Type Checking

- Type Systems
- Type Equivalence
- Typing Expressions
  - Coercion
  - Overloading
  - Error Recovery
- Typing Statements
- Polymorphic Types

Type Checking Situations

- Expression typing
  - Operands matching
  - Selecting operand
- Coercing types
- Selecting among overload possibilities
- Polymorphic type expansion

Type Systems - Rules

- Rules of a language
  - Definition:
    - What are the base/immutable types?
    - What type constructors are available?
    - Can types be named?
  - Resolution:
    - What operators can be applied to what types?
    - What forms of coercion are allowed?
    - How are overloading situations resolved?

Type Systems – Base types

- Base/immutable types
  - Generally objects with a direct machine representation, where the objects can not be further divided
  - Examples: bool, char, int, float, double, long int, unsigned int, …
  - Simple objects have direct types
    - Literals: 3 (int), -5.0 (double)
    - Variables: int x; (int) double y; (double)

Type Systems – Pointer Constructor

- Constructors
  - Pointer
    - In C++ *type* **name*, result is a pointer to a type (ptr to *type*)
    - Examples:
      - int *x; (x is a ptr to an int)
      - float **y; (y is a ptr to a ptr to a float)
    - Operators related to pointer:
      - *****x – dereference x (go to what x points at), in terms of types, * applied to *ptr to y* , results in y
      - x**->** equivalent to **(*x).** – compose * and . operations
      - x**[...]** – pointers can be used as standins for arrays (array ref is just an address, pointer operation)

Type Systems – Array Constructor

- Constructors
  - Arrays
    - In C++ *type name[size]*, result is array (0..*size*-1) of *type*
    - Examples:
      - int z[10]; array (0..9) of int
      - float *w[5]; array (0..4) of pointer to float
      - double t[10][9] is array (0..9) of array (0..8) of double
    - Operators related to pointer:
      - x**[expr]** – refers to element of an array – equivalent to *(x + sizeof(ArrayEl) * **expr**) – why arrays start at 0 in C/C++
        - If x of type array (lo..hi) of *type* x[…] returns *type*

Type Systems – Products

- Constructors
  - Products
    - Certain operations result in products
      - Function parameter lists
      - Function argument lists
      - Class/structure fields
    - Type is the product composition of the individual types
    - Examples:
      - (int x, float y, char z) – int X float X char
      - (3,'X',2.0,0) – int X char X double X int
      - class x { int y; float z; }; - int X float
      - Parameters, class fields are named

Type Systems – Class/Struct/Record Constructor

- Constructors
  - Structures – classes, records, etc.
    - Types are products with named fields (can refer to indivdual members by giving field name)
    - Type is the product of the field types with names attached
    - Example:
      - class t { char x; int y; float z;}; - type is char(x) X int(y) X float(z) with names attached
    - Operators:
      - . operator (x.y) – if x is of type … X *typ* (y) X …, resulting type is *typ*
      - -> operator – composition of * and . operator x->y is (*x).

Type Systems – Function Constructor

- Constructors
  - Functions
    - Types are left hand side of products, followed by ->, followed by result type
    - Example:
      - int foo (float x, char y) { } – float X char → int
    - Operator:
      - *fname*(*args*) – function call, if *args* match left hand size of type associated with *fname*, resulting type is the right hand side of type associated with *fname*
      - example: foo(3.0,'X') has result type of int (from above)

Type Systems – Type Variables

- Type variables
  - Some languages allow the definition of type variables (often useful in dealing with cyclic/recursively defined types)
  - Type variable names often associated with constructed types (e.g., class names)
  - But allowing type names can introduce some equivalence problems (more later)

Forms of Checking

- Static type checking – type checking done at compile type
  - Used in many strongly typed languages (where all variables/objects must have types)
- Dynamic type checking – done at runtime
  - Often used in languages where objects not strongly type (e.g., Lisp)
  - Type checking must be done at runtime (since objects not guaranteed to have a single type)

Type Equivalence

- Name equivalence – objects are considered to be equivalent if they have the same (or in the case of operators – appropriate names)
  - Problem – if a type name is given to a type (Number declared a synonym for int) this may introduce type errors that are not real errors
- Structural equivalence – objects are considered equivalent if they have similar structures
  - Useful but can allow some mappings we may not want:
    - class x { int y; float z;};
    - class position { int angle, float distance; };
    - x and position would be considered to be structurally equivalent

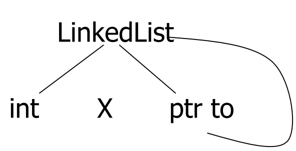
Cyclic Types

- Many structures in programming languages are declared recursively (linked lists, trees, etc.)
- Example:

```
class LinkedList {
  int data;
  LinkedList *next;
};
```

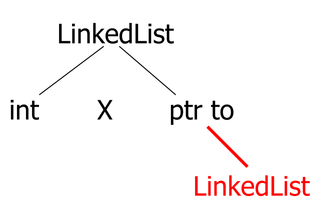
- next field's type is based on the type it is part of

Cyclic Type Graphs



But how to compare this type (structurally or by name)?

Often use names for types to make graphs easier (makes equivalence easier to determine)



Sample Language

- Expressions
  - *intLit* (e.g., 1, 3, -5)
  - *boolLit* (e.g., 0, 1)
  - *varname*
  - $e_1 + e_2$
  - e_1 / e_2
  - $e_1 == e_2$
  - $e_1 < e_2$
  - not e
  - e_1 and e_2
  - *f(arglist)*
  - *v.field*
  - ** e*
- Statements

if (e) $stmt_1$; else $stmt_2$; fi;

ident = *expr*;

f(*arglist*);

- Simple types
  - bool, int, void
- Constructors
  - Products
  - Structures
  - Pointers

Possible Nodes

- IntLitNode (ival)
- BoolLitNode (bval)
- VariableNode (name)
- BinaryNode (op,leftarg,rightarg)
- UnaryNode (op,arg)
- RecFieldNode (recexpr,fname)
- FuncCallNode (fname,arglist)
- IfNode(bexpr,ifstmt,elsestmt)
- Other:
  - ArgListNode(arg,next)
  - StmtListNode(stmt,next)

When to Type Check

- Depending on language, type checks can often be done in parsing
- Can also be done as a separate process
- If done as a separate process, performed as a traversal of the AST(s) from the program
- Generally two routines:
  - Type of expressions
  - Type of statements

Typing Expressions

- Deals with expressions, possibly complex expressions where we assume the expressions will result in type
- Some simple:

Type IntLitNode::expr_type() { return int; }

Type BoolLitNode::expr_type() { return int; }

Type VariableNode::expr_type() {
    return *type of variable*; }

Typing Expressions - Operations

- Type arguments, then check/compare results

```
Type BinaryNode::expr_type() {
 lefttype = leftarg->expr_type();
 righttype = rightarg->expr_type();
 if (op is + or /) {
   rettype = int;
   if (lefttype != int) {
     rettype = error;
     if (lefttype != error)
       report error;
   }
   if (righttype != int) {
     rettype = error;
     if (righttype != error)
       report error;
   }
 }
else if (op is == or <) {
  if ((lefttype == righttype) &&
       (lefttype is int or bool)) {
     rettype = bool;
   }
   else {
     rettype = error;
     if ((lefttype != error) &&
         (righttype != error))
       report error;
   }
  }
  else if (op is and)
   /* check both args are bool, if so, return
     bool */
  return rettype;
}
```

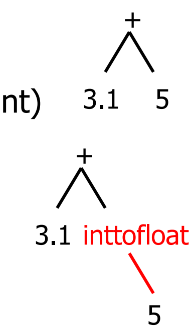
Typing Expressions

- What is missing?
  - What about pointers?
    - Probably should at least check for ==, and maybe for <, + is a more interesting question (and && and / seem unlikely)
  - Can we compare records?
    - Compare field by field?
    - Compare all of memory?
    - Structural or name equivalence (note, looking up names of objects corresponding to records should give us a product type)

Coercion

- It is reasonable (and in most cases desirable) to allow some automatic coercion (ints to floats for an addition)
- When do we do it? - During expression type checking

```
if (op is +) {
  if (lefttype is float) and (righttype is int)
    insert a inttofloat in right child
}
```



Typing Expressions

- UnaryNode – similar to binary
- RecFieldNode – expression must return product with field names (check if field name fits)
- FuncCallNode – build up product type from arglist, then check if type is in symbol table for function name
  - ArgListNode – returns type consisting of product of type of current argument together with type from remainder of argument list

Checking Function Call

foo(3,4.5,'A')

```
        FuncCallNode      Look for int X float X char for function named foo
       /        \
    foo         ArgListNode      Type: int X (rest) = int X float X char
                /        \
               3         ArgListNode      Type: float X (rest) = float X char
                         /        \
                       4.5        ArgListNode      Type: char
                                  /        \
                                'A'        null
```

If type int X float X char → float associated with foo, return type float

Overloading

- Allow name for function to be inserted into symbol table multiple times if type for parameter list (product) differs
- When looking up function to be called match argument type to each of the possibilities and pick the one that matches
- Complicating issue – when coercion is allowed matches may not be perfect
  - If two possibilities are close, which one to choose
  - Example:
    - Definitions:
      - foo : int X float -> int
      - foo : float X int -> float
    - Which to choose when matching arguments int X int?

Error Recovery

- As with parsing, often want to find multiple errors
- What to do when one error detected?
  - Generally, return error as type but don't generate further errors
  - E.g., left argument of + returns error, still want the right argument to be some type that can be added

Typing Statements

- Statements checking causes expression type checks:

```
type StmtListNode::stmt_type() {
  stmttype = stmt->stmt_type();
  if (stmttype != void) report error;
  nexttype = next->stmt_type();
  if (nexttype != void) report error;
  return void;
}
```

Typing Statements

- Parts of statement often must return type:
  - Example: if condition must return boolean type

```
type IfNode::stmt_type() {
  bexprtype = bexpr->expr_type();
  if (bexprtype != bool) report error;
  ifstmttype = ifstmt->stmt_type();
  if (ifstmttype != void) report error;
  if (elsestmt != 0) {
    elsestmttype = elsestmt->stmt_type();
    if (elsestmttype != void) report error;
  }
  return void;
}
```

Polymorphic Type

- C++ has templates – types that have placeholders so they can be applied to multiple types of objects
  - E.g., LinkedList of any type
- Polymorphic classes inserted into symbol table
  - Type resolved when needed