Type Checking

Types

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

Types provide semantic "sanity checks" (consistency checks) and determine efficient implementations for data objects.

Types help identify:
- errors, if an operator is applied to an incompatible operand
  - dereferencing of a non-pointer
  - adding a function 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 + i)
- identification of polymorphic functions

Type Systems

Type system: Each language construct (operator, expression, statement, ...) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for:
- basic types
  - integer, char, real, boolean, typeError
- constructed types
  - array(lb…ub, T) // array of T
  - pointer(T) // pointer to T
  - T1 X T2 // tuple of T1, T2
  - T1 -> T2 // function w/ arg T1 returning T2

Type Checker

- A type checker implements a type system. It computes or "constructs" type expressions for each language construct.

  Static type checking
  - Detects type errors at compile time
  - No run time overhead
  - Not always possible (e.g., A[1])

  Dynamic type checking
  - Performed at run time
  - More flexible, allows prototyping
  - Run-time overhead to maintain & check tags

Type Inference Rules

- Specifies the type of an expression

  Example
  - If operands of addition are of type integer, result is of type integer
  - Result of unary & operator is pointer to type of operand

Denotational semantics of type inference rule

\[ E \left( e_1 ; e_2 \right); \text{integer} = E \left( e_1 + e_2 \right); \text{integer} \]

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

Question
- How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

\[ 3.0 + 1 \text{ or } 1 + 3.0 \]
Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

```plaintext
type A is array(1..10) of integer;

type B is array(1..10) of integer;

a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence: (a, b, c, d, e)
name equivalence: (a), (b), (c, d, e)

Syntax Directed Translation Scheme (in CUP)

Revisit our type inference rule for "*".

```plaintext
exp ::= exp.e1 PLUS exp.e2
      {
        if (e1 == sym.INT && e2 == sym.INT)
          RESULT = sym.INT;
        else {
          RESULT = typeError;
          System.out.println("Error: illegal operand types");
        }
      }
```

• The definition of type expression as Java types (static final int fields in class sym) should be done in mycc.cup.
• The assignment of type expression Java types to terminals and nonterminals of the grammar is done in mycc.cup.

Syntax Directed Translation Scheme (in Yacc)

Revisit our type inference rule for "*".

```plaintext
exp :  exp \ '+' \ exp \ {  \ if ($1 == integer \ && \ $3 == integer)
                              $$ = integer;
                      \ else {  
                              $$ = typeError;
                              printf("Error: illegal operand types
");  
                      }  
                }
```

• The definition of type expression as C types (structs) should be done in attr.h, attr.c may contain helper functions.
• The assignment of type expression C types to terminals and nonterminals of the grammar is done in parse.y.

Type Checker Example

Grammar for source language:

```plaintext
P ::= D : E
D ::= D : E | id : T
T ::= char | integer | array [num] of T | T
E ::= literal | num | id | E mod E | E[E] | E |
T ::= T -> T declaration
E ::= E (E ) application
```

• 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)

Type Checker Example (cont.)

• Handling declarations

```plaintext
D ::= id : T
T ::= char
T ::= integer
T ::= T | T1
T ::= array [num] of T1
```

Type Checker Example (cont.)

• Handling expressions

```plaintext
E ::= literal
E ::= num
E ::= id
E ::= E1 mod E2
E ::= E1 [E2]
E ::= E1 |
```

CS430
Type Checker Example (cont.)

- Handling statements
  - Handling statements
    - S := id → E
      - if id.type = E.type then
        - void
      - else typeError
    - S := if E then S₁
      - if E.type = boolean
        - then S₁.type
      - else typeError
    - S := while E do S₁
      - if E.type = boolean
        - then S₁.type
      - else typeError
    - S := S₁ : S₂
      - if S₁.type = void
        - then void
      - else typeError

Symbol Tables

- Symbol table
  - Compile-time structures for resolving references to names
  - Will look at run-time structures later
  - Can also associate attributes with name

- Attributes possibly associated with name
  - Type
  - Declaring procedure
  - Lexical level
  - If array, number and size of dimensions
  - If function, number and type of parameters

Lexically-scoped Symbol Tables

- The problem
  - The compiler needs a distinct record for each declaration
  - Nested lexical scopes admit duplicate declarations

- The interface
  - insert(name, level) - creates record for name at level
  - lookup(name, level) - returns pointer or index
  - delete(level) - removes all names declared at level

Many implementation schemes have been proposed (see § B.4)

- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Example

procedure p {
  int a, b, c,
  procedure q {
    int v, x, w
    procedure r {
      int x, y, z
    }...}
  }...} - q...

Picturing it as a series of Algol-like procedures
Lexically-scoped Symbol Tables

High-level idea
- Create a new table for each scope
- Chain them together for lookup

"Chain of tables" implementation
- `insert()` may need to create table
- It always inserts at current level
- `lookup()` walks chain of tables & returns first occurrence of name
- `delete()` throws away table for level `p`, if it is top table in the chain

Individual tables can be hash tables.

Implementing Lexically Scoped Symbol Tables

Stack organization

<table>
<thead>
<tr>
<th>nextFree</th>
<th>a (level 2)</th>
<th>b (level 1)</th>
<th>c (level 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>y</code></td>
<td><code>z</code></td>
<td><code>w</code></td>
</tr>
</tbody>
</table>

Implementation
- `insert()` creates new level pointer if needed and inserts at `nextFree`
- `lookup()` searches linearly from `nextFree` forward
- `delete()` sets `nextFree` to the equal the start location of the level deleted.

Advantage
- Uses much less space
- Lookups can be expensive

Disadvantage
- Lookups can be expensive

Threaded stack organization

<table>
<thead>
<tr>
<th>nextFree</th>
<th>a (level 2)</th>
<th>b (level 1)</th>
<th>c (level 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>y</code></td>
<td><code>z</code></td>
<td><code>w</code></td>
</tr>
</tbody>
</table>

Implementation
- `insert()` puts new entry at the head of the list for the name
- `lookup()` goes direct to location
- `delete()` processes each element in level being deleted to remove from head of list

Advantage
- Lookup is fast
- `delete()` takes time proportional to number of declared variables in level

Disadvantage
- Lookup is fast
- `delete()` takes time proportional to number of declared variables in level