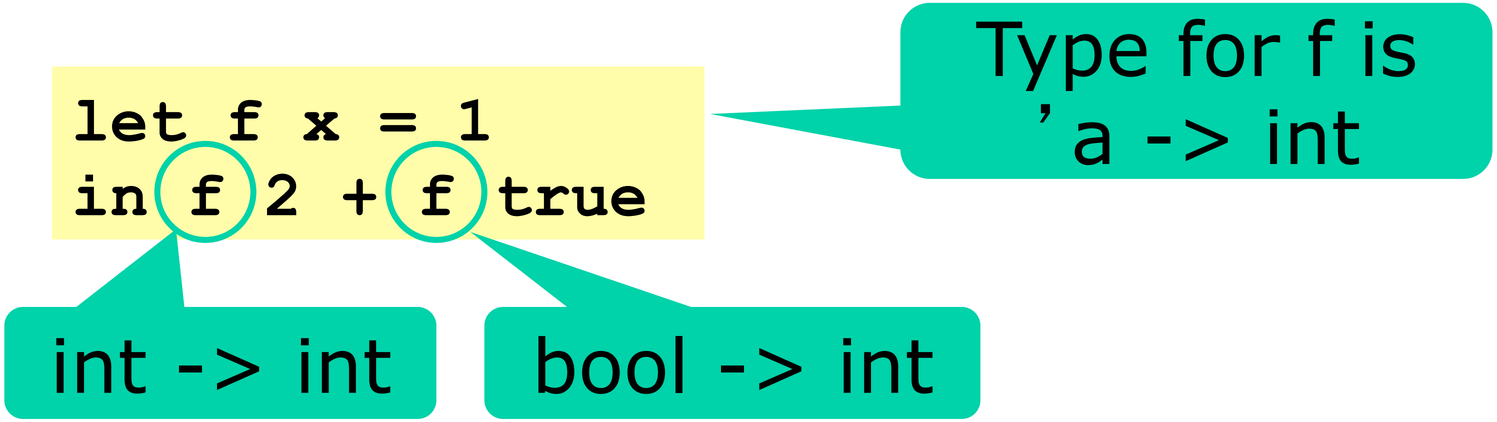
Programs as data
Higher-order functions, polymorphic types, and type inference

Peter Sestoft

Monday 2013-09-16

Note by Baris Aktemur:
These slides have been adapted from the originals available at http://www.itu.dk/courses/BPRD/E2013/.
I thank Peter Sestoft for making the PPT's available.

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 compile-time type
- The type may be *polymorphic* (‘many forms’) and have multiple *type instances*

Type generalization and specialization

- If f has type (α → int) and α appears nowhere else, the type gets generalized to a *type scheme* written ∀α.(α → int):

```
let f x = 1
```

∀α.(α → int)

- If f has type scheme ∀α.(α → int) then α may be instantiated by/specialized to any type:

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

f : int → int

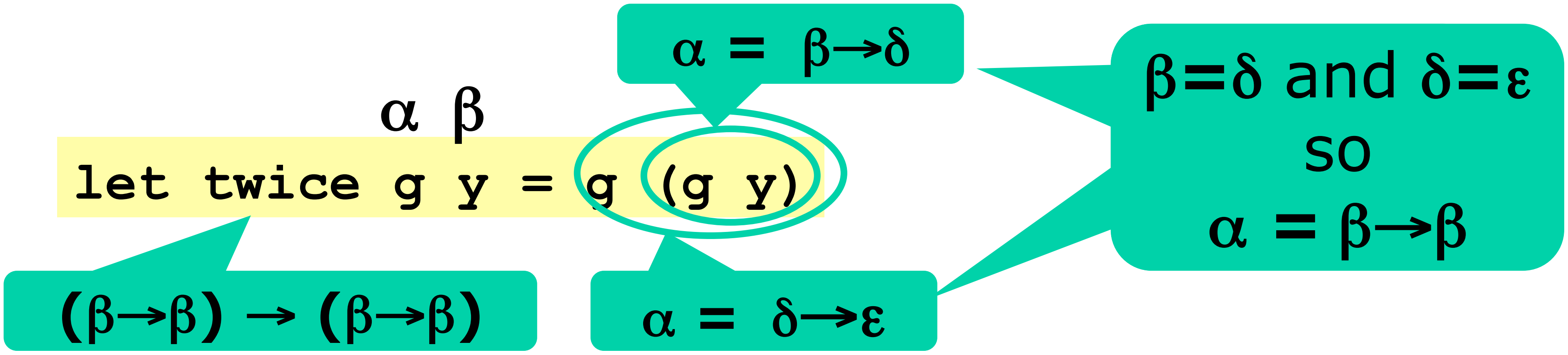
f : bool → int

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



$\alpha \ \beta$

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

$\alpha = \beta \rightarrow \delta$

$\alpha = \delta \rightarrow \varepsilon$

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

$(\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)$

- We *generalize* β, so **`twice`** gets the type scheme $\forall \beta.\ (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)$, hence “β 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" $\alpha = \beta\to\delta$ …
- *Generalize* types of let-bound variables/funs. to obtain type schemes $\forall\beta.\ (\beta\to\beta) \to (\beta\to\beta)$
- *Specialize* type schemes at variable use

- This type system has several names:
  - ML-polymorphism
  - let-polymorphism
  - Hindley-Milner polymorphism (Hindley 1969 & Milner 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

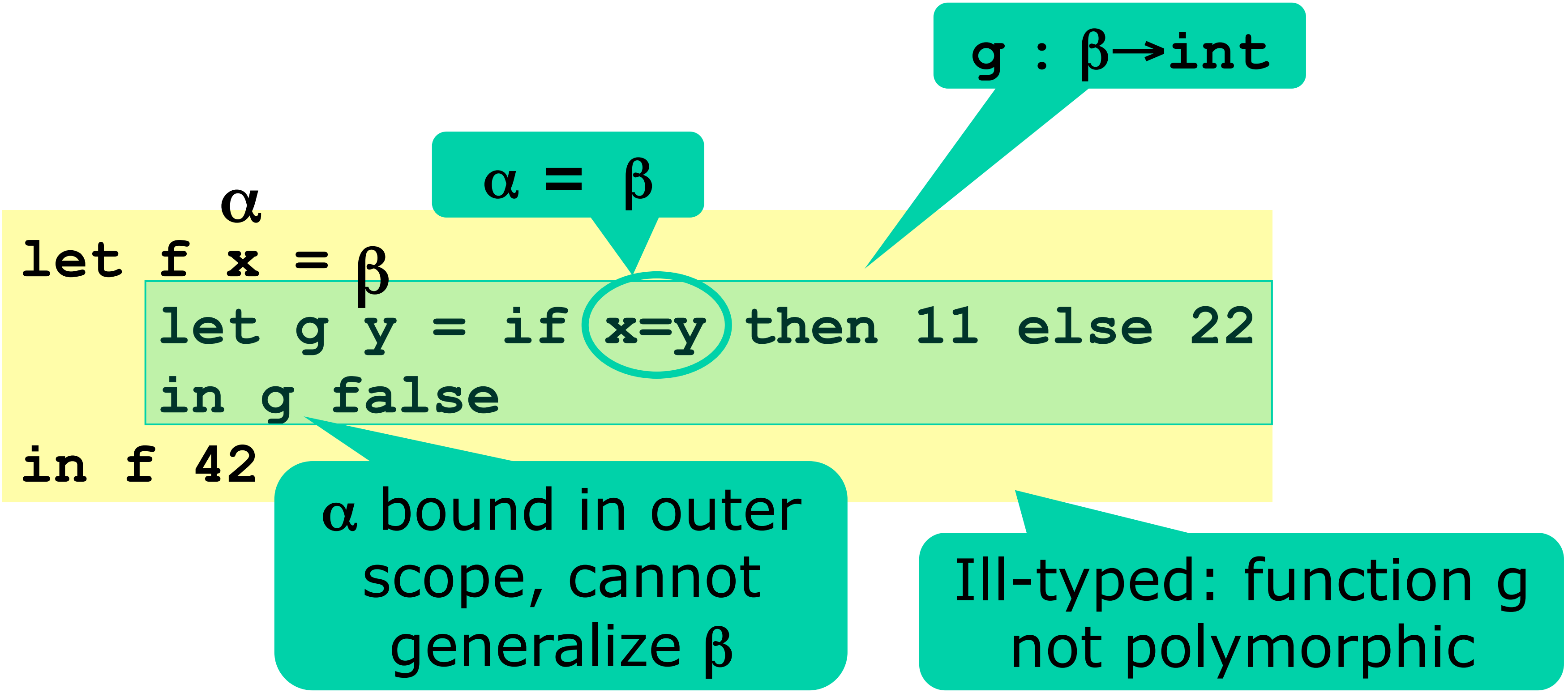
f not polymorphic in its own body

```
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



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

g : β→int

α = β

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
```

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
- But types can be large, 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

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!