# CSCI 334:
# Principles of Programming Languages

## Lecture 17: HOFs / Type inference

Instructor: Dan Barowy

Williams

## Topics

Higher order functions

Type inference

## Your to-dos

1. Lab 7, **due Sunday 4/17** (partner lab)
2. Reading response, **due Wednesday 4/20**.

## Quiz

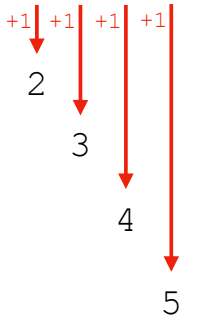
# Mapping and Folding

## map

Intuition:

## map

```
List.map (fun x -> x + 1) [1;2;3;4];
```




```
[2;3;4;5]
```

## map

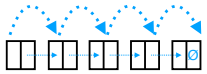
```
[2;8;22;4]
|> List.map (fun x -> x + 1)
|> List.map float
|> List.map (fun x -> x / 3.3)
|> List.sort
```

```
[0.9090909091; 1.515151515; 2.727272727;
6.96969697]
```

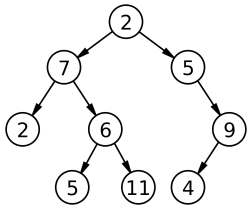
# fold

structural recursion → fold it!

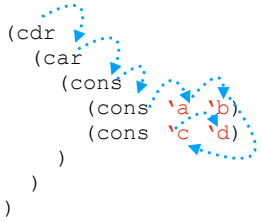
(in a nutshell: any problem that recurses on a subset of input)



list length




tree height




evaluation

# fold

Intuition:

# fold left

```
List.fold (fun acc x -> acc+x) 0 [1;2;3;4]
```

```
acc = 0, [1;2;3;4]
acc = 0+1, [2;3;4]
  acc = 1+2, [3;4]
    acc = 3+3, [4]
      acc 6+4, []
returns acc = 10
```

# fold right

```
List.foldBack
  (fun x acc -> acc+x) [1;2;3;4] 0
```

```
[1;2;3;4], acc = 0
[1;2;3], acc = 0+4
[1;2], acc = 4+3
[1] acc = 7+2
[], acc = 9+1
returns acc = 10
```

## Try this at home!

```
let number_in_month(ds: Date list)(month: int) : int =
```

- Write a function `number_in_month` that takes a list of dates (where a date is `int*int*int` representing year, month, and day) and an `int month` and returns how many dates are in `month`
- Use `List.fold`

## Type checking & type inference

Finally—cool things enabled by the lambda calculus!

## type inference

Not everybody loves this part of PL.

I hope that you can appreciate the absence of magic.

## Type checking

(or, "how does my compiler know that my expression is wrong?")

```
let f(x:int) : int = "hello" + x
```

```
  let f(x:int) : int = "hello" + x;;
  ------------------------------^

stdin(1,32): error FS0001: The type 'int' does not
match the type 'string'
```

# A note about “curried” expressions

```
let f(a: int, b: int, c: char) : float = …

    f is a:int * b:int * c:char -> float

let f(a: int)(b: int)(c: char) : float = …

      f is int -> int -> char -> float

            let f a b c = …

            f = λa.λb.λc.…
```

# Type checking

## step 1: convert into lambda form

```
let f(x:int) : int = “hello” + x
f = λx.“hello ” + x
f = λx.(+ “hello ” x)
```

convert into λ expression

assume + = $\lambda x.\lambda y.(x + y)$

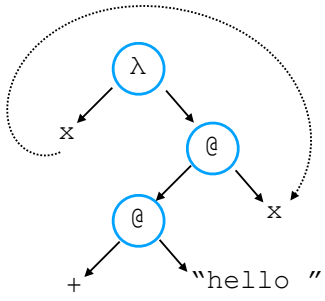
The purpose of this step is to make all of the parts of an expression clear

# Type checking

## step 2: generate parse tree

```
f = λx.((+ “hello ”) x)
```

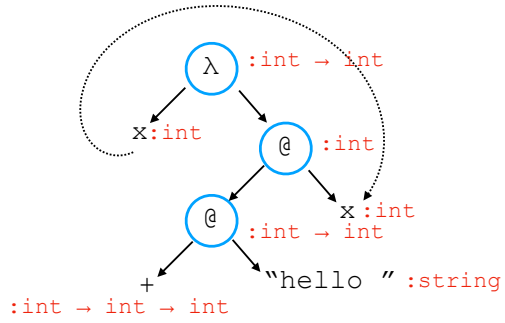
f has form `λx.((EE)E)`




# Type checking

## step 3: label parse tree with types

read “:” as “has type”



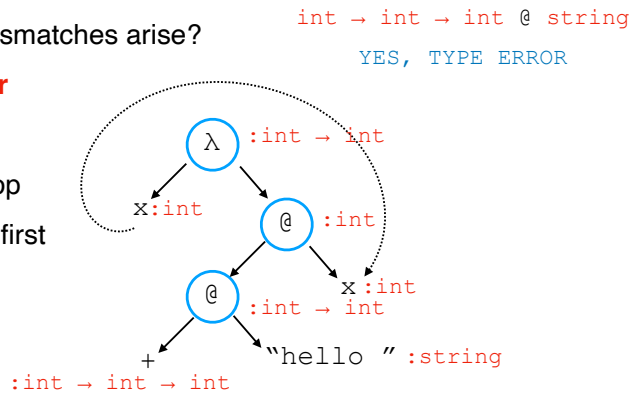
# Type checking

step 4: check that types are used consistently

1. Start at the leaves
2. Do type mismatches arise?

   Yes = **error**

   No = **ok**
3. if **error**, stop and report first mismatch




# Type inference

notice that we had a typed expression

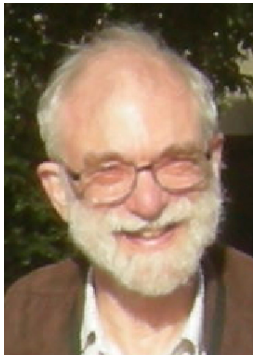
```
let f(x:int) : int = "hello " + x
```

what if, instead, we had

```
let f(x) = "hello " + x
```

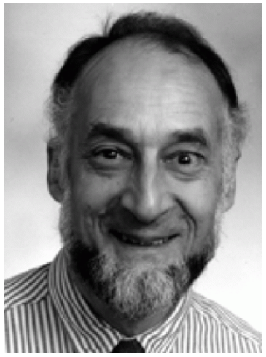
?

# Hinley-Milner algorithm



J. Roger Hindley

- Hindley and Milner invented algorithm independently.
- Infers types from known data types and operations used.
- Depends on a step called "unification".
- I will demonstrate informal method for unification; works for small examples



Robin Milner

# Hinley-Milner algorithm

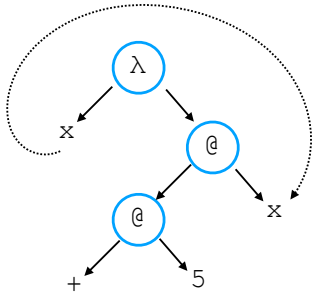
Has three main phases:

1. Assign known types to each subexpression
2. Generate type constraints based on rules of λ calculus:
   a. Abstraction constraints
   b. Application constraints
3. Solve type constraints using unification.

# Type inference

step 1: convert to lambda AST

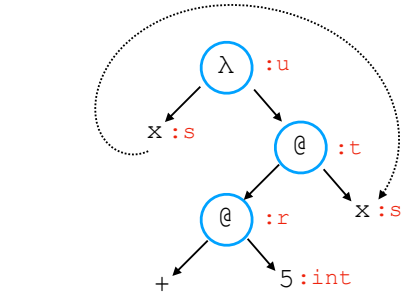
```
let f(x) = 5 + x
f = λx.((+ 5) x)
```




# Type inference

step 2: label parse tree with known/unknown types

```
let f(x) = 5 + x
f = λx.((+ 5) x)
```




:int → int → int

# Type inference

it is often helpful to have types in tabular form

| subexpression | type |
|---|---|
| + | int → int → int |
| 5 | int |
| (+5) | r |
| x | s |
| (+5)x | t |
| λx.((+ 5) x) | u |

# Type inference

step 3: generate constraints

```
<expr> ::= <var>          variable
    |   λ<var>.<expr>     abstraction
    |   <expr><expr>      function application
```

Three rules, each corresponding to a kind of λ expression.

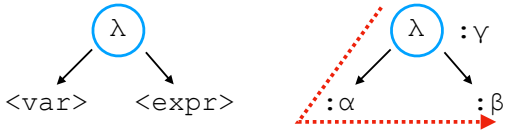
# 3.1. <var> constraint

No constraint.

# 3.2. abstraction constraint

λ<var>.<expr>

“left triangle rule”




Constraint: If the type of <var> is α and the type of <expr> is β, and the type of λ is γ, then the constraint is γ = α → β.

# 3.3. application constraint

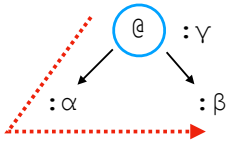
<expr><expr>

“right triangle rule”

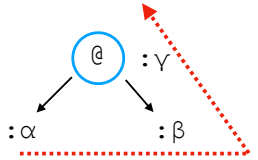

Constraint: If the type of <expr1> is α and the type of <expr2> is β, and the type of @ is γ, then the constraint is α = β → γ.

# constraints summary

Abstraction: If the type of <var> is a and the type of <expr> is b, and the type of λ is c, then the constraint is c = a → b.




Application: If the type of <expr1> is a and the type of <expr2> is b, and the type of @ is c, then the constraint is a = b → c.



# Type inference

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r | int → int → int = int → r |
| x | s | n/a |
| (+5)x | t | r = s → t |
| λx.((+ 5) x) | u | u = s → t |

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r | int → int → int = int → r |
| x | s | n/a |
| (+5)x | t | r = s → t |
| λx.((+ 5) x) | u | u = s → t |

Start with the topmost unknown. What do we know about r?

int → int → int = int → r
r = int → int

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int → int → int = int → int → r |
| x | s | n/a |
| (+5)x | t | r = s → t |
| λx.((+ 5) x) | u | u = s → t |

Eliminate r from the constraint.

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s | n/a |
| (+5)x | t | int → int = s → t |
| λx.((+ 5) x) | u | u = s → t |

Eliminate r from the constraint.

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s | n/a |
| (+5)x | t | int → int = s → t |
| λx.((+ 5) x) | u | u = s → t |

What do we know about s and t?

int → int = s → t
s = int
t = int

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s = int | n/a |
| (+5)x | t = int | int → int = s → t |
| λx.((+ 5) x) | u | u = s → t |

Eliminate s and t from constraint.

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s = int | n/a |
| (+5)x | t = int | int → int = int → int |
| λx.((+ 5) x) | u | u = int → int |

What do we know about u?

u = int → int

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s = int | n/a |
| (+5)x | t = int | int → int = int → int |
| λx.((+ 5) x) | u = int → int | u = int → int |

Eliminate u from constraint.

# Type inference

## step 3: unify

| subexpression | type | constraint |
|---|---|---|
| + | int → int → int | n/a |
| 5 | int | n/a |
| (+5) | r = int → int | int→int→int = int→int→int |
| x | s = int | n/a |
| (+5)x | t = int | int → int = int → int |
| λx.((+ 5) x) | u = int → int | int → int = int → int |

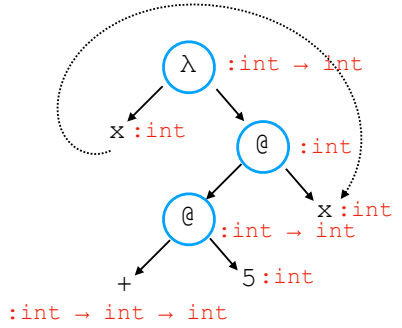
Done when there is nothing left to do.

Sometimes unknown types remain.

An unknown type means that the function is polymorphic.

# Completed type inference

```
let f x = 5 + x
f = λx.((+ 5) x)
```




# Recap & Next Class

## Today:

Higher order functions in F#

Type inference

## Next class:

More type inference

Parsing