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**
  
  ```fsharp
  let rec map f xs =
  match xs with
  | []    -> []
  | x::xr -> f x :: map f xr
  ```
  
  ```fsharp
  let mul2 x = 2.0 * x;;
  map mul2 [4.0; 5.0; 89.0];;
  ```
  
  ```fsharp
  let mul2 x = 2.0 * x;;
  map mul2 [4.0; 5.0; 89.0];;
  ```

- **Anonymous functions**

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

  ```fsharp
  map (fun x -> x > 10.0) [4.0; 5.0; 89.0]
  ```
Higher-order functions in C#

• Delegate types

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

• Anonymous method expressions

```csharp
delegate (int x) { return x>10; }  // Func<int,bool>
delegate (int x) { return x*x; }   // Func<int,int>
(int x) => x>10
x => x>10
x => x*x
```

C# 3.0

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

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

• 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)

The foldr function replaces :: by f, and [] by e:
foldr ◊ (x₁::x₂::...::xₙ::[]) e = x₁ ◊ (x₂ ◊ (... ◊ (xₙ ◊ e) ...))
Many functions definable using foldr

\begin{align*}
\text{len } \textit{xs} & = \text{foldr} \ (\text{fun } \_ \ \textit{res} \rightarrow 1+\textit{res}) \ \textit{xs} \ 0 \\
\text{sum } \textit{xs} & = \text{foldr} \ (\text{fun } \textit{x} \ \textit{res} \rightarrow \textit{x}+\textit{res}) \ \textit{xs} \ 0 \\
\text{prod } \textit{xs} & = \text{foldr} \ (\text{fun } \textit{x} \ \textit{res} \rightarrow \textit{x}*\textit{res}) \ \textit{xs} \ 1 \\
\text{map } \textit{g} \ \textit{xs} & = \text{foldr} \ (\text{fun } \textit{x} \ \textit{res} \rightarrow \textit{g} \ \textit{x} :: \textit{res}) \ \textit{xs} \ [] \\
\text{listconcat } \textit{xss} & = \text{foldr} \ (\text{fun } \textit{xs} \ \textit{res} \rightarrow \textit{xs} @ \textit{res}) \ \textit{xss} \ [] \\
\text{strconcat } \textit{ss} & = \text{foldr} \ (\text{fun } \textit{s} \ \textit{res} \rightarrow \textit{s} ^ {\text{"}} \ \textit{res}) \ \textit{ss} \ ""
\end{align*}

\begin{align*}
\text{filter } \textit{p} \ \textit{xs} & = \text{list of those } \textit{x} \text{ in } \textit{xs} \text{ for which } \textit{p} \ \textit{x} \text{ is true} \\
\text{forall } \textit{p} \ \textit{xs} & = \text{p} \ \textit{x} \text{ is true for all } \textit{x} \text{ in } \textit{xs} \\
\text{exists } \textit{p} \ \textit{xs} & = \text{p} \ \textit{x} \text{ is true for some } \textit{x} \text{ in } \textit{xs}
\end{align*}
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:

$$\text{sum} \ (\text{map} \ (\text{fun} \ x \rightarrow x \cdot x) \ (\text{filter} \ (\text{fun} \ x \rightarrow x > 10) \ xs))$$

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

$$x \ |> f = f \ x$$

• Now the list operations combine naturally:

$$xs \ |> \text{filter} \ (\text{fun} \ x \rightarrow x > 10) \ |> \text{map} \ (\text{fun} \ x \rightarrow x \cdot x) \ |> \text{sum}$$
F# mutable references

• A reference is a cell that can be updated

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

Create int reference
Dereference
Assign to reference

• Useful for generation of new names etc:

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

- 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:

```ml
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

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

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

Type for f is `'a -> int`
Type generalization and specialization

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

\[
\text{let } f \; x = 1 \quad \forall \alpha.(\alpha \rightarrow \text{int})
\]

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

\[
\begin{align*}
\text{f 42} & \quad \text{f : int \rightarrow int} \\
\text{f false} & \quad \text{f : bool \rightarrow int} \\
\text{f [22]} & \quad \text{f : int list \rightarrow int} \\
\text{f (3,4)} & \quad \text{f : int*int \rightarrow int}
\end{align*}
\]
Polymorphic type inference

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

\[ \alpha = \beta \rightarrow \delta \]
\[ \beta = \delta \rightarrow \varepsilon \]
\[ \alpha = \delta \rightarrow \varepsilon \]

We generalize \( \beta \), so twice gets the polymorphic type \( \forall \beta. (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \), hence “\( \beta \) may be any type”

let mul2 y = 2 * y  
let twice g y = g (g y)  
let mul2 y = 2 * y  
twice mul2 11

\[ \text{mul: int -> int} \]
\[ \text{twice : (int->int)->(int->int)} \]
Basic elements of type inference

• “Guess” types using type variables $\alpha$, $\beta$, ...
• 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
Restrictions on ML polymorphism, 1

• Only let-bound variables and functions can have a polymorphic type.
• A parameter’s type is never polymorphic:

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

Ill-typed: parameter g never polymorphic

• A function is not polymorphic in its own body:

```ocaml
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

\[
\text{let rec } f \ x = f \ f
\]

\(f\) not polymorphic in its own body

• Guess \(x\) has type \(\alpha\)
• Then \(f\) must have type \(\alpha \rightarrow \beta\) for some \(\beta\)
• But because we apply \(f\) to itself in \((f \ f)\), we must have \(\alpha = \alpha \rightarrow \beta\)
• But then \(\alpha = (\alpha \rightarrow \beta) \rightarrow \beta = ((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \beta = \ldots\)
  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

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

$\alpha = \beta$

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

$g : \beta \rightarrow \text{int}$

$\alpha$ bound in outer scope, cannot generalize $\beta$

Ill-typed: function $g$ not polymorphic
Joint exercises

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

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

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

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

```ml
let f x =
  let g y = if true then y else x
  in g false
in f 42
```
Type rules for ML-polymorphism

\[
\frac{
\rho \vdash i : \text{int}
}{
\rho \vdash b : \text{bool}
}
\]

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

\[
\frac{
\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}
}{
\rho \vdash e_1 + e_2 : \text{int}
}\]

\[
\frac{
\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}
}{
\rho \vdash e_1 < e_2 : \text{bool}
}\]

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

\[
\frac{
\rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t
}{
\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
}\]

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

\[
\frac{
\rho \vdash e_1 : t_x \rightarrow t_r \quad \rho \vdash e_2 : t_x
}{
\rho \vdash e_1 \ e_2 : t_r
}\]
Joint exercises

• Draw the type trees for some of these

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

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

```ml
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 \( \rho \)” formalized by binding levels:

\[
\begin{align*}
0 & \quad \alpha : 0 \\
1 & \quad \beta : 0 \\
& \quad \text{let } g \ y = \text{if } x = y \text{ then } 11 \text{ else } 22 \\
& \quad \text{in g false} \\
& \quad \text{in f 42}
\end{align*}
\]

- Since \( \beta \)-level < g-level, do not generalize \( \beta \)
# Unification of two types, unify($t_1,t_2$)

<table>
<thead>
<tr>
<th>Type $t_1$</th>
<th>Type $t_2$</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>No action</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>No action</td>
</tr>
<tr>
<td>$t_{1x} \rightarrow t_{1r}$</td>
<td>$t_{2x} \rightarrow t_{2r}$</td>
<td>unify($t_{1x},t_{2x}$) and unify($t_{1r},t_{2r}$)</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>No action</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\beta$</td>
<td>Make $\alpha=\beta$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$t_2$</td>
<td>Make $\alpha=t_2$ unless $t_2$ contains $\alpha$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$\beta$</td>
<td>Make $\beta=t_1$ unless $t_1$ contains $\beta$</td>
</tr>
<tr>
<td>All other cases</td>
<td>Failure, type error!</td>
<td></td>
</tr>
</tbody>
</table>
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)
  | ...
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)
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
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
  | ...

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

```ocaml
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

```csharp
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

```java
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

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

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

- Type parameter constraints

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

```csharp
void <T extends Comparable<T>> Sort(T[] arr)
```
Variance in type parameters

• Assume Student subtype of Person

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

```csharp
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

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

- Contra-variance is unsafe in general

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

- 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

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

• Use-side contra-variance

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

```csharp
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

```csharp
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