Type inference: basic idea

- infer(e: expression) : typ =
  - Call infer recursively on subexpressions of e
  - Figure out the type of e from the types of subexpressions
    - Use unification to enforce any constraints on types
    - If we ever don’t know the type of something, make a new unification variable

```
int + [ ] ?1 list
    |        |         |
int 1 2 int
```

Lecture 16
We need to know the types of variables

```
let x = 1 in
let y = 2 in
x + y
```
A *context* keeps track of the types of variables

- **infer(ctx: context, e: expression) : typ =**
  - Call infer recursively on subexpressions of e
  - Figure out the type of e from the types of subexpressions
    - Use unification to enforce any constraints on types
    - If we see a variable, look it up in ctx (if not in ctx, it’s unbound)
    - If we ever don’t know the type of something, make a new unification variable

- How do we represent a context?
  - Map? Association list?
We need to know the types of variables

```
let x = 1 in
let y = 2 in
x + y
```

Context does not store the values of variables! We’re not computing anything here!
How do we compute the type of $e$ from the types of subexpressions?

• Depends on what $e$ is.
• Example: $e = e_1 + e_2$
• Remember: $e_1 + e_2$ has type int if $e_1$, $e_2$ have type int
  • Let $t_1 = \text{infer}(\text{ctx}, e_1)$
  • Let $t_2 = \text{infer}(\text{ctx}, e_2)$
  • Unify($t_1$, int)
  • Unify($t_2$, int)
  • (If neither unification failed) return int
We also need to keep track of substitutions

- \( \text{infer([x -> ?1], x + \text{List.length } x)} \)
- \( \text{infer([x -> ?1], x) = ?1} \)
- \( \text{infer([x -> ?1], \text{List.length } x) = \text{int}} \quad \text{unify(?1, ?2 list)} \)
- \( \text{unify(?1, int)} \)
- \( \text{unify(int, int)} \)
- \( \text{return int} \)

- But ?1 can’t be int and ?2 list!
Infer should also return a substitution

- \( \text{infer([x -> ?1], x + \text{List.length x})} \)
- \( \text{infer([x -> ?1], x) = (?1, [])} \)
- \( \text{infer([], x - ?1, \text{List.length x}) = (\text{int, [(?1, ?2 list)])}} \)

\( \text{unify([[(?1, ?2 list)]?1, [[(?1, ?2 list)]int) = unify(?2 list, int) \rightarrow \text{Shape Mismatch}} \)
Infer should also return a substitution

- \( \text{infer}([x \rightarrow ?1], \text{List.length } x)::x) \)
- \( \text{infer}([x \rightarrow ?1], \text{List.length } x) = (\text{int}, [(?1, ?2 \text{ list})]) \)
- \( \text{infer}([[(?1, ?2 \text{ list})]][x \rightarrow ?1], x) = \text{infer}([x \rightarrow ?2 \text{ list}], x) = (?2 \text{ list}, []) \)
- \( \text{unify}([], \text{int list}, ?2 \text{ list}) = [(?2, \text{int})] \)
- \( \text{return} (\text{int list}, [] @ [(?1, ?2 \text{ list})] @ [(?2, \text{int})]) = (\text{int list}, [(?1, ?2 \text{ list}); (?2, \text{int})]) \)

Append all substitutions at the end
A context keeps track of the types of variables

- `infer(ctx: context, e: expression) : typ * subst =`
  - Call `infer` recursively on subexpressions of `e`
    - Need to apply previous substitutions to `ctx`
  - Figure out the type of `e` from the types of subexpressions
    - Use unification to enforce any constraints on types
    - If we see a variable, look it up in `ctx` (if not in `ctx`, it’s unbound)
    - If we ever don’t know the type of something, make a new unification variable

- How do we represent a context?
  - Map? Association list?