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

Peter Sestoft Monday 2012-09-24

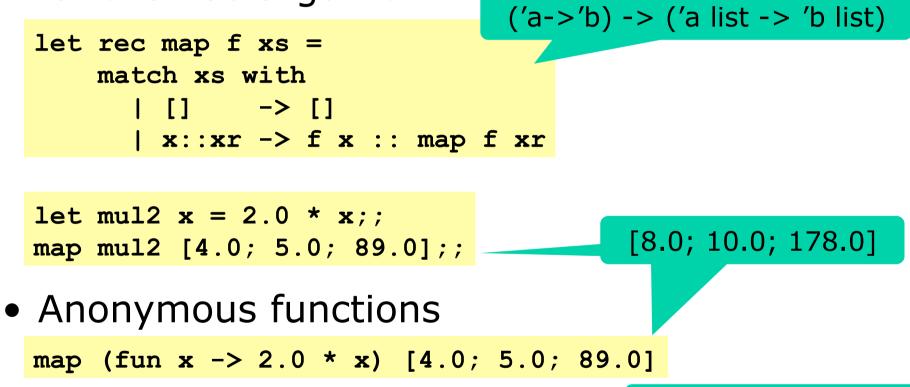


Plan for today

- 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#

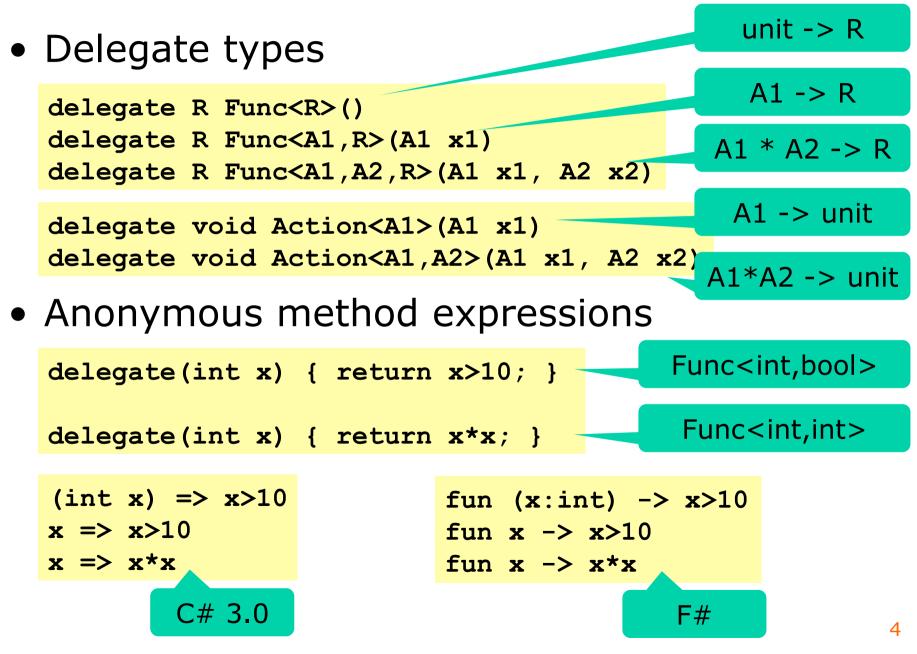
 A higher-order function takes another function as argument



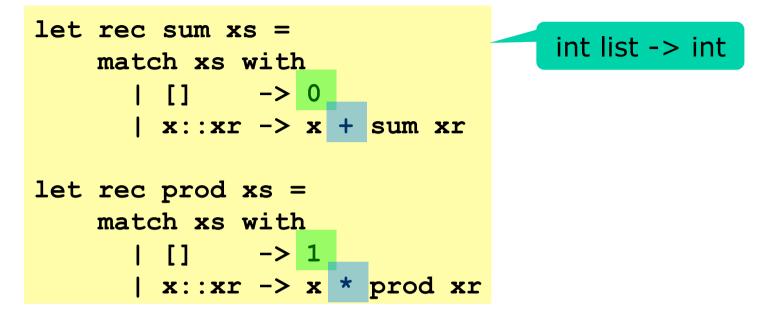
map (fun $x \rightarrow x > 10.0$) [4.0; 5.0; 89.0]

[false; false; true]

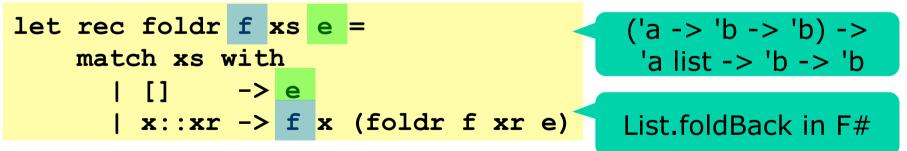
Higher-order functions in C#



Uniform iteration over a list



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



The foldr function replaces :: by f, and [] by e: foldr \diamond (x₁::x₂::..::x_n::[]) e = x₁ \diamond (x₂ \diamond (... \diamond (x_n \diamond e) ...))

Many functions definable using foldr

<pre>len xs = foldr (fun _ res -> 1+res) xs 0</pre>			
sum xs = foldr (fun x res -> x+res) xs 0			
prod xs = foldr (fun x res -> x*res) xs 1			
<pre>map g xs = foldr (fun x res -> g x :: res) xs []</pre>			
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 $x \rightarrow x^*x$) (filter (fun $x \rightarrow x>10$) xs))

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

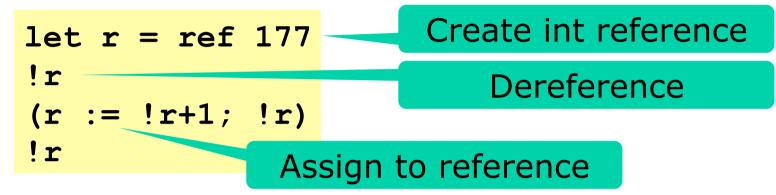
 $x \mid > f = f x$

• Now the list operations combine naturally:



F# mutable references

• A reference is a cell that can be updated



• Useful for generation of new names etc:

IT University of Copenhagen

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

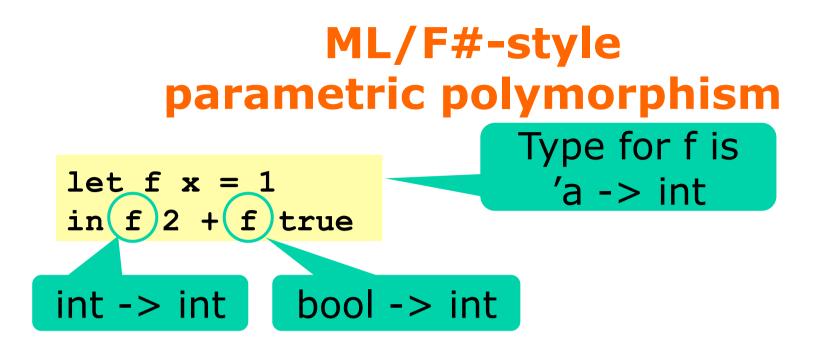
add has two arguments!

- 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:





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

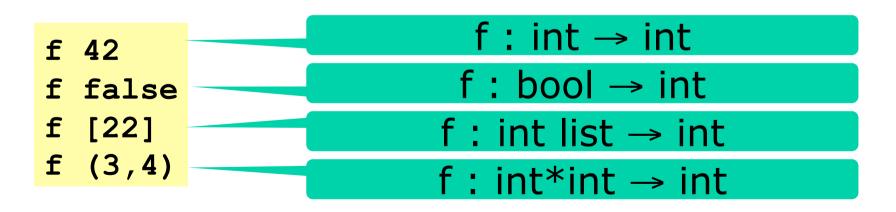


Type generalization and specialization

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

let f x = 1
$$\forall \alpha.(\alpha \rightarrow int)$$

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





Polymorphic type inference

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

$$\alpha = \beta \rightarrow \delta$$

$$\beta = \delta \text{ and } \delta = \varepsilon$$

$$\beta = \delta \text{ and } \delta = \varepsilon$$

$$\beta = \delta \text{ and } \delta = \varepsilon$$

$$\beta = \delta \text{ and } \delta = \varepsilon$$

$$\alpha = \beta \rightarrow \beta$$

$$\alpha = \delta \rightarrow \varepsilon$$

• We generalize β , so twice gets the polymorphic type $\forall \beta$. ($\beta \rightarrow \beta$) \rightarrow ($\beta \rightarrow \beta$), hence " β may be any type"

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

twice mul2 11

Basic elements of type inference

- "Guess" types using type variables α , β , ...
- Build and solve "type equations" $\alpha = \beta \rightarrow \delta \dots$
- *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 <u>polymorphic</u>:

Ill-typed: parameter g never polymorphic

• A function is not polymorphic in its own body:





let f g = g 7 + g false

Restrictions on ML polymorphism, 2

Types must be finite and non-circular

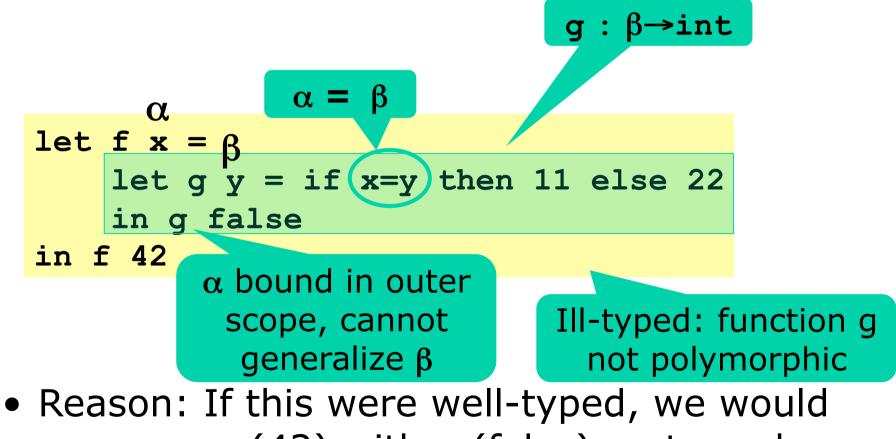
f not polymorphic in its own body

let rec f x = f f

- Guess x has type $\boldsymbol{\alpha}$
- Then **f** must have type $\alpha \rightarrow \beta$ for some β
- But because we apply **f** to itself in (**f f**), we must have $\alpha = \alpha \rightarrow \beta$
- But then $\alpha = (\alpha \rightarrow \beta) \rightarrow \beta = ((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \beta = ...$ 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



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
IT University of Copenhagen
                              www.itu.dk
```

Type rules for ML-polymorphism

Joint exercises

• Draw the type trees for some of these

$$let x = 1$$
$$in x < 2$$

let f x = 1in f 2 + f false

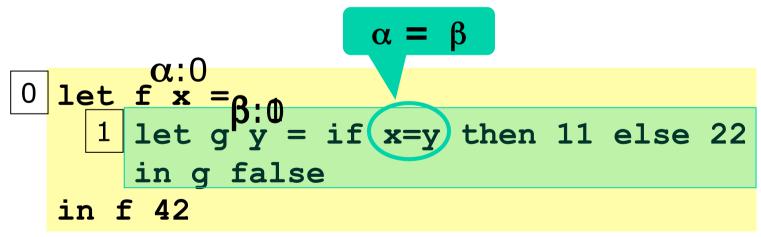
let
$$f x = 1$$

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 $\rho^{\prime\prime}$ formalized by binding levels:



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

IT University of Copenhagen

www.itu.dk

Unification of two types, unify(t₁,t₂)

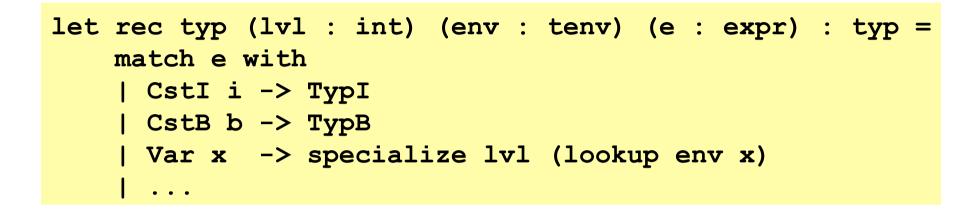
Type t ₁	Type t ₂	Action
int	int	No action
bool	bool	No action
$t_{1x} \rightarrow t_{1r}$	$t_{2x} \rightarrow t_{2r}$	unify(t_{1x} , t_{2x}) and unify(t_{1r} , t_{2r})
α	α	No action
α	β	Make $\alpha = \beta$
α	t ₂	Make $\alpha = t_2$ unless t_2 contains α
t ₁	β	Make $\beta = t_1$ unless t_1 contains 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







 $\rho \vdash x : [t_1/\alpha_1, \ldots, t_n/\alpha_n]t$



 $\rho \vdash e_1$: int $\rho \vdash e_2$: int

 $\rho \vdash e_1 + e_2$: int

$$\rho \vdash e_1 : \texttt{int} \quad \rho \vdash e_2 : \texttt{int}$$

 $\rho \vdash e_1 < e_2$: bool

```
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
```

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

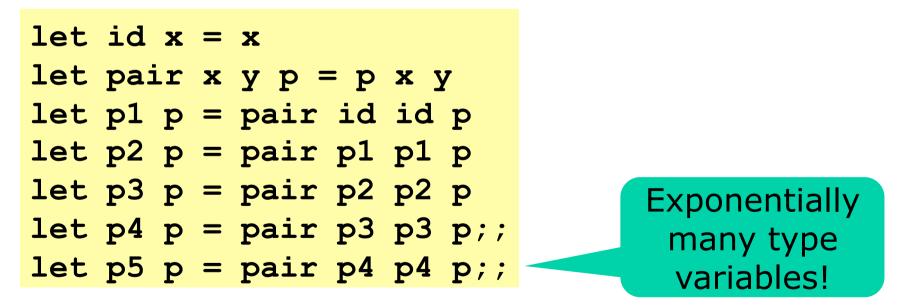


$$\begin{array}{ll} \rho \vdash e_r : t_r & \rho[x \mapsto \forall \alpha_1, \dots, \alpha_n.t_r] \vdash e_b : t & \alpha_1, \dots, \alpha_n \text{ not free in } \rho \\ & \rho \vdash \texttt{let } x = e_r \text{ in } e_b \texttt{ end } : t \end{array}$$



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:



• In practice types are small and inference fast

IT University of Copenhagen

Type inference in C# 3.0

<pre>var x = "hello";</pre>	<pre>// Inferred type: String</pre>
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 ...

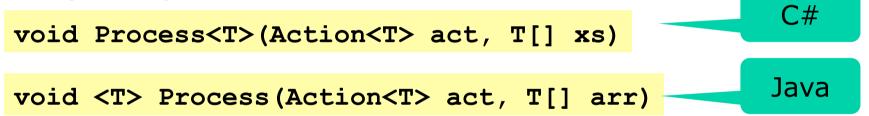


Polymorphism (generics) in Java and C#

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



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) { ... }



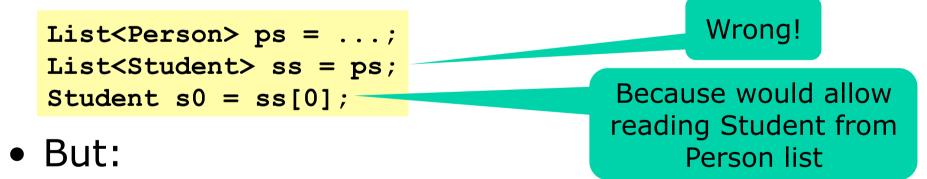
- C# 3 and Java:
 - A generic type is *invariant* in its parameter
 - I<Student> is not subtype of I<Person>
- Co-variance (co=with):
 - I<Student> is subtype of I<Person>
- Contra-variance (contra=against):
 - I<Person> is subtype of I<Student>

Co-/contra-variance is unsafe in general

• Co-variance is unsafe in general



Contra-variance is unsafe in general



- 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>());
```

• Use-side contra-variance

```
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

```
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)

IT University of Copenhagen

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 contravariance (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

