# Type Inference

CSCI 334

Stephen Freund



## Type Checking

- Dynamic
  - + **Prevents Errors**
  - - run-time overhead
  - - catch errors only at run time
  - - coverage: only catch errors on paths executed
  - + **more flexible**
- Static
  - + **Prevents Errors**
  - + **no run-time cost**
  - + **catch errors at compile time**
  - + **coverage: catches errors on all possible paths**
  - - restricts flexibility



## Limitations of Static Type Checking

- Types restrict expressiveness:
  - Lisp: `'(1 "cow" 4.5)`
  - ML: `[1,2,3] : int list`
  - `zip_any` in ML
- Decidability

```
fun f() {
  while (big-test) {
    if (big-test-2) {
       return 3;
    }
  }
  return 3 + true;
}
```

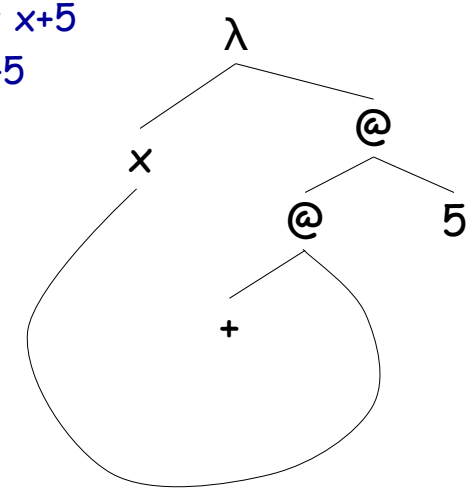
```
fun h(int x) {
  int a[] = new int[10];
  int y = (x*x+sqrt(x)*2)/g(x);
  a[y] = 3;
}
```



## Checking vs. Inference

- Type Checking
  - `int f(int x) { return x + 1; }`
  - `int g(int y) { return f(y) * 2; }`
- Type Inference
  - `fun f(x) = x + 1;`
  - `fun g(y) = f(y) * 2;`

fun g(x) = x+5
g = λx.x+5




5

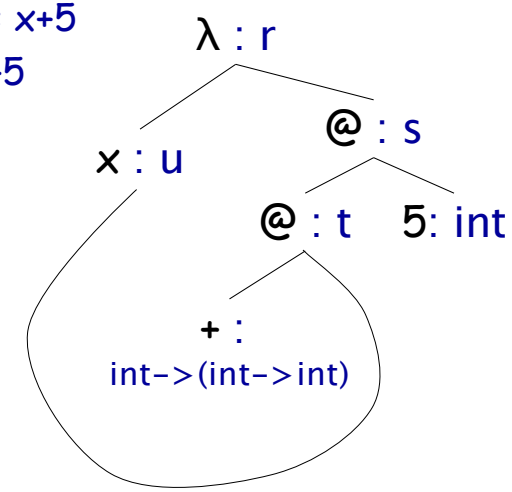
## Hindley-Milner Algorithm

- **Goal:** Annotate each node in tree with a type

1. Annotate nodes with
   - actual type for constants, ops, known names
   - unique type variable for all others
2. Generate constraints over type variables using structure of tree.
3. Solve constraints via unification

6

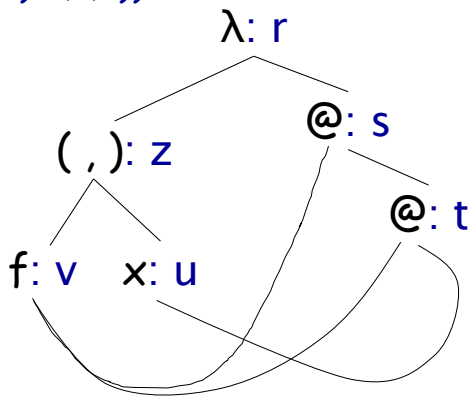
fun g(x) = x+5
g = λx.x+5




Step 1

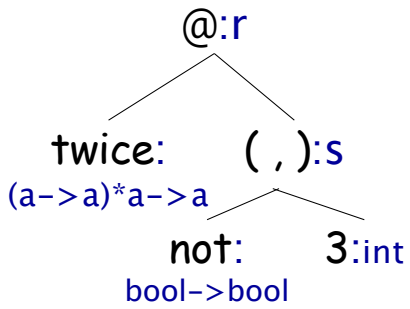
7

fun twice(f,x) = f(f(x))
twice = λ(f,x).f(f(x))




8

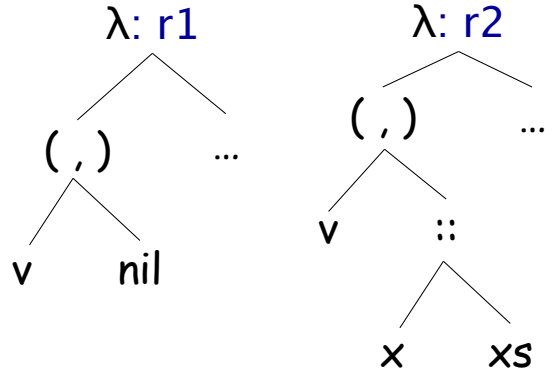
not: bool -> bool
twice(not, 3);




s = (bool -> bool) * int
(a -> a) * a -> a = s -> r
(a -> a) * a = s
a = r
(a -> a) * a = (bool -> bool) * int
(a -> a) = (bool -> bool)
a = int
a = bool



fun search(v,nil) = …
| search(v,x::xs) = …






## Java Autoboxing

```
class Vector<E> {
   E elementData[];
   int elementCount;

   void add(E o) {
      elementData[elementCount++] = o;
   }
}

Vector<Integer> v = new Vector <Integer>();
v.add(3);
...
println(v.get(0) * 2);
```



## Java Reality: Boxing/Unboxing

```
class Vector {
   Object elementData[];
   int elementCount;

   void add(Object o) {
      elementData[elementCount++] = o;
   }
}

Vector v = new Vector();
v.add(new Integer(3));
...
println(((Integer)v.get(0)).intValue()*2);
```

## C++ Templates

```
template <typename T> void sort(T d[], int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (d[j] < d[j+1]) {
        T temp = d[j];
        d[j] = d[j+1];
        d[j+1] = temp;
      }
    }
  }
}

int a[100];
char b[100];
double c[100];
sort(a,100);
sort(b,100);
sort(c,100);
```

13

## add for ints

```
template <typename T> T add(T x, T y)
  { return x+y; }

add(1,2);

add<int>:
 pushl    %ebp
 movl     %esp, %ebp
 movl     12(%ebp), %eax
 addl     8(%ebp), %eax
 leave
 ret
```

14

## add for reals

```
add(1.3,2.3);

add<double>:
  pushl %ebp
  movl    %esp, %ebp
  movsd %xmm0, -8(%rbp)
  movsd %xmm1, -16(%rbp)
  movsd -8(%rbp), %xmm0
  addsd -16(%rbp), %xmm0
  popq %rbp
  retq
```

15

## Code Specialization

```
int f(int g, int r, int x, int y) {
  return g*g*g*x + y/(r*r);
}


int main() {
 for (int i = 0; i < 50000000; i++) {
    int a = f(100, 100, i, 30);
    int b = f(10, 90, 1, 3);
    int c = f(20, 20, i, 3);
    …
  }
}
```

16

## Code Specialization

```
template <int G, int R> int f(int x, int y) {
  return G*G*G*x + y/(R*R);
}


int main() {
  for (int i = 0; i < 50000000; i++) {
    int a = f<100,100>(i,30);
    int b = f<10,90>(1,3);
    int c = f<20,20>(i,3);
    …
  }
}
```

17

## Code Specialization

```
template <int G, int R> int f(int x, int y) {
  return G*G*G*x + y/(R*R);
}

int main() {
  for (int i = 0; i < 50000000; i++) {
    int a = f0(i,30);
    int b = f1(1,3);
    int c = f2(i,3);
    …
  }
}
```

```
f0(int x, int y) {
  return 1000000*x + y/10000;
}
```

```
f2(int x, int y) {
  return 8000*x + y/400;
}
```

18