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 $P \rightarrow F\ F\ F \cdots F$

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

name of variable — expression

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

an expression $E \rightarrow I$ // integer

$| X$

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

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

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

a variable $X \rightarrow x \mid y \mid z \mid \ldots$

$E \rightarrow$ alloc E

$| \; \& X$

$| \; * E$

$| \;$ null

---

e.g.

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

integer

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

$X_1 = 1 + Y$

$X_2 = X_1 + Z$

$X_3 = X_2 + 3$

$\vdots$

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

::=

$T \rightarrow$ int

| &T

| $(T, \ldots, T) \rightarrow T$

arguments treated as a tuple

Recall

$T \rightarrow T \rightarrow T \rightarrow T$ in λ calculus

E.g.

int

& int

(int, int) → & int

Recall constraints

$S = T$

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

$[\ ]$ for simplicity

---

generating type constraints

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

alloc E	$[\![\text{alloc } E]\!] = \&[\![E]\!]$
&X	$[\&X] = \&[X]$
*E	$[E] = \&[*E]$
*X = E	$[X] = \&[E]$
null	$[\![\text{null}]\!] = \&\alpha$

α: new type variable

$[*E] = i$

E.g.

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

$[\text{short}] = () \to [z]$

$[I] = \text{int}$

$[x] = [I]$

$[\text{alloc } x] = \&[x]$

$[y] = [\text{alloc } x]$

$[y] = \&[x]$

$[z] = [*y]$

$[y] = \&[*y]$

[short] = () → 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 flow insensitive

```
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

lifetimes in RUST