Types

Type: A set of values and meaningful operations on them.

Types provide semantic “sanity checks” (consistency checks). Types help identify:

- errors, if an operator is applied to an incompatible operand.
  - dereferencing of a non-pointer
  - adding a function name to something
  - incorrect number of parameters to a procedure
  - ...
- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- identification of polymorphic functions. Such functions can be executed with arguments of several types (e.g: list of objects of unknown type α)

Type systems

Each language construct (operator, expression, statement, …) has a type

Basic types: integer, real, character, …

Constructed types: arrays, records, sets, pointers, functions

A type system is a collection of rules for assigning type expressions to operators, expressions, … in the program. Type systems are language dependent.

A type checker implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker “computes” or “reconstructs” type expressions.

Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).
  Rule for + (analogue rules for − and *):

  \[ E \vdash e_1 : \text{integer} \quad E \vdash e_2 : \text{integer} \]

  \[ E \vdash (e_1 + e_2) : \text{integer} \]

  where \( E \) is a type environment that maps constants and variables to their types.

  In combination with the following two axioms in the type system \{ c : α \} \vdash c : α and \{ v : α \} \vdash v : α we can now infer, that \((2 + 3)\) is of type integer:

  \[ E \vdash 2 : \text{integer} \quad E \vdash 3 : \text{integer} \]

  \[ E \vdash (2 + 3) : \text{integer} \]

  where \( E = \{ 2: \text{integer}, 3: \text{integer} \} \).

  In general, type deduction proofs work bottom up.

Example type rules

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type “foo”, then the type of the result is a “pointer to foo”. (C and C++ definition)

  \[ E \vdash e : \alpha \]

  \[ E \vdash &e : \text{pointer}(\alpha) \]

- Two expressions can only be compared if they have the same types. The result is of type boolean.

  \[ E \vdash e_1 : \alpha \quad E \vdash e_2 : \alpha \]

  \[ E \vdash (e_1 - e_2) : \text{boolean} \]

  In the examples, integer and α are type expressions.
1. A basic type is a type expression. A special basic type, `type Error` will signal an error. A basic type `void` denotes an untyped statement.

2. Since type expressions may be named, a type name is a type expression.

3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).

4. A **type constructor** applied to type expressions is a type expression. Examples:

   - arrays
   - cartesian products
   - records
   - pointers
   - functions

**Type constructors** (cont.)

- **Records**
  The difference between a record and a product is that the fields of a record have names. The record type constructor will be applied to a tuple formed from field names and field types. *e.g.:

    ```
    type row = record
    address: integer
    lexeme: array [1..15] of char
    end
    var table: array[1..101] of row
    ```

declarates the type name `row` representing the type expression:

    ```
    record((address × integer) ×
    (lexeme × array(1..15, char)))
    ```

and the variable `table` to be an array of records of this type

**Type constructors** (cont.)

- **Pointers**
  If `T` is a type expression, then `pointer(T)` is a type expression denoting the type "pointer to an object of type `T`.

- **Functions**
  Functions map elements of one set, the domain, into another set, the range.

  *E.g.,* Pascal’s `mod` maps a pair of integers, `int × int` into an integer, type `int`

  ```
  int × int → int
  ```

Note that type constructors are recursive

Can construct types such as:

1. pointer to pointer to integer
2. pointer to array of integer
3. array of pointer to integer
4. array of record of pointer to integer
Type constructors (cont.)

A convenient way to represent a type expression is to use a graph.

Example:

```
    X  integer
   /     \
char   char
```

$(char \times char) \rightarrow integer$

Type checking

The purpose of type checking is to prevent **type errors**. Type errors can occur during the execution of a program, for instance, when a function is applied with an argument of the wrong type or if a variable with a non-function type is called.

“How much” type checking is done is part of the definition of the programming language.

Type checking can be performed

- at compile-time (only)
- at compile-time and program execution time, or
- at program execution time (only).

Type checking — discussion

**compile-time:**

+ points out type errors early
+ no overhead at program execution time (run-time)
  - cannot always be done
+ parts of the program that may never be executed are still checked (too restrictive)

**program execution time:**

+ allows more flexibility (e.g.: type may change depending on the use of a variable)
  - overhead in terms of space and time
+ part of program that are not executed are not checked
  + rapid prototyping
  - harder to debug

Goal: Strongly typed languages with as much type checking as possible at compile-time
A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source 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 ↑

* Basic types char, integer, typeError

* assume all arrays start at 1, e.g.,
  array [256] of char
  results in the type expression array(1..256,char)

* ↑ builds a pointer type, so ↑ integer
  results in the type expression pointer(integer)

A simple type checker (cont.)

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 typeError }
E ::= E₁[E₂] { E.type ← if E₂.type ← integer and
E₁.type ← array(s,t) then t
else typeError }
E ::= E₁ ↑ { E.type ← if E₁.type ← pointer(t) then t
else typeError }

Is our example language strongly typed?

Type checking statements

Statements do not typically have values, therefore we assign them the type void. If an error is detected within the statement, it gets type typeError.

S ::= id ← E { S.type ← if id.type ← E.type
then void
else typeError }
S ::= if E then S₁ { S.type ← if E.type ← boolean
then S₁.type
else typeError }
S ::= while E do S₁ { S.type ← if E.type ← boolean
then S₁.type
else typeError }
S ::= S₁ ; S₂ { S.type ← if S₁.type ← void
and S₂.type ← void then void
else typeError }

A simple type checker (cont.)

(Partial) translation scheme (attribute grammar with evaluation order) for the type system

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 T { array(1..num.val, T₁.type) }

These rules save the type of identifier id in the symbol table.
Type checking functions

We add two new productions to the grammar to represent function declarations and applications:

\[ T ::= \ T \mapsto \ T \quad \text{declaration} \]
\[ E ::= \ E \ ( \ E \ ) \quad \text{application} \]

To capture the argument and return type, we use:

\[ T ::= \ T_1 \mapsto \ T_2 \quad \{ \ T.\text{type} \leftarrow (T_1.\text{type} \to T_2.\text{type}) \ \} \]
\[ E ::= \ E_1 \ ( \ E_2 \ ) \quad \{ \ E.\text{type} \leftarrow \text{if } E_2.\text{type} = s \text{ and } E_1.\text{type} = s \rightarrow t \text{ then } t \text{ else typeError} \ \} \]

Type equivalence

**Declaration equivalence**

Problems arise in languages that use name equivalence but allow declarations to use a type that is not a name. Solution: Implicit new (fresh) type names are created for such types for each declaration.

- \( \text{type link} = \uparrow A \)
- \( \text{var } a_1 : \uparrow A // \text{implicit type name type1} \)
- \( \text{var } a_2, a_3 : \uparrow A // \text{implicit type name type2} \)
- \( \text{var } a_4 : \text{link} \)
- \( \text{var } a_5 : \text{link} \)

- \( a_1 \ (\text{type1}) \) has a different type from \( a_2, a_3 \ (\text{type2}) \).
- \( a_4, a_5 \) are of the same type (link), but have a different type from \( a_1 \) and \( a_2, a_3 \).

Note: There are languages that use a hybrid approach.

Example: C uses structural equivalence for all types except records. Records may contain recursive references. C treats the name of the record as part of its type, and therefore testing for structural equivalence stops when a record constructor is reached.

**Structural type equivalence algorithm**

```plaintext
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 array (t1, t2) then
    return sequiv (s1, t1) and sequiv (s2, t2)
  else if s - s1 x s2 and and t - t1 x 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
```
Next lecture

Symbol table organization

Please read ASU Chapter 7.6