CS415 Compilers

Type systems
Symbol tables
Code generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework #6 has been posted.

• Project 2 has been posted.

• May need to postpone my office hours tomorrow. Please see announcements.
**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 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, e.g., one-dimensional arrays:
  \[\text{array}(lb, ub, \text{elem}_{\text{type}})\]
  where elem_type is a type expression

A type checker implements a type system. It computes or “constructs” type expressions for each language construct.
Example type inference rule pointer dereferencing:

\[
\frac{E \vdash e : \text{pointer}(\text{integer})}{E \vdash *e : \text{integer}}
\]

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

\text{pointer}(...) is now part of the type expression language such as \text{array}(...).
Example type inference rule pointer dereferencing:

\[
\frac{E \vdash e : \text{pointer}(\beta)}{E \vdash *e : \beta}
\]

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

Type expressions may also contain type variables such as \(\beta\). Type variables can denote any type expression.

Type variables are needed to express polymorphic types.
Example type inference rule address computation:

\[
\begin{array}{c}
E \vdash e : \text{integer} \\
\hline
E \vdash \&e : \text{pointer(}\text{integer}\text{)}
\end{array}
\]

where $E$ is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?

\[
\begin{array}{c}
E \vdash e : \beta \\
\hline
E \vdash \&e : \text{pointer}(\beta)
\end{array}
\]
Formal proof that a program can be typed correctly.

```c
int a;
...*(&a) + 3 ...
```
Programmers may define their own types and give them names:

```
type my_int is int;
...
int a;
my_int b;
...
... a + b ...
```

Type names can also be part of the type expression language. Note: type names and type variables are different!
Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

    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:
    name equivalence:
Type Equivalence

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

Example:

```
  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);
Revisit our type inference rule for “+”.

```c
exp : exp '+' exp { if ($1 == TYPE_INT && $3 == TYPE_INT)
    $$ = TYPE_INT;
    else {
        $$ = TYPE_INT; // educated “guess”
        printf("\n***Error: illegal operand types\n");
    }
}
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in `attr.h`. `attr.c` may contain helper functions.
Lexically-scoped Symbol Tables

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

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

Many implementation schemes have been proposed (see § B.4)
• We’ll stay at the conceptual level
• Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We’ll see the corresponding run-time structures that are used to establish addressability later.
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
            ...
        }
        ...
    }
    ...
    q ...
}

B0: {
    int a, b, c
    B1: {
        int v, b, x, w
        B2: {
            int x, y, z
            ...
        }
        B3: {
            int x, a, v
            ...
        }
        ...
    }
    ...
}
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_0, b_1, c_2
    ...
}

B1: {
    int v_3, b_4, x_5, w_6
    ...
}

B2: {
    int x_7, y_8, z_9
    ...
}

B3: {
    int x_{10}, a_{11}, v_{12}
    ...
}

Picturing it as a series of Algol-like procedures

a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z
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.
Lexically-scoped Symbol Tables

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

Remember
- If we add the subscripts, the relationship between the code and the table becomes clear.

The names visible in s:
- $a_{11}$, $b_4$, $c_2$, $v_{12}$, $w_6$, $x_{10}$,
- no $y$ or $z$
Implementing Lexically Scoped Symbol Tables

Stack organization

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
Stack organization

\[ a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, \text{no y or z} \]

Implementation

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

Advantage

- Uses much less space

Disadvantage

- Lookups can be expensive
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
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

---

### h(x)

- \(a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, \) no y or z
A compiler is a lot of fast stuff followed by some hard problems

→ The hard stuff is mostly in code generation and optimization
→ For superscalars, its allocation & scheduling that is particularly important
The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - When only 1 name can reference its value (no aliasing)
  - Pointers, parameters, aggregates & arrays all cause trouble

- When should a value be allocated to a register?
  - When it is both safe & profitable

Encoding this knowledge into the IR (register-register model)

- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference

Relies on a strong register allocator
Recursive Treewalk vs. Ad-hoc SDT

int expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case \times, \div, +, -:
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit(loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit(loadI, val(node), none, result);
            break;
    }
    return result;
}

Goal:

Expr: Expr \{ $$ = $1; \} ;

Expr PLUS Term
{ t = NextRegister();
    emit(add, $1, $3, t); $$ = t; }

| Expr MINUS Term {…}
| Term { $$ = $1; } ;

Term: Term TIMES Factor
{ t = NextRegister();
    emit(mult, $1, $3, t); $$ = t; }

| Term DIVIDES Factor {…}
| Factor { $$ = $1; } ;

Factor: NUMBER
{ t = NextRegister();
    emit(loadI, val($1), none, t );
    $$ = t; }

| ID
{ t1 = base($1);
    t2 = offset($1);
    t = NextRegister();
    emit(loadAO, t1, t2, t);
    $$ = t; }
Work on the project!

More on code generation

Intermediate representations
Read EaC: Chapter 5.1 - 5.3