Strategy for Inferring Polytypes

1. Assign symbolic type names $t_1, t_2, \ldots$ to all subexpressions of the AST of the function definition.
    - Constant nodes receive known type names (possibly polymorphic).
    - Two uses of a common declaration receive the same type name.

Strategy for Inferring Polytypes

1. Assign symbolic type names $t_1, t_2, \ldots$ to all subexpressions of the AST of the function definition.
   - Constant nodes receive known type names (possibly polymorphic).
   - Two uses of a common declaration receive the same type name.
2. Form a system of equations among unknown *type variables,* following the language's typing rules "in reverse".
   - For the judgment $\frac{E \vdash f : T \to U, E \vdash e : T}{E \vdash f(e) : U}$,
     if $f, e, \mathbf{apply}$ have been assigned symbolic type names t_1, t_2, t_3,
     then add the equation $t_1 = t_2 \to t_3$.
   - For the judgment $\frac{E \vdash e_0 : \mathbf{bool}, E \vdash e_1 : T, E \vdash e_2 : T}{E \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2}$,
     if $e_0, e_1, e_2, \mathbf{if}$ have been assigned symbolic type names t_0, t_1, t_2, t_3,
     then add the equations $\{t_0 = \mathbf{bool}, t_1 = t_2, t_3 = t_1\}$.

Strategy for Inferring Polytypes

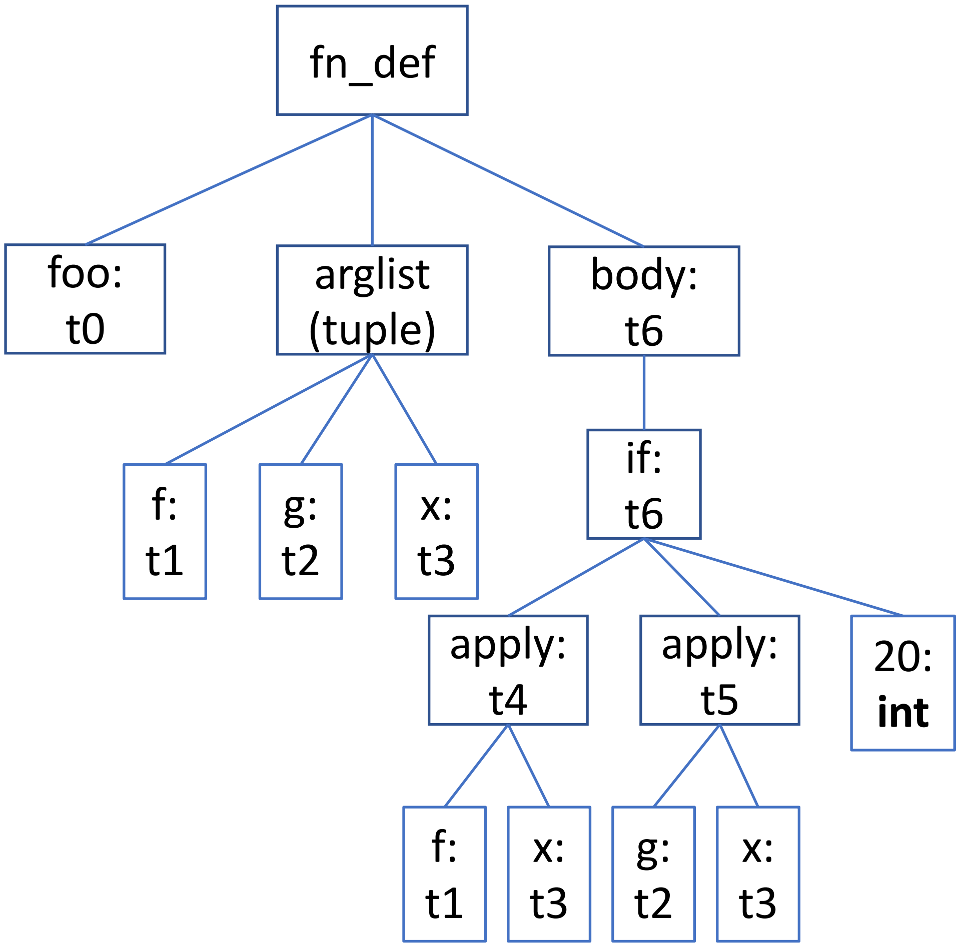
1. Assign symbolic type names t_1, t_2, ... to all subexpressions of the AST of the function definition.
    - Constant nodes receive known type names (possibly polymorphic).
    - Two uses of a common declaration receive the same type name.
2. Form a system of equations among unknown *type variables,* following the language's typing rules "in reverse".
    - For the judgment $\frac{E \vdash f : T \to U, E \vdash e : T}{E \vdash f(e) : U}$,
      if $f, e, \mathbf{apply}$ have been assigned symbolic type names t_1, t_2, t_3, then add the equation $t_1 = t_2 \to t_3$.
    - For the judgment $\frac{E \vdash e_0 : \mathbf{bool}, E \vdash e_1 : T, E \vdash e_2 : T}{E \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2}$,
      if $e_0, e_1, e_2, \mathbf{if}$ have been assigned symbolic type names t_0, t_1, t_2, t_3, then add the equations $\{t_0 = \mathbf{bool}, t_1 = t_2, t_3 = t_1\}$.
3. Solve the resulting system of equations for the unknown type variables.

Example of The Strategy in Action

- Consider the function definition
  `foo f g x = if f(x) then g(x) else 20`

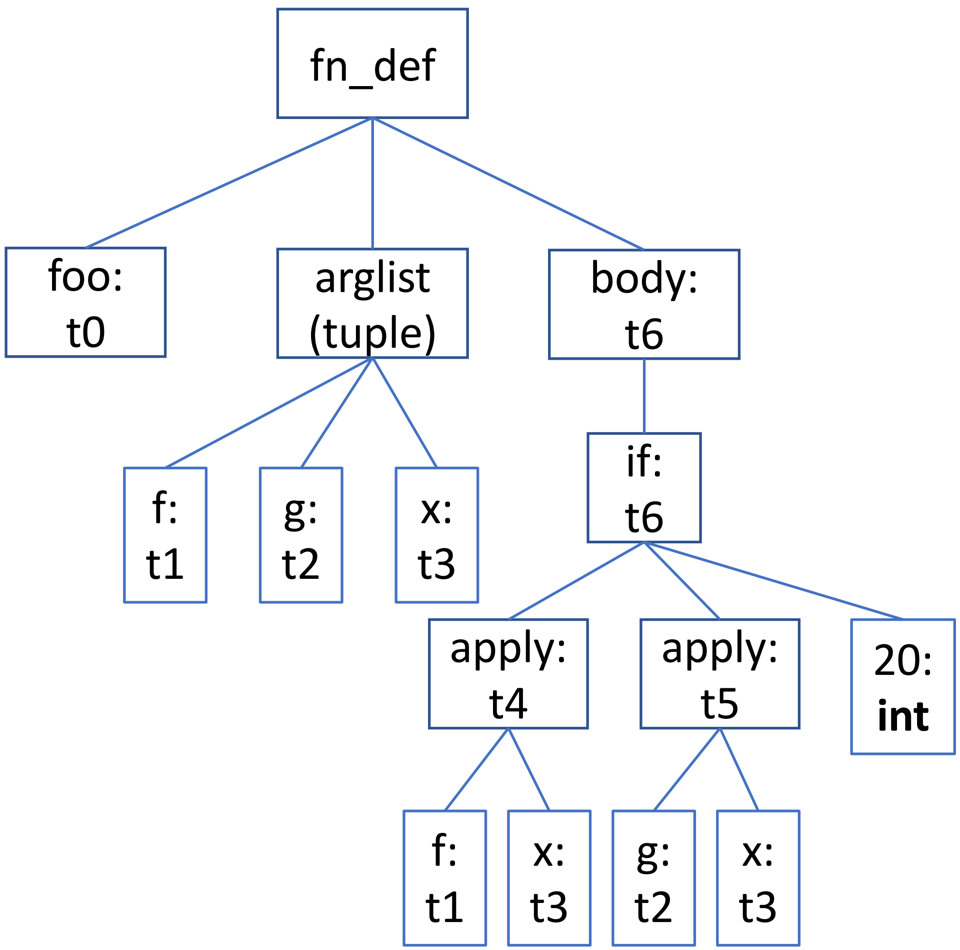
Example of The Strategy in Action

- Consider the function definition
  `foo f g x = if f(x) then g(x) else 20`



Example of The Strategy in Action

- Consider the function definition
  `foo f g x = if f(x) then g(x) else 20`



t0 = t1 * t2 * t3 -> t6

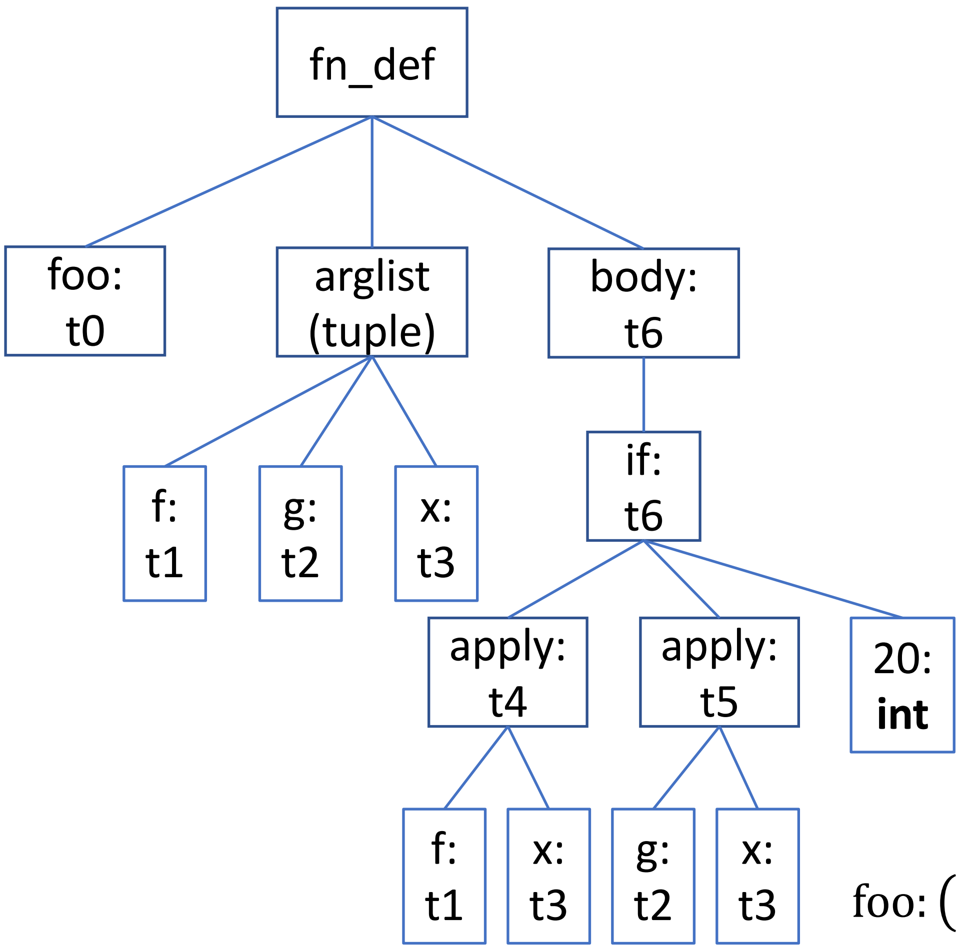
t4 = **bool**
t5 = **int**
t6 = t5

t1 = t3 -> t4

t2 = t3 -> t5

Example of The Strategy in Action

- Consider the function definition
  `foo f g x = if f(x) then g(x) else 20`



t0 = t1 * t2 * t3 -> t6

t4 = **bool**
t5 = **int**
t6 = t5

t1 = t3 -> t4

t2 = t3 -> t5

t0 = ((t3 -> **bool**) * (t3 -> **int**) * t3) -> **int**

foo: $\left((\tau_3 \rightarrow \mathbf{bool}) \times (\tau_3 \rightarrow \mathbf{int}) \times \tau_3\right) \rightarrow \mathbf{int}$

Solving the System of Type Constraints

- The Hindley-Milner type inference algorithm is a technique for solving such a system of polymorphic type constraints.
  - The type system is sound.
  - The type system is decidable and produces the most general polytype of an expression.

Solving the System of Type Constraints

- The Hindley-Milner type inference algorithm is a technique for solving such a system of polymorphic type constraints.
  - The type system is sound.
  - The type system is decidable and produces the most general polytype of an expression.

- Key ideas
  - Match type operators and instantiate type variables, respecting contextual dependencies, using Robinson's unification algorithm from first-order logic. Unification fails in two situations.
    - Trying to match two different constant types (such as **int** and **bool**) or type operators (such as **array** and **func**).
    - Trying to instantiate a variable to a term containing the variable (such as α and $\alpha \rightarrow \beta$), where a circular structure would be built.

Solving the System of Type Constraints

- The Hindley-Milner type inference algorithm is a technique for solving such a system of polymorphic type constraints.
  - The type system is sound.
  - The type system is decidable and produces the most general polytype of an expression.

- Key ideas
  - Match type operators and instantiate type variables, respecting contextual dependencies, using Robinson's unification algorithm from first-order logic. Unification fails in two situations.
    - Trying to match two different constant types (such as **int** and **bool**) or type operators (such as **array** and **func**).
    - Trying to instantiate a variable to a term containing the variable (such as α and $\alpha \rightarrow \beta$), where a circular structure would be built.
  - There is one crucial extension to the core unification algorithm for handling variable bindings (i.e., declarations): the notion of *generic* and *non-generic* type variables.

Unification As Extended Pattern Matching

- Simpler problem: pattern matching.
  - Given a *constant* type expression C and a *pattern* type expression P, where only P contains type variables, find an assignment U of (constant) terms to type variables that will make the two expressions structurally equivalent.
  - Such an assignment U, which we will write as a map from variables to terms, is also called a *substitution* or a unifier.
  - We will write $P[U]$ for the expression resulting from applying the unifier U to the expression P. So we need to compute U such that $P[U] \equiv_S C$.
  - If U cannot be computed for all type variables, then matching fails.

Unification As Extended Pattern Matching

- Simpler problem: pattern matching.
  - Given a *constant* type expression C and a *pattern* type expression P, where only P contains type variables, find an assignment U of (constant) terms to type variables that will make the two expressions structurally equivalent.
  - Such an assignment U, which we will write as a map from variables to terms, is also called a *substitution* or a unifier.
  - We will write $P[U]$ for the expression resulting from applying the unifier U to the expression P. So we need to compute U such that $P[U] \equiv_S C$.
  - If U cannot be computed for all type variables, then matching fails.

- Examples
  - $C = f\big(a, b, g(t)\big), P = f(a, \alpha, \beta)$. $U = \{\alpha : b, \beta : g(t)\}$.
  - $C = f\Big(h(a), a, g\big(h(a)\big), t\Big), P = f(\alpha, a, g(\alpha), t)$. $U = \{\alpha : h(a)\}$.
  - $C = f\Big(h(b), a, g\big(h(a)\big), t\Big), P = f(\alpha, a, g(\alpha), t)$. $U = \emptyset$.

From Pattern Matching To Unification

- Small change to problem statement: *both* expressions can now contain variables.
  - Given two type expressions T_1 and T_2, where both expressions contain type variables, find a unifier U that will make $T_1[U] \equiv_S T_2[U]$.
  - Now U will be a map from variables to (possibly non-constant) terms.

From Pattern Matching To Unification

- Small change to problem statement: *both* expressions can now contain variables.
  - Given two type expressions T_1 and T_2, where both expressions contain type variables, find a unifier U that will make $T_1[U] \equiv_S T_2[U]$.
  - Now U will be a map from variables to (possibly non-constant) terms.
- Examples
  - $T_1 = f\big(a, \alpha, g(\beta)\big), T_2 = f\big(\beta, k, g(a)\big)$. $U = \{\alpha : k, \beta : a\}$.

From Pattern Matching To Unification

- Small change to problem statement: *both* expressions can now contain variables.
  - Given two type expressions T_1 and T_2, where both expressions contain type variables, find a unifier U that will make $T_1[U] \equiv_S T_2[U]$.
  - Now U will be a map from variables to (possibly non-constant) terms.
- Examples
  - $T_1 = f\big(a, \alpha, g(\beta)\big), T_2 = f\big(\beta, k, g(a)\big)$. $U = \{\alpha: k, \beta: a\}$.
  - $T_1 = f(\alpha, \beta), T_2 = f\big(\gamma, g(\alpha)\big)$.
    - $U_1 = \{\alpha: \gamma, \beta: g(\alpha)\}$, $U_2 = \{\alpha: \delta, \beta: g(\delta), \gamma: \delta\}$, $U_3 = \big\{\alpha: h(\delta), \beta: g\big(h(\delta)\big), \gamma: h(\delta)\big\}$ and many others will unify the terms.

From Pattern Matching To Unification

- Small change to problem statement: *both* expressions can now contain variables.
  - Given two type expressions T_1 and T_2, where both expressions contain type variables, find a unifier U that will make $T_1[U] \equiv_S T_2[U]$.
  - Now U will be a map from variables to (possibly non-constant) terms.
- Examples
  - $T_1 = f\big(a, \alpha, g(\beta)\big), T_2 = f\big(\beta, k, g(a)\big)$. $U = \{\alpha{:}\, k, \beta{:}\, a\}$.
  - $T_1 = f(\alpha, \beta), T_2 = f\big(\gamma, g(\alpha)\big)$.
    - $U_1 = \{\alpha{:}\, \gamma, \beta{:}\, g(\alpha)\}$, $U_2 = \{\alpha{:}\, \delta, \beta{:}\, g(\delta), \gamma{:}\, \delta\}$, $U_3 = \big\{\alpha{:}\, h(\delta), \beta{:}\, g\big(h(\delta)\big), \gamma{:}\, h(\delta)\big\}$ and many others will unify the terms.
  - Unifier U_1 is the simplest, the least constrained, and the most general.
    - That is, there exists another map S_{12} such that $U_2 = U_1 \circ S_{12}$, etc. E.g., $S_{12} = \{\gamma{:}\, \delta\}, S_{13} = \{\gamma{:}\, h(\delta)\}$. But we can't find maps S_{21} or S_{31} that will go the other way.
    - This most general unifier (MGU) U_1 is our desired solution.

Implementing A Unifier

- More-or-less obvious recursive implementation, but need to watch out for some subtle corner cases.
  - See https://github.com/eliben/code-for-blog/blob/master/2018/unif/unifier.py for a simple Python implementation.

Implementing A Unifier

- More-or-less obvious recursive implementation, but need to watch out for some subtle corner cases.
  - See https://github.com/eliben/code-for-blog/blob/master/2018/unif/unifier.py for a simple Python implementation.

```
def unify(x, y, subst):
    if subst is None: return None
    elif x == y: return subst
    elif isinstance(x, Var):
        return unify_variable(x, y, subst)
    elif isinstance(y, Var):
        return unify_variable(y, x, subst)
    elif isinstance(x, App) and isinstance(y, App):
        if x.fname != y.fname
        or len(x.args) != len(y.args):
            return None
        else:
            for i in range(len(x.args)):
                subst = unify(x.args[i], y.args[i], subst)
                return subst
    else: return None
```

```
def unify_variable(v, x, subst):
    assert isinstance(v, Var)
    if v.name in subst:
        return unify(subst[v.name], x, subst)
    elif isinstance(x, Var) and x.name in subst:
        return unify(v, subst[x.name], subst)
    elif occurs_check(v, x, subst):
        return None
    else:
        return {**subst, v.name: x}
```

```
def occurs_check(v, term, subst):
    assert isinstance(v, Var)
    if v == term: return True
    elif isinstance(term, Var) and term.name in subst:
        return occurs_check(v, subst[term.name], subst)
    elif isinstance(term, App):
        return any(occurs_check(v, arg, subst) for arg in term.args)
    else: return False
```

Handling Variable Bindings

- What is the type of this anonymous function (think Python-ish)?

```
lambda f: pair(f(3), f(true))
```

- This expression cannot be assigned a meaningful type without violating the soundness of the type system.
  - The symbol `f` must a function type. From its first use, `f` must have (poly)type $\mathbf{int} \rightarrow \alpha$, while from its second use, it must have (poly)type $\mathbf{bool} \rightarrow \alpha$. But unification fails on these two terms.
  - The type variable α in the type expression of `f` is called *non-generic*, and it must have the same value in all its uses in the expression.

Handling Variable Bindings

- What is the type of this anonymous function (think Python-ish)?

  `lambda f: pair(f(3), f(true))`

- This expression cannot be assigned a meaningful type without violating the soundness of the type system.
  - The symbol `f` must a function type. From its first use, `f` must have (poly)type $\mathbf{int} \to \alpha$, while from its second use, it must have (poly)type $\mathbf{bool} \to \alpha$. But unification fails on these two terms.
  - The type variable α in the type expression of `f` is called *non-generic*, and it must have the same value in all its uses in the expression.
- How about this expression?

  `{f = lambda a: a; pair(f(3), f(true))},`

  which is equivalent to

  `pair((lambda a: a)(3), (lambda a: a)(true)).`

- This expression can be assigned a completely reasonable type.
  - The symbol `f` has the polytype $\alpha \to \alpha$ within the scope of the block.
  - The two uses of `f` within the block can be typed independently (i.e., heterogeneously, if needed).
  - The type variable α in the type expression of `f` is called *generic*, and it can have different values in its multiple uses within the expression.

Handling Variable Bindings

- What is the type of this anonymous function (think Python-ish)?

  `lambda f: pair(f(3), f(true))`

- This expression cannot be assigned a meaningful type without violating the soundness of the type system.
  - The symbol `f` must a function type. From its first use, `f` must have (poly)type $\mathbf{int} \rightarrow \alpha$, while from its second use, it must have (poly)type $\mathbf{bool} \rightarrow \alpha$. But unification fails on these two terms.
  - The type variable α in the type expression of `f` is called *non-generic*, and it must have the same value in all its uses in the expression.
- How about this expression?

  `{f = lambda a: a; pair(f(3), f(true))}`,

  which is equivalent to

  `pair((lambda a: a)(3), (lambda a: a)(true))`.
- This expression can be assigned a completely reasonable type.
  - The symbol `f` has the polytype $\alpha \rightarrow \alpha$ within the scope of the block.
  - The two uses of `f` within the block can be typed independently (i.e., heterogeneously, if needed).
  - The type variable α in the type expression of `f` is called *generic*, and it can have different values in its multiple uses within the expression.
- See https://github.com/eliben/code-for-blog/tree/master/2018/type-inference for a Python implementation of Hindley-Milner type inference.