3

```
let rec typ (lvl : int) (env : tenv) (e : expr) : typ =
    match e with
    | If(e1, e2, e3) ->
      let t2 = typ lvl env e2
      let t3 = typ lvl env e3
      unify TypB (typ lvl env e1);
      unify t2 t3;
      t2
```

$$\frac{\rho \vdash e_1 : \texttt{bool} \qquad \rho \vdash e_2 : t \qquad \rho \vdash e_3 : t}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t}$$

Type inference for micro-ML, 4

```
let rec typ (lvl : int) (env : tenv) (e : expr) : typ =
    match e with
    | ...
    | Let(x, eRhs, letBody) ->
      let lvl1 = lvl + 1
      let resTy = typ lvl1 env eRhs
      let letEnv = (x, generalize lvl resTy) :: env
      typ lvl letEnv letBody
    | ...
```

$$\frac{\rho \vdash e_r : t_r \qquad \rho[x \mapsto \forall \alpha_1, \ldots, \alpha_n . t_r] \vdash e_b : t \qquad \alpha_1, \ldots, \alpha_n \text{ not free in } \rho}{\rho \vdash \texttt{let}\ x = e_r\ \texttt{in}\ e_b\ \texttt{end}\ : t}$$

Properties of ML-style polymorphism

- The type found by the inference algorithm is the most general one: the *principal type*
- Consequence: Type checking can be modular
- Types can be large and type inference slow:

```
let id x = x
let pair x y p = p x y
let p1 p = pair id id p
let p2 p = pair p1 p1 p
let p3 p = pair p2 p2 p
let p4 p = pair p3 p3 p;;
let p5 p = pair p4 p4 p;;
```

Exponentially many type variables!

- In practice types are small and inference fast

Type inference in C# 3.0

```
var x = “hello”;        // Inferred type: String
… x.Length …
x = 17;                 // Type error
```

- No polymorphic generalization
- Can infer parameter type of anonymous function from context: **`xs.Where(x=>x*x>5)`**
- Cannot infer type of anonymous function
- Parameter types in methods
  - must be declared
  - cannot be inferred, because C# allows method overloading …

Polymorphism (generics) in Java and C#

- Polymorphic types

```
interface IEnumerable<T> { ... }
class List<T> : IEnumerable<T> { ... }
struct Pair<T,U> { T fst; U snd; ... }
delegate R Func<A,R>(A x);
```

- Polymorphic methods

```
void Process<T>(Action<T> act, T[] xs)
```

C#

```
void <T> Process(Action<T> act, T[] arr)
```

Java

- Type parameter constraints

```
void Sort<T>(T[] arr) where T : IComparable<T>
```

C#

```
void <T extends Comparable<T>> Sort(T[] arr)
```

Java

Variance in type parameters

- Assume Student subtype of Person

```
void PrintPeople(IEnumerable<Person> ps) { ... }
```

```
IEnumerable<Student> students = ...;
PrintPeople(students);
```

Java and C# 3 say NO: Ill-typed!

- C# 3 and Java:
  - A generic type is *invariant* in its parameter
  - I<Student> is *not* subtype of I<Person>
- Co-variance (co=with):
  - I<Student> is subtype of I<Person>
- Contra-variance (contra=against):
  - I<Person> is subtype of I<Student>

Co-/contra-variance is unsafe in general

- Co-variance is unsafe in general

```
List<Student> ss = new List<Student>();
List<Person> ps = ss;
ps.Add(new Person(...));
Student s0 = ss[0];
```

Wrong!

Because would allow writing Person to Student list

- Contra-variance is unsafe in general

```
List<Person> ps = ...;
List<Student> ss = ps;
Student s0 = ss[0];
```

Wrong!

Because would allow reading Student from Person list

- But:
  - co-variance OK if we *only read (output)* from list
  - contra-variance OK if we *only write (input)* to list

Java 5 wildcards

- Use-side co-variance

```
void PrintPeople(ArrayList<? extends Person> ps) {
  for (Person p : ps) { … }
}
...
PrintPeople(new ArrayList<Student>());
```

OK!

- Use-side contra-variance

```
void AddStudentToList(ArrayList<? super Student> ss) {
  ss.add(new Student());
}
...
AddStudentToList(new ArrayList<Person>());
```

OK!

Co-variance in interfaces (C# 4)

- When an I<T> only produces/*outputs* T's, it is safe to use an I<Student> where a I<Person> is expected
- This is co-variance
- Co-variance is declared with the **`out`** modifier

```
interface IEnumerable<out T> {
  IEnumerator<T> GetEnumerator();
}
interface IEnumerator<out T> {
  T Current { get; }
}
```

- Type T can be used only in *output* position; e.g. not as method argument (input)

Contra-variance in interfaces (C# 4)

- When an I<T> only consumes/*inputs* T's, it is safe to use an I<Person> where an I<Student> is expected
- This is contra-variance
- Contra-variance is declared with **`in`** modifier

```
interface IComparer<in T> {
  int Compare(T x, T y);
}
```

- Type T can be used only in *input* position; e.g. not as method return type (output)

Variance in function types (C# 4)

- A C# delegate type is
  - co-variant in return type (output)
  - contra-variant in parameters types (input)
- Return type co-variance:

```
Func<int,Student> nthStudent = ...
Func<int,Person> nthPerson = nthStudent;
```

- Argument type contra-variance:

```
Func<Person,int> personAge = ...
Func<Student,int> studentAge = personAge;
```

- F# does not support co-variance or contra-variance (yet?)

Reading and homework

- This week's lecture:
  - PLC sections A.11-A.12 and 5.1-5.5 and 6.1-6.7
  - Exercises 6.1, 6.2, 6.3, 6.4, 6.5

- Next week's lecture:
  - PLCSD chapter 7
  - Strachey: Fundamental Concepts in …
  - Kernighan & Richie: The C programming language, chapter 5.1-5.5