CSC312 Principles of Programming Languages :
Type & Type System
Extra Notes on Type

Big Endian V.S. Little Endian
Array: row-major
Pointer Array
Polymorphism
Endianness

```
  3 1 2
  0
```

memory address space

“three hundred twelve”: Big-endian.
“two hundred thirteen”: Little-endian.
Array Layout in Memory

int A[2][3];

// A[0][*]: is called one row;
// A[*][0]: is called a column.

Row-major:

Column-major:

Most languages use row-major.
Fortran uses column-major.
Pointer Array

int a[100];
int * b;
int *c[100];

int d[2][3];
int * p = &(d[0][0]);
p = p + 1;

What does p point to?
Polymorphism

Def: A function or operation is *polymorphic* if it can be applied to any one of several related types and achieve the same result.

Important for code reuse and productivity!

<table>
<thead>
<tr>
<th>intPrint(a)</th>
<th>print(a)</th>
<th>Which do you prefer? (As either developer or user.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatPrint(b)</td>
<td>print(b)</td>
<td></td>
</tr>
<tr>
<td>stringPrint(c)</td>
<td>print(c)</td>
<td></td>
</tr>
<tr>
<td>boolPrint(d)</td>
<td>print(d)</td>
<td></td>
</tr>
</tbody>
</table>
Ways to Get Polymorphism

* Operator Overloading
* Inheritance: class hierarchy
* Generic (or called Template)
Operator Overloading

Common in language design

– *Built-in overloading*
  - e.g., “+” for integer and floating point

– *Customizable overloading*
  - e.g., allowed in C++/Java/Python
Operator Overloading Example in C++

class Point
{
    public:
        int x;
        int y;
        void operator=(Point anotherPoint);
};

void Point::operator=(Point p)
{
    (*this).a = p.a;
    (*this).b = p.b;
}

Point A = readpoint();
B = A; // B.x=A.x; B.y=A.y;
Inheritance

Common in Object-Oriented Languages.

Property:

If an operation or function is defined to work on data of a class R, it can work on data of subclasses of R.
Example in your parser

class Block( Statement ):
    def __init__(self, stmts):
        self.stmts = stmts

        def meaning(self, state):
            for s in self.stmts:
                state = s.meaning(state)
            return state

class Assign( Statement ): ...
class WhileStmt( Statement ): ...
class IfStmt( Statement ): ...

if (y>0)
{
    x = 5;
    if (x>y)  x = y;
    while (x>0) {
        x = x -y;
    }
}

What if you forget to define meaning function of Assign?
How to prevent that through language support?

Abstract class or interface.
Abstract Class or Interface

Contract in at least what a subclass must have.

class Statement (object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def meaning(self, state): pass
Generic (or Template)

Called parametric polymorphism

Type binding delayed from code implementation to compile time

Example: a generic sort function in Ada.
// define generic types

generic

    type element is private;

    type list is array(natural range <> ) of element;

    with function ""(a, b : element) return boolean;

// declare a package to put this generic function

package sort_pck is

    procedure sort (in out a : list);

end sort_pck;
// implementation of the package
package sort_pck is
procedure sort (in out a : list) is begin
  for i in a'first .. a'last - 1 loop
    for j in i+1 .. a'last loop
      if a(i) > a(j) then
        declare t : element;
        begin
          t := a(i);
          a(i) := a(j);
          a(j) := t;
        end;
      end if;
    end for;
  end for;
end if;

No specification on what type the operands of “>” and “=” have.
// instantiate the generic function to a specific type
package integer_sort is
    new generic_sort( Integer, "->" );

package float_sort is
    new generic_sort( Float, "->" );

Compiler will generate corresponding two instances of functions
at compile time.
Ch. 6  Type System

6.1  Type System for Clite
6.2  Implicit Type Conversion
6.3  Formalizing the Clite Type System
Motivation: Detecting Type Errors

The detection of type errors, either at compile time or at run time, is called type checking.

- Type errors occur frequently in programs.
- Type errors can’t be prevented/detected by EBNF.
- If undetected, type errors can cause severe run-time errors.
- A type system can identify type errors before they occur.
6.1 Type System for CLite

Static binding

Single function: main

Single scope: no nesting, no globals

Name resolution errors detected at compile time
  – Each declared variable must have a unique identifier
  – Identifier must not be a keyword (syntactically enforced)
  – Each variable referenced must be declared.
Example Clite Program (Fig 6.1)

// compute the factorial of integer n
void main ( ) {
    int n, i, result;
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
Designing a Type System

• A set of rules $V$ in highly-stylized English
  – return true or false
  – based on abstract syntax
    • Note: standards use concrete syntax
  – Mathematically a function:
    $$V: \text{AbstractSyntaxClass} \rightarrow \text{Boolean}$$

• Facilitates static type checking.

• Implementation throws an exception if invalid
Type Rule 6.1

All referenced variables must be declared.

- Type map is a set of ordered pairs
  E.g., {<n, int>, <i, int>, <result, int>}
- Can implement as a hash table
- Function **typing** creates a type map
- Function **typeOf** retrieves the type of a variable:
  ```
  typeOf(id) = type
  ```
The typing Function creates a type map

```java
public static TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap( );
    for (Declaration di : d) {
        map.put (di.v, di.t);
    }
    return map;
}
```
Type Rule 6.2

All declared variables must have unique names.

```java
public static void V (Declarations d) {
    for (int i=0; i<d.size() - 1; i++) {
        Declaration di = d.get(i);
        for (int j=i+1; j<d.size(); j++) {
            Declaration dj = d.get(j);
            check( ! (di.v.equals(dj.v)),
                "duplicate declaration: " + dj.v);
        }
    }
}
```
Rule 6.2 example

```c
// compute the factorial of integer n
void main ( ) {
    int n, i, result;  // These must all be unique
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
```
Type Rule 6.3

A program is valid if

– its Declarations are valid and
– its Block body is valid with respect to the type map for those Declarations

public static void V (Program p) {
    V (p.decpart);
    V (p.body, typing (p.decpart));
}
Rule 6.3 Example

// compute the factorial of integer n
void main ( ) {
    int n, i, result;  
    n = 8; 
    i = 1; 
    result = 1; 
    while (i < n) {
        i = i + 1; 
        result = result * i; 
    }
}

These must be valid.
Type Rule 6.4

Validity of a Statement:

– *A Skip is always valid*

– *An Assignment is valid if:*

  • Its target *Variable* is declared
  • Its source *Expression* is valid
  • If the target *Variable* is float, then the type of the source *Expression* must be either float or int
  • Otherwise if the target *Variable* is int, then the type of the source *Expression* must be either int or char
  • Otherwise the target *Variable* must have the same type as the source *Expression*. 
Type Rule 6.4 (continued)

- *A Conditional is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its thenbranch and elsebranch *Statements* are valid

- *A Loop is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its Statement body is valid

- *A Block is valid if all its Statements are valid.*
Rule 6.4 Example

// compute the factorial of integer n
void main ( ) {
    int n, i, result;
    n = 8;  // this assignment is valid if:
    i = 1;  // n is declared,
    result = 1;  // 8 is valid, and
    while (i < n) {
        i = i + 1;  // the type of 8 is int or char
        result = result * i;
    }
}

This assignment is valid if:
  n is declared,
  8 is valid, and
  the type of 8 is int or char
  (since n is int).
Rule 6.4 Example

// compute the factorial of integer n
void main ( ) {
    int n, i, result;
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
Type Rule 6.5

Validity of an Expression:

- A Value is always valid.
- A Variable is valid if it appears in the type map.
- A Binary is valid if:
  - Its Expressions term1 and term2 are valid
  - If its Operator op is arithmetic, then both Expressions must be either int or float
  - If op is relational, then both Expressions must have the same type
  - If op is && or ||, then both Expressions must be bool
- A Unary is valid if:
  - Its Expression term is valid,
  - …
Type Rule 6.6

The type of an Expression $e$ is:

- If $e$ is a Value, then the type of that Value.
- If $e$ is a Variable, then the type of that Variable.
- If $e$ is a Binary $\text{op} \; \text{term}_1 \; \text{term}_2$, then:
  - If $\text{op}$ is arithmetic, then the (common) type of $\text{term}_1$ or $\text{term}_2$
  - If $\text{op}$ is relational, && or ||, then bool
- If $e$ is a Unary $\text{op} \; \text{term}$, then:
  - If $\text{op}$ is ! then bool
  - ...

Rule 6.5 and 6.6 Example

// compute the factorial of integer n
void main ( ) {
    int n, i, result;
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
6.2 Implicit Type Conversion

Clite Assignment supports implicit widening conversions

We can transform the abstract syntax tree to insert explicit conversions as needed.

The types of the target variable and source expression govern what to insert.
Example: Assignment of int to float

Suppose we have an assignment

\[ f = i - \text{int}(c); \]

(f, i, and c are float, int, and char variables).

The abstract syntax tree is:
Example (cont’d)

So an implicit widening is inserted to transform the tree to:

Here, $c2i$ denotes conversion from char to int, and $i2f$ denotes conversion from int to float.

Note: $c2i$ is an explicit conversion given by the operator int() in the program.