Type Checking

Roadmap (Where are we?)

Last lecture
- Context-sensitive analysis
  - Motivation
  - Attribute grammars
  - Ad hoc Syntax-directed translation

This lecture
- Type checking
  - Type systems
  - Using syntax directed translation
- Symbol tables
  - Lexical scoping
  - Implementation
    - Stack
    - Threaded stack
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 + 1)
- 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(1b..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 System Properties

• Progress
  → Given expression e, either
    • e is a value with some type T
    • e -> e’ (e can be evaluated to produce expression e’)
  → I.e., prevent expressions such as (2 3) that cannot be evaluated

• Preservation
  → Given expression e with type T
    • If e -> e’ then e’ must also have type T
  → I.e., prevents type of expression e from changing at run time

• Soundness
  → If a type system supports progress & preservation, then
    • Given expression e, e ->* v (for some value v) or e diverges
  → I.e., well-typed programs don’t go wrong

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[i])

• Dynamic type checking
  → Performed at run time
  → More flexible, allows prototyping
  → Run-time overhead to maintain & check tags
Type Inference (for Expressions)

- 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

\[
\begin{align*}
E & \quad e_1 : \text{integer} \quad E \quad e_2 : \text{integer} \\
\hline
E \quad (e_1 + e_2) : \text{integer}
\end{align*}
\]

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?

\[
3.0 + 1 \quad \text{or} \quad 1 + 3.0
\]

Type Inference (in General)

- Goal
  - Given expression with no type annotations, reconstruct valid type for the expression, or determine there is no valid typing
- Approach
  - Use type variables for unknown types
  - Generate equality constraints among types and type variables
  - Solve constraints to determine valid typing (unification)
  - May require general Constraint Logic Programming (CLP)
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 "+".

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

- The definition of type expression as C types (structs) should be done in `attr.h` and `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:

```
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

\[
D := \text{id: T} \quad \{ \text{addtype(id.entry, T.type)} \} \\
T := \text{char} \quad \{ \text{T.type} \leftarrow \text{char} \} \\
T := \text{integer} \quad \{ \text{T.type} \leftarrow \text{integer} \} \\
T := \top T_1 \quad \{ \text{T.type} \leftarrow \text{pointer}(T_1.type) \} \\
T := \text{array [num] of } T_1 \quad \{ \text{T.type} \leftarrow \text{array(1..num.val, T_1.type)} \}
\]

Type Checker Example (cont.)

- Handling expressions

\[
E := \text{literal} \quad \{ \text{E.type} \leftarrow \text{char} \} \\
E := \text{num} \quad \{ \text{E.type} \leftarrow \text{integer} \} \\
E := \text{id} \quad \{ \text{E.type} \leftarrow \text{lookup(id.entry)} \} \\
E := E_1 \mod E_2 \quad \{ \text{E.type} \leftarrow \text{if } E_1.type = \text{integer and } E_2.type = \text{integer then integer} \} \\
& \text{else typeError} \} \\
E := E_1[E_2] \quad \{ \text{E.type} \leftarrow \text{if } E_2.type = \text{integer and } E_1.type = \text{array(s,t) then t} \} \\
& \text{else typeError} \} \\
E := E_1[\downarrow] \quad \{ \text{E.type} \leftarrow \text{if } E_1.type = \text{pointer} \} \\
& \text{then } t \text{ else typeError} \} \\
\]
Type Checker Example (cont.)

• Handling statements

\[
\begin{align*}
S &::= \text{id} \leftarrow E \
&\quad \begin{array}{l}
S.\text{type} \leftarrow \text{if } E.\text{type} = \text{void} \\
&\quad \text{then void} \\
&\quad \text{else typeError}
\end{array} \\
S &::= \text{if } E \text{ then } S_1 \
&\quad \begin{array}{l}
S.\text{type} \leftarrow \text{if } E.\text{type} = \text{boolean} \\
&\quad \text{then } S_1.\text{type} \\
&\quad \text{else typeError}
\end{array} \\
S &::= \text{while } E \text{ do } S_1 \
&\quad \begin{array}{l}
S.\text{type} \leftarrow \text{if } E.\text{type} = \text{boolean} \\
&\quad \text{then } S_1.\text{type} \\
&\quad \text{else typeError}
\end{array} \\
S &::= S_1 ; S_2 \
&\quad \begin{array}{l}
S.\text{type} \leftarrow \text{if } S_1.\text{type} = \text{void} \\
&\quad \text{then void} \\
&\quad \text{else typeError}
\end{array}
\end{align*}
\]

Type Checker Example (cont.)

• Handling functions

\[
\begin{align*}
T &::= T_1 \rightarrow T_2 \
&\quad T.\text{type} \leftarrow (T_1.\text{type} \rightarrow T_2.\text{type}) \\
E &::= E_1 \left( E_2 \right) \
&\quad E.\text{type} \leftarrow \text{if } E_1.\text{type} = s \rightarrow t \\
&\quad \text{and } E_2.\text{type} = s \text{ then } t \\
&\quad \text{else typeError}
\end{align*}
\]
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, b, x, w
        procedure r {
            int x, y, z
            ...
        }
        procedure s {
            int x, a, v
            ...
        }
        ...
    }
}

B0: {
    int a, b, c
    B1: {
        int v, b, x, w
        B2: {
            int x, y, z
            ...
        }
        B3: {
            int x, a, v
            ...
        }
        ...
    }
}

Picturing it as a series of Algol-like procedures

B0: {
    int a₂₅, b₂₆, c₂₇
    B1: {
        int v₀, b₄₈, x₅₉, w₆₀
        B2: {
            int x₀₅, y₆₆, z₇₇
            ...
        }
        B3: {
            int x₁₀₂, a₁₁₁, v₁₂₂
            ...
        }
        ...
    }
}
Lexically-scoped Symbol Tables

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

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

Individual tables can be hash tables.

Remember
\begin{itemize}
\item \( a_{10}, b_4, c_2, v_{12}, w_6, x_{10} \)
\item no \( y \) or \( z \)
\end{itemize}

If we add the subscripts, the relationship between the code and the table becomes clear.
Implementing Lexically Scoped Symbol Tables

Stack organization

```
nextFree  
v   12
a   11
x   10
w   6
x   5
b   4
v   3
c   2
b   1
a   0
```

Implementation
- **insert()** creates new level pointer if needed and inserts at nextFree
- **lookup()** searches linearly from nextFree-1 forward
- **delete()** sets nextFree to the equal the start location of the level deleted.

Advantage
- Uses much less space

Disadvantage
- Lookups can be expensive

p (level 0)
q (level 1)
s (level 2)

CS430  23
Implementing Lexically Scoped Symbol Tables

Threaded stack organization

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

Disadvantage
- `delete` takes time proportional to number of declared variables in level