Lecture 16

Unification Static Types

type inference as constraint solving with Prolog

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2012
UC Berkeley

Type inference

In OO static types like Java’s, programmers annotate variables/parameters with types.

Foo myFunction(Bar b)

Why ask for these annotations when the type can be (often) inferred automatically?

today we will look at one such type inference

Example

Consider this factorial program.

```
def fact(n):
if (n==0) { 1 } else { n * fact(n-1) }
```

Let's *type* this function. *Typing a function* includes type inference and type checking. Three questions:

- what is the type of the parameter n?
- what is the return type of fact?
- is the function type safe, ie will it perform only operations sanctioned by their type?

Let’s write type rules of our arithmetic

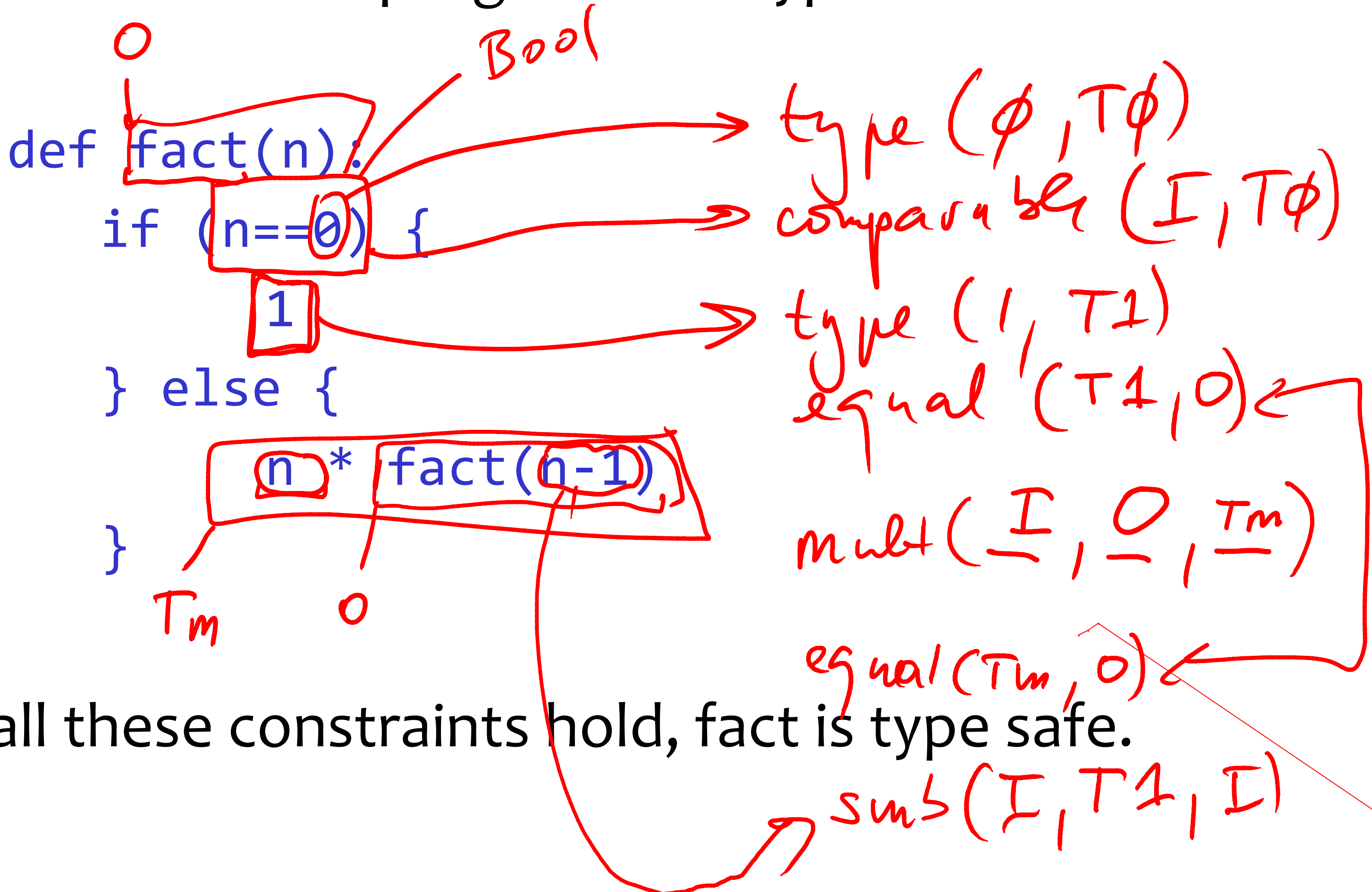
Conveniently, we will use Prolog:

```
type(0,int).                % 0 is an int value
type(1,int).
mult(int,int,int).          % E * E
mult(float,float,float).
sub(int,int,int).           % E - E
sub(float,float,float).
comparable(int,int).        % E == E
comparable(float,float).
```

These rules hold for all programs in our language.

Collect constraints from the program

Now translate a program into type constraints.



If all these constraints hold, fact is type safe.

Constraints for the factorial function

I → O

```
fact(fun(I,O)) :-          % I is the type of n
    type(0,T0),            % T0 is the type of value 0
    type(1,T1),            % T1 is the type of value 1
    comparable(I,T0),      % is n==0 legal?
    T1=O,                  % type(1) must equal ret type
    mult(I,O,O),
    sub(I,T1,I).           % (2)
```

Solving type constraints

We ask Prolog to solve these constraints.

```
?- fact(fun(I,O)).
I = int
O = int
```

There is a solution to these constraints, so fact is type safe when if called with the parameter type I=int.

We also learn that it will return value of type O=int.

Notes

fun(I,O) is our (Prolog) way of denoting the function type. The usual notation is I -> O

How do we know that the return type of fact(n-1) is O?

We have decided that fact has the same type in each invocation, hence the type fact(n-1) must be the same as that of fact(n), which we denoted O.

ML

A language that has influenced moder static languages, such as Scala.

ML is based on unification type system, like we used in our fact example.

Let's look at ML's type more closely.

Function definition are composed of cases

Function definition:

```
fun fact 0 = 1
  | fact n = n * fact(n-1);
```

If the definition type checks, the compiler accepts the definition, and prints out:

```
val fact = fn : int -> int
```

Lists

The cons operator

```
1::2::3::[]
```

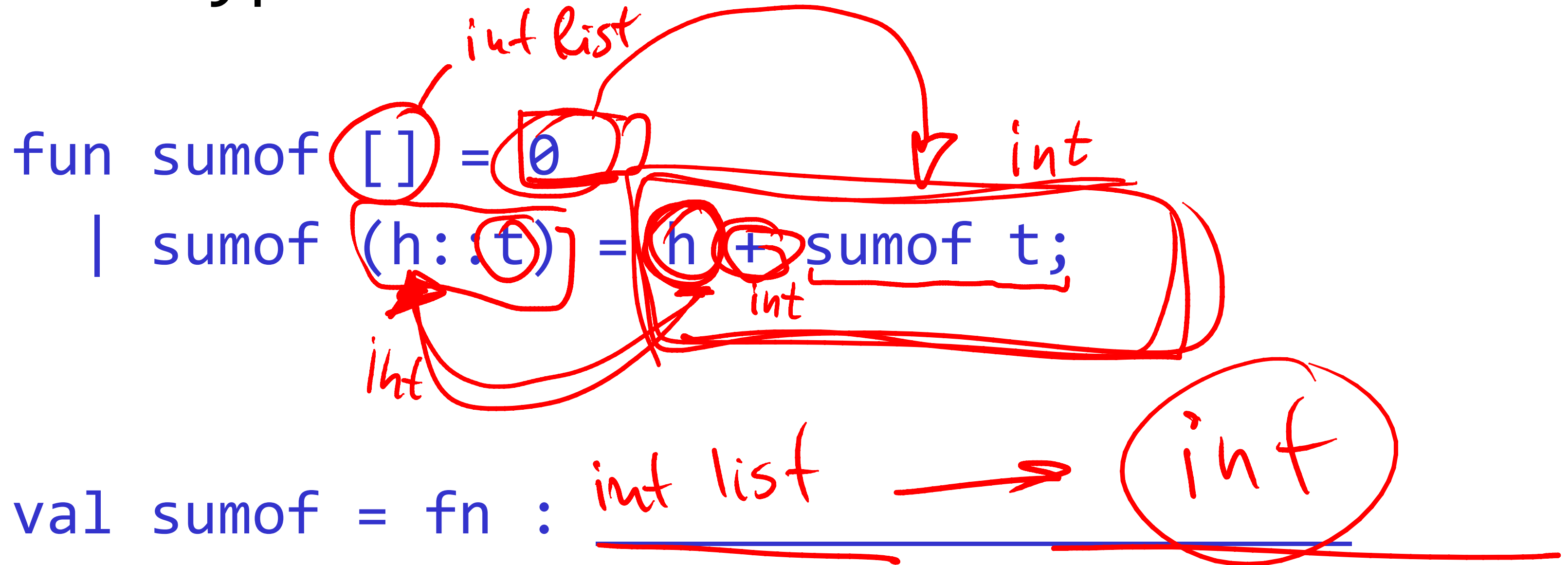
is the same as

```
[1,2,3]
```

The :: operator is a binary function with infix syntax.

Type inference for lists

Let's type this recursive function:



More list type inference

What's different in this function definition?

```
fun map(f, [])      = []
  | map(f,(h::t)) = f(h)::(map(f, t));
```

map $(\alpha \rightarrow \beta * \alpha \text{ list}) \rightarrow (\beta \text{ list})$

Two usage examples

```
map (sqrt, [1.0,2.0,3.0])

map (rev, [[1,2,3],[4,5,6],[7,8,9]])
```

Polymorphic types

What would be the type of function :: ?

```
'a * 'a list -> 'a list
```

α α α

:: is a polymorphic function

Let’s work out the type inference

```
fun map(f, [])      = []
  | map(f,(h::t)) = f(h)::(map(f, t));
```

Another version of map

```
fun map f [] = []
  | map f (h::t) = (f h)::(map f t);

val map = fn : ('a -> 'b) -> ('a list -> 'b list)

val sqrtall = map sqrt;
val sqrtall = fn : real list -> real list

sqrtall [1.0,4.0,9.0];
val it = [1.0, 2.0, 3.0] : real list
```