TYPE CHECKING/INFERENCE FOR SEQUENTIAL PROGRAMS

① define a simple C-like language

② define a type inference algorithm

Based on Ch. 2/3 of SPA

---

a program $P \rightarrow F \; F \; F \; \ldots \; F$

a function $F \rightarrow X(X, \ldots, X) \{ \text{var } X, \ldots, \text{var } X;$
$S; S; \ldots S;$
$\text{return } E$
$\}$

name of variable

expression

a statement $S \rightarrow X = E$
$\mid \text{if } (E) \{ S; \ldots; S \} \text{ else } \{ S; \ldots; S \}$
$\mid \text{while } (E) \{ S; \ldots; S \}$
$\mid {*}X = E$

an expression $E \to I$ // integer

$| \; X$

$| \; E + E \; | \; E - E \; | \; E * E \; | \; \ldots$

$| \; E == E \; | \; E > E \; | \; \ldots$

$| \; X(E, \ldots, E)$

a variable $X \to x \; | \; y \; | \; z \; | \; \ldots$

$E \to \text{alloc } E$

$| \; \& X$

$| \; * E$

$| \; \text{null}$

E.g.

```
iterate (n) {
  var f;
  f = 1;
  while (n>0) {
     f = f*n;
  }
  return f; }
```

no type

$X = 1 + Y + Z + 3 \ldots$

$X_1 = 1 + Y$

$X_2 = X_1 + Z$

$X_3 = X_2 + 3$

$\vdots$

What checks should the type system do?

- arithmetic operations should be only over integers
- conditions of if/while should be integers
- only integers can be args/return of "main"
- * only applies to pointers/null
- call a function with the right types

::=

$$
\begin{aligned}
T \to\ & \text{int} \\
\mid\ & \&T \\
\mid\ & \underbrace{(T, \ldots, T)}_{\text{arguments treated as a tuple}} \to T
\end{aligned}
$$

Recall

$T \to T \to T \to T$

in λ calculus

E.g.:

int

& int

(int, int) → & int

Recall constraints

$S = T$

$[\![\cdot]\!]$ type variable

$[\ \ ]$ for simplicity

---

generating type constraints

I	$[I] = \text{int}$
$E_1 == E_2$	$[\![E_1 == E_2]\!] = \text{int}$ $\quad [\![E_1]\!] = [\![E_2]\!]$
	$[E_1] = \text{int} \quad [E_2] = \text{int}$
$E_1 \text{ op } E_2$	$[E_1 \text{ op } E_2] = [E_1] = [E_2] = \text{int}$
$X = E$	$[X] = [E]$
if $(E)\{S..S\}$ else...	$[E] = \text{int}$
while $(E)\{...\}$	$[E] = \text{int}$
$X(X_1 \ldots X_n)\{\ldots \text{return } E\}$	$[\![X]\!] = ([\![X_1]\!] \ldots [\![X_n]\!]) \to [E]$
$X(E_1 \ldots E_n)$	$[\![X]\!] = ([\![E_1]\!] \ldots [\![E_n]\!]) \to [\![X(E_1 \cdots E_n)]\!]$

alloc E	[[alloc E]] = & [[E]]
& X	[&X] = & [X]
*E	[E] = & [*E]
*X = E	[X] = & [E]
null	[[null]] = & α

new type variable

(*E]
=
;

e.g.

```
short() {
    var x, y, z;
    x = 1
    y = alloc x
    *y = x
    z = *y
    return z
}
```

[short] = () → [z]
[I] = int
[x] = [1]
[alloc x] = & [x]
[y] = [alloc x]
[y] = & [x]
[z] = [*y]
[y] = & [*y]

$[\![short]\!] = () \rightarrow int$

$[\![x]\!] = int$

$[\![y]\!] = \& int$

$[\![z]\!] = int$

E.g.

```
f() {
    var x;
  → x = alloc 17;   → x must &int
    x = 42;         → x must be int
    return x + 12;
}
```

We reject this program because our type system is <u>flow insensitive</u>

```
baz() {
    var x;   x=1;
    return &x;
}

main() {
    var p;
    p= baz();
    *p = 1;
    return *p;
}
```

X sits on the stack and is deallocated when baz terminates

$\boxed{\text{lifetimes}}$

in RUST