CSE 304
Compiler Design
Type Checking

YOUNGMIN KWON / TONY MIONE

Type Expressions

Type Expressions

- Basic type or constructed type by applying type constructors to type expressions.

Basic type:

- `boolean, integer, real, type_error, void,…`

Type name is a type expression

Type constructor

- Arrays: if T is a type expression, then array(I,T) is a type expression (I is the size of the array)
  - `array[10] of integer:  array(10, integer)`
- Products: if T1 and T2 are type expressions, then T1 * T2 is a type expression (Cartesian product)
  - `integer * integer`

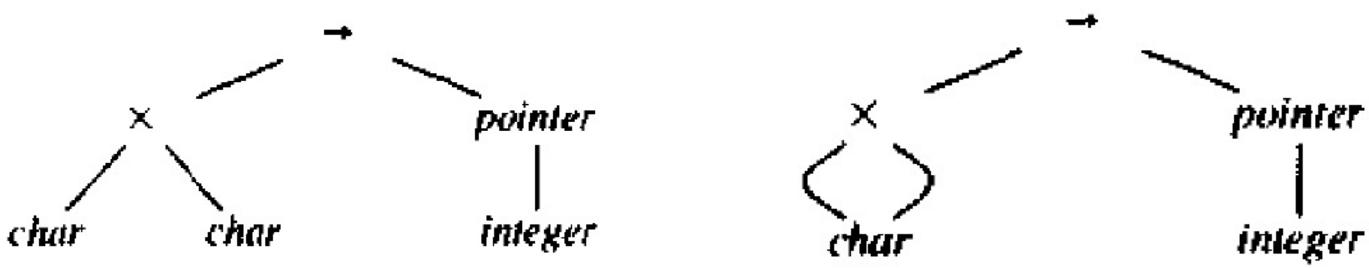
Type Expressions

- Records: similar to products, but the fields have names
  - `record( (row * integer) * (column * integer) )`
- Pointers: if T is a type, then pointer(T) is a type
  - `pointer(integer)`
- Functions: if D and R are types, D -> R is a type
  - `integer * integer -> integer`

Type expressions may contain variables whose values are type expressions.

Tree and dag representation of

`(char * char) -> pointer(integer)`



Type Systems

Type system

- A collection of rules for assigning type expressions to the various parts of a program
- A type checker implements a type system

Type checking

- Static type checking: checking is done by a compiler
- Dynamic type checking: checking is done when the target program runs
- Sound type system: if a type system assigns a type to a program part, type error shouldn't occur for the program part.
- A language is strongly typed: static type checking guarantees that type errors cannot occur during the runtime.
  - Difficult to achieve

```
int a[100];
a[ a_very_complex_function() ] = 1;
```

A Simple Type Checker

A Simple Language

```
P -> D ; E
D -> D ; D | id : T
T -> char | integer
   | array [ num ] of T
   | ^ T
E -> literal | num | id
   | E mod E
   | E [ E ]
   | E ^
```

Type of an Id

A Part of Translation Scheme

```
P -> D ; E
D -> D ; D
D -> id : T   { addtype(id.entry, T.type) }
T -> char     { T.type := char }
T -> integer { T.type := integer }
T -> ^ T1     { T.type := pointer(T1.type) }
T -> array [ num ] of T1 {
                  T.type := array(num.val, T1.type) }
```

Type Checking of Expressions

```
E -> literal    { E.type := char }
E -> num        { E.type := integer }
E -> id         { E.type := lookup(id.entry) }
E -> E1 mod E2 { E.Type :=
    if E1.type = integer and E2.type = integer
    then integer
    else type_error }
E -> E1 [ E2 ] { E.type :=
    if E2.type = integer and E1.type = array(s,t)
    then t
    else type_error }
E -> E1^        { E.type :=
    if E1.type = pointer(t)
    then t
    else type_error }
```

Type Checking of Statements

```
S -> id := E       { S.type :=
    if id.type = E.type
    then void else type_error }
S -> if E then S1  { S.type :=
    if E.type = boolean
    then S1.type else type_error }
S -> while E do S1 { S.type :=
    if E.type = boolean
    then S1.type else type_error }
S -> S1 ; S2       { S.type :=
    if S1.type = void and S2.type = void
    then void else type_error }
```

Type Checking of Functions

Type expression for functions

```
T -> T1 '->' T2 {
     T.type := T1.type -> T2.type }
```

Rules for checking a function application

```
E -> E1 ( E2 ) { E.type :=
     if E2.type = s and E1.type = s->t
     then t else type_error }
```

For more than one arguments

- `T1,… ,Tn` can be views as a single argument type `T1*… *Tn`

Equivalence of Type Expressions

"Two type expressions are equal" what does that mean?

Structural Equivalence

- Two expressions are either the same basic type,
- They are formed by applying the same constructor to structurally equivalent types.

Structural Equivalence

```
function sequiv(s, t): boolean
begin
   if s and t are the same basic type then
      return true
   else if s = array(s1,s2) and t = array(t1,t2) then
      return s1 = t1 and sequiv(s2,t2)
   else if s = s1*s2 and t = t1*t2 then
      return sequiv(s1,t1) and sequiv(s2,t2)
   else if s = pointer(s1) and t = pointer(t1) then
      return sequiv(s1,t1)
   else if s = s1->s2 and t = t1->t2 then
      return sequiv(s1,t1) and sequiv(s2,t2)
   else
      return false

end
```

Names for Type Expressions

In some languages types can have names

```
type link = ^cell;
var next : link;
    last : link;
    p    : ^cell;
    q, r : ^cell;
```

- Allow type expressions to be named, and allow the names to appear in type expressions
- `p: pointer(cell)` type, `next: link` type

Name Equivalence

- Views each type name as a distinct type.
- `last` and `p` are not the same type

Structural Equivalence

- Names are replaced by the type expressions they define.
- `next, last, p, q`, and `r` are the same type

Type Conversions

Coercions: Implicit conversion from one type to another by the compiler

```
E -> num       { E.type := integer }
E -> num . num { E.type := real }
E -> id        { E.type = lookup(id.entry) }
E -> E1 op E2  { E.type :=
   if E1.type = integer and E2.type = integer then
      integer
   else if E1.type = real and E2.type = integer then
      real
   else if E1.type = integer and E2.type = real then
      real
   else if E1.type = real and E2.type = real then
      real
   else
      type_error }
```

Overloading

An overloaded symbol has different meanings depending on its context

- E.g. +: integer addition, real addition, complex addition, string concatenation, …
- Overloading is said to be resolved if a unique meaning is determined.

Set of Possible Types

It is not always possible to resolve overloading immediately.

- E.g.)

```
function "*" (i, j: integer) return integer;
function "*" (i, j: integer) return complex;
function "*" (i, j: complex) return complex;
```

- 3*5 : either integer or complex
- 2*(3*5) : 3*5 must be integer because * takes same arg. types
- z*(3*5) : 3*5 must be complex if z is a complex type.

```
E -> E1 { E.types := E1.types }
E -> id { E.types := lookup(id.entry) }
E -> E1(E2) { E.types :=
    { t | s->t ∊ E1.types and s ∊ E2.types }}
```

Narrowing the Set of Possible Types

Two depth-first traversals

- Synthesize types during the first path
- Update type during the second path

```
E -> E1       E.types := E1.types
              E.type  := if E1.types = {t}
                         then t else type_error

E -> id       E.types := lookup(id.entry)

E -> E1(E2)   E.types :=
                   {r|s in E2.types and s->r in E1.types}
              t := E.type
              S := {s|s in E2.types and s->t in E1.types}
              E2.type := if S={s}
                         then s else type_error
              E1.type := if S={s}
                         then s->t else type_error
```

Polymorphic Functions

Polymorphic functions

- The statements in the body can be executed with arguments of different types.

Type Variables

- Variables representing type expressions.
- Allow us to talk about unknown types.
- Checking consistent usage of identifiers that don't need to be declared.

Polymorphic Functions

Type Inference

- Problem of determining the type of a language construct from the way it is used.
- E.g.
```
function deref(p)
  begin
    return p^
  end;
```
  - From `deref(p)`, assume that the type of `p` is β
  - From `p^`, infer that the type of `p` is `β = pointer(α)`
  - The type of `deref` is `pointer(α) -> α`

Language with Polymorphic Functions

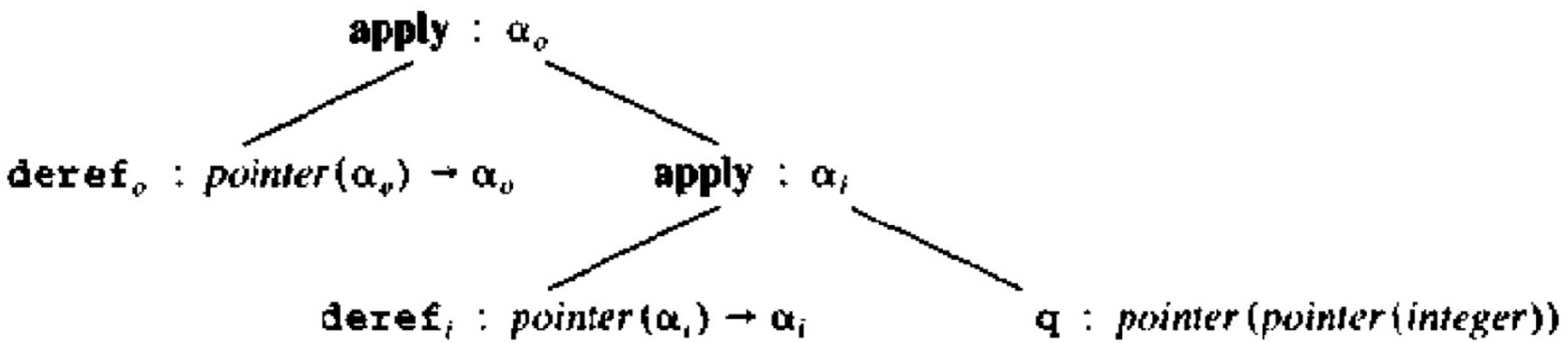
Grammar for language with polymorphic functions

```
P -> D ; E
D -> D ; D | id : Q
Q -> ∀ type_var . Q | T
T -> T '->' T
   | T * T
   | unary ( T )
   | basic_type
   | type_var
   | ( T )
E -> E ( E ) | E, E | id
```

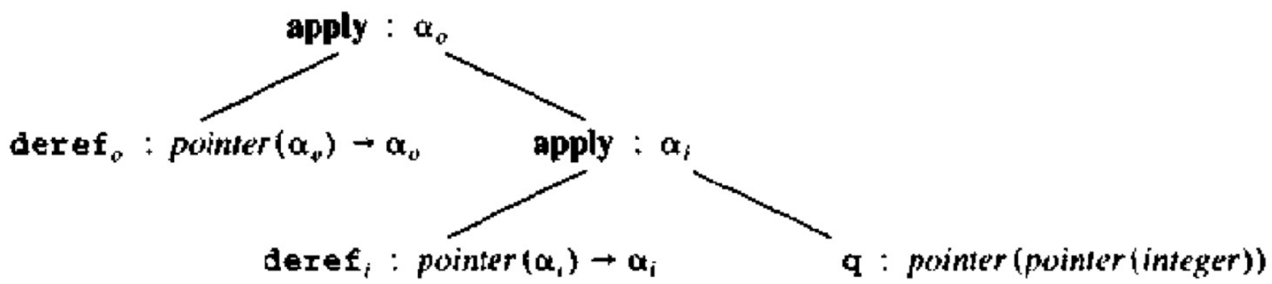
Language with Polymorphic Functions

Example Program

```
deref: ∀α.pointer(α) -> α;
q    : pointer(pointer(integer));
deref(deref(q))
```



Type-Checking Polymorphic Functions



Distinct occurrences of a polymorphic function may have different types

- deref$_o$ and deref$_i$ have different types

Equivalence of type need to be updated

- Unification: make s and t structurally equivalent by replacing the type variables in s and t by type expressions

Record the effect of unifying two expressions

- If after unification a type variable α represents a type t then it should keep represent t through out type-checking.

Substitution

Substitution:

- `S(α)`: mapping from type variables to type expressions
- `S(t)`: consistent replacement of type variables with their mapped type expressions (= `subst(t)`)

```
function subst(t:type_expr): type_expr
begin
    if t is a basic type then return t
    if t is a variable then return S(t)
    if t is t1->t2 then
            return subst(t1)->subst(t2)
end
```

Instance

- `S` is the substitution function, `S(t)` is an instance of `t`.
- We write `s < t` to indicate that `s` is an instance of `t`
  - `pointer(integer) < pointer(α)`
  - `integer -> integer < α -> α`
  - `pointer(α) < β`
  - `α < β`

  - `integer ? real`
  - `integer -> real ? α -> α`
  - `integer -> α ? α -> α (all occurrences of α must be replaced)`

Unification

Two type expressions `t1` and `t2` **unify** if there is a substitution `S` such that
`S(t1) = S(t2)`

A substitution is the **most general unifier** if

- `S(t1) = S(t2)`
- for any other unifier `S'`, `S'` is an instance of `S` (for any `t, S'(t) < S(t)`)

Checking Polymorphic Functions

```
E -> E1 ( E2 ) {
    p := mkleaf(newtypevar)
    unify(E1.type, mknode(->, E2.type, p))
    E.type := p
}
E -> E1, E2 {
    E.type := mknode(*, E1.type, E2.type)
}
E -> id {
    E.type := fresh(id.type)
}
```

Type Checking rule for `E -> E1(E2)`

- If `E1.type = α` and `E2.type = β`,
  - then `E.type = β -> γ`

Unification Algorithm

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```

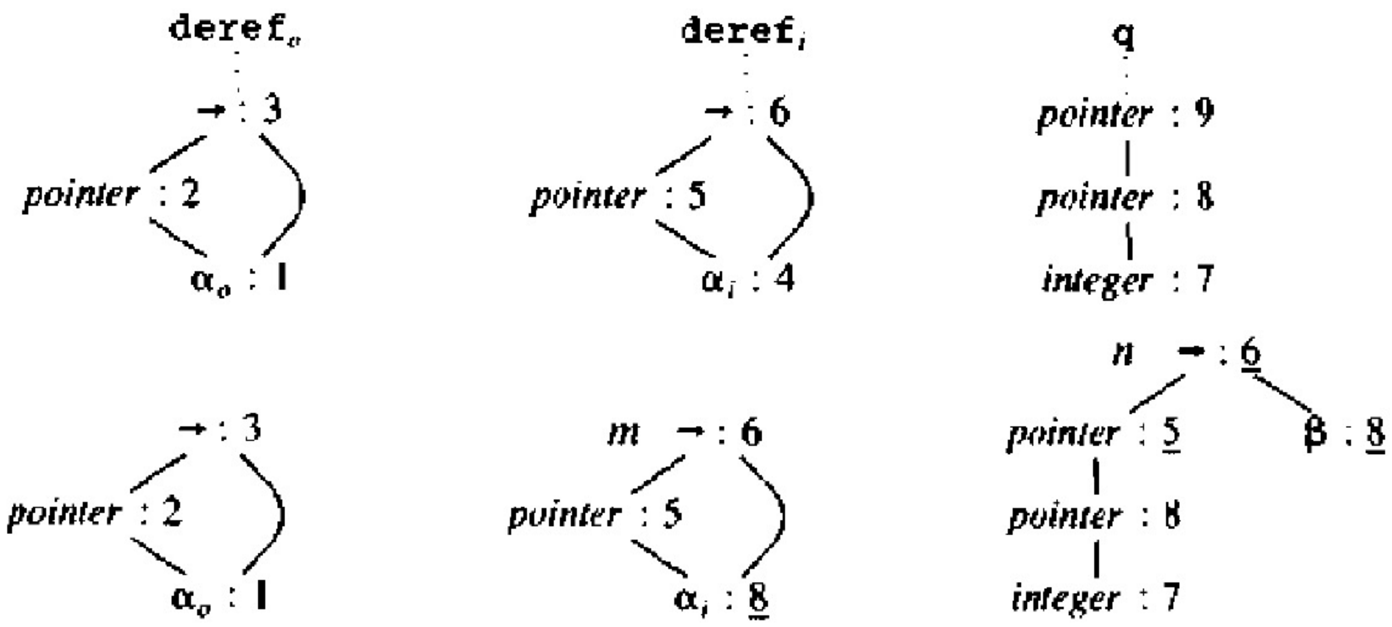
find(n) returns the representative of n

union prefers to make non-variable node a representative node

Checking Polymorphic Functions

E.g.

```
deref: ∀α.pointer(α) -> α;
q    : pointer(pointer(integer));
deref(deref(q))
```



Unification

Exercise

- Draw a dag for the following type expressions
- Unify the following two type expressions
  - `((α1->α2)*list(α3))->list(α2)`
  - `((α3->α4)*list(α3))->α5`
- Check the structural equivalence of the following two type expressions
  - `e: real->e`
  - `f: real->(real->f)`

Questions?