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
Types and 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, e.g., one-dimensional arrays:
  \[
  \text{array}(lb, ub, \text{elem\_type})
  \]
  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:

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

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:

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

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

What about a polymorphic version of this rule?

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

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

```java
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!
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:
name equivalence:
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);
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

§ 5.5 in EaC

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

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
    ...
  }
  ...
  r ... s
}
  ... q ...
}

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

Picturing it as a series of Algol-like procedures

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

procedure q ...

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 $x_{10}$, $a_{11}$, $v_{12}$
        ...
        ...
    }
    ...
}

... $a_{11}$, $b_4$, $c_2$, $v_{12}$, $w_6$, $x_{10}$, 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
• \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.
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

```
nextFree
v
a
x
w
x
b
```

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

 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
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

```c
int expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×, ÷, +, −:
            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:**

```c
Expr { $$ = $1; } ;
```

**Expr:**

```c
PLUS Term
```

```c
{ t = NextRegister();
emit(add,$1,$3,t); $$ = t; }
```

**Expr MINUS Term**

```c
{ ...}
```

**Term:**

```c
TIMES Factor
```

```c
{ t = NextRegister();
emit(mul,$1,$3,t); $$ = t; }
```

**Term DIVIDES Factor**

```c
{ ...}
```

**Factor:**

```c
NUMBER
```

```c
{ t = NextRegister();
emit(loadI,val($1),none,t);
$$ = t; }
```

**ID**

```c
{ t1 = base($1);
  t2 = offset($1);
  t = NextRegister();
  emit(loadAO,t1,t2,t);
  $$ = t; }
```
Handling Assignment (just another operator)

\[ \text{lhs} \leftarrow \text{rhs} \]

**Strategy**

- Evaluate \( \text{rhs} \) to a *value* *(an rvalue)*
- Evaluate \( \text{lhs} \) to a *location* *(memory address)* *(an lvalue)*
  - \( \text{lvalue} \) is an address \( \Rightarrow \) store rhs
- If \( \text{rvalue} \) & \( \text{lvalue} \) have different types
  - Evaluate \( \text{rvalue} \) to its “natural” type
  - Convert that value to the type of \( \text{lhs} \) value, if possible

Unambiguous scalars may go into registers (no aliasing)
Ambiguous scalars or aggregates go into memory (possible aliasing)
Handling Assignment

What if the compiler cannot determine the rhs’s type?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a tag field to the data items to hold type information

Code for assignment becomes more complex

evaluate rhs
If lhs.type_tag ≠ rhs.type_tag then
    convert rhs to type(lhs) or
    signal a run-time error
lhs ← rhs

This is much more complex than if it knew the types
Handling Assignment

Compile-time type-checking
- Goal is to eliminate both the runtime check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy
- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)
Handling Assignment (with reference counting)

Garbage Collection

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

\[
\text{evaluate rhs} \\
\text{lhs} \rightarrow \text{count} \leftarrow \text{lhs} \rightarrow \text{count} - 1 \\
\text{lhs} \leftarrow \text{addr(rhs)} \\
\text{rhs} \rightarrow \text{count} \leftarrow \text{rhs} \rightarrow \text{count} + 1
\]

This adds 1 +, 1 -, 2 loads, & 2 stores

With extra functional units & large caches, this may become either cheap or free. What about power consumption?
First, must agree on a storage scheme

**Row-major order** (most languages)
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest

**Column-major order** (Fortran)
- Lay out as a sequence of columns
- Leftmost subscript varies fastest

**Indirection vectors** (Java)
- Vector of pointers to pointers to ... to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis
Laying Out Arrays

The Concept

<table>
<thead>
<tr>
<th>A</th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Row-major order

| A | 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 |

Column-major order

| A | 1,1 | 2,1 | 1,2 | 2,2 | 1,3 | 2,3 | 1,4 | 2,4 |

Indirection vectors

| A | 1,1 | 1,2 | 1,3 | 1,4 |
|   | 2,1 | 2,2 | 2,3 | 2,4 |

These have distinct & different cache behavior
Computing an Array Address

Declaration: \( A[\text{low} .. \text{high}] \) of ...

\[
A[i]
\]

- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: base(A) + (i - low) \times \text{sizeof}(A[1])
Computing an Array Address

Declaration: \( A[\text{low .. high}] \) of ...

\[ A[ i ] \]
- \( \@A + ( i - \text{low} ) \times \text{sizeof}(A[1]) \)
- In general: base\((A) + ( i - \text{low} ) \times \text{sizeof}(A[1]) \)

\( \text{int A[1:10]} \Rightarrow \text{low is 1} \)
Make low 0 for faster access (saves a - )

Almost always a power of 2, known at compile-time \( \Rightarrow \) use a shift for speed
Computing an Array Address

Declaration: \( A[\text{low1 .. high1, low2 .. high2}] \) of ...

\( A[i] \)
- \( \@A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: base(A) + (i - low) \times \text{sizeof}(A[1])

What about \( A[i_1, i_2] \)?

Row-major order, two dimensions
\( @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1]) \)

Column-major order, two dimensions
\( @A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1]) \)

Indirection vectors, two dimensions
\( *(A[i_1])[i_2] \) — where \( A[i_1] \) is, itself, a 1-d array reference

This stuff looks expensive!
Lots of implicit +, -, \times ops
where $w = \text{sizeof}(A[1,1])$

In row-major order

$$@A + (i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w$$

Which can be factored into

$$@A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w$$

$$- (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w)$$

If $\text{low}_i$, $\text{high}_i$, and $w$ are known, the last term is a constant

Define $@A_0$ as

$$@A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w + \text{low}_2 \times w$$

And $\text{len}_2$ as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times \text{len}_2 + j) \times w$$

Compile-time constants
Control Flow

One possible approach for code generation:

Loops
• Evaluate condition before loop (if needed)
• Evaluate condition after loop
• Branch back to the top (if needed)
Merges test with last block of loop body

while, for, do, & until all fit this basic model
for (i = 1; i < 100; i++) {
    body
}

next statement

---

Load I: 1 ⇒ r₁
loadI 1 ⇒ r₂
loadI 100 ⇒ r₃

cmp_GE r₁, r₃ ⇒ r₄

cbr r₄ ⇒ L₂, L₁

L₁: body

add r₁, r₂ ⇒ r₁

cmp_LT r₁, r₃ ⇒ r₅

cbr r₅ ⇒ L₁, L₂

L₂: next statement

---

Initialization

Pre-test

Post-test
Many modern programming languages include a `break`

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

Translates into a jump

- Targets statement outside control-flow construct
- Creates multiple-exit construct
- `Skip` in loop goes to next iteration
Case Statements
1 Evaluate the controlling expression
2 Branch to the selected case
3 Execute the code for that case
4 Branch to the statement after the case
Parts 1, 3, & 4 are well understood, part 2 is the key
Control Flow

Case Statements

1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case (use `break`)

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies

• Linear search (nested if-then-else constructs)
• Build a table of case expressions & binary search it
• Directly compute an address (requires dense case set: jump table)

Surprisingly many compilers do this for all cases!
Compiler Optimization

- tries to improve quality of code (may fail in some cases)
- optimizer typically consists of multiple passes
- different optimization (code improvement) objectives:
  - execution time reduction
  - reduction in resource requirements (memory, registers)
  - (peak) power and energy reduction

- criteria for effectiveness of optimizations
  - safety - program semantics must be preserved
  - opportunity - how often can it be applied?
  - profitability - how much improvement?
We will focus on two optimizations:

1. Common subexpression elimination (CSE - local, ILOC level)

2. Vectorization / parallelization (source level) - will do this later

Local CSE reference: ALSU, chapter 8.5.2
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \(a(i)\) (1-based indexing)

\[
\begin{align*}
4. & \quad t_1 = \text{addr}(a) - 4 \\
5. & \quad t_2 = i \times 4 \\
6. & \quad t_3 = t_1[t_2] \\
& \quad \ldots
\end{align*}
\]
Optimization: Local Common Subexpression Elimination (CSE)

Source code: \( a(i) \times a(i) \) (1-based indexing)

\[
\begin{align*}
4. & \quad t1 = \text{addr}(a) - 4 \\
5. & \quad t2 = i \times 4 \\
6. & \quad t3 = t1[t2] \\
7. & \quad t4 = \text{addr}(a) - 4 \\
8. & \quad t5 = i \times 4 \\
9. & \quad t6 = t4[t5] \\
10. & \quad t7 = t3 \times t6
\end{align*}
\]
Optimization: Local Common Subexpression Elimination (CSE)

Source code: \( a(i) \times a(i) \) (1-based indexing)

Basic Block DAG Construction

code generated:

\[
\begin{align*}
    t1 &= \text{addr}[a] - 4 \\
    t2 &= i \times 4 \\
    t3 &= t1[t2] \\
    t4 &= \text{addr}[a] - 4 \\
    t5 &= i \times 4 \\
    t6 &= t4[t5] \\
    t7 &= t3 \times t6
\end{align*}
\]
How to add a subexpression into a partially constructed DAG:

\[ A = B + C \]

Is there a node already for \( B + C \)?

- If so, add \( A \) to its list of labels.
- If not:
  - is there a node labeled \( B \) already?
    - If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already?
    - If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for +, with left child \( B \) and right child \( C \).

How to do this? \texttt{HASHING \langle op, node(opd1), node(opd2)\rangle}
DAG Construction Algorithm

How to add a subexpression into a partially constructed DAG:

\[ A = B + C \]

Is there a node already for \( B + C \)? \( <+, \text{node}(B), \text{node}(C)> \) defined?

- If so, add \( A \) to its list of labels.
- If not:
  - is there a node labeled \( B \) already? \( \text{node}(B) \) defined?
    If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already? \( \text{node}(C) \) defined?
    If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for +, with left child \( B \) and right child \( C \).

Create \( \text{node}(+) \) with children \( \text{node}(B), \text{node}(C) \)

How to do this? HASHING \( <\text{op}, \text{node}(\text{opd}_1), \text{node}(\text{opd}_2)> \)
DAG Construction Algorithm

Summary:
- every expression is assigned a value number
  examples: node(a),
  node(4),
  node(<+, ValNum1, ValNum2>)
- assignment changes value number associated with LHS variable
- implementation of value numbers
  • use pointers of nodes in DAG
  • use virtual register numbers (code shape encoding!)

You could do this in a single pass in our compiler!
Things to do and next class

Work on the project!

Intermediate representations
Read EaC: Chapter 5.1 - 5.3