Naming and Data Types

Chapters 4-6

Naming and Typing

- We’ll go quickly and skip a bit of material from chapters 4-5 as they should be familiar to you already
  - Scoping
  - Overloading
  - IEEE 754 representation
  - Etc.
- We’ll cover major definitions for chapters 4-5
Binding and Naming

- Recall that the term binding is an association between an entity (such as a variable) and a property (such as its value).
- A binding is static if the association occurs before run-time.
- A binding is dynamic if the association occurs at run-time.
- Name bindings play a fundamental role.
- The lifetime of a variable name refers to the time interval during which memory is allocated.

Variables

- Basic bindings
  - Name
  - Address
  - Type
  - Value
  - Lifetime
  - Scope
Scoping

- The scope of a name is the collection of statements which can access the name binding.
- In static scoping, a name is bound to a collection of statements according to its position in the source program.
  - Most modern languages use static (or lexical) scoping.

Dynamic Scoping

- Two different scopes are either nested or disjoint.
  - In disjoint scopes, same name can be bound to different entities without interference.
  - Nested scope is scope within scope (e.g. blocks within blocks like Java)
- What constitutes a scope? Depends on language
  - Java: Package, Class (nested), Function, Block (nested), Loop
  - C: Function, Block (nested)
1 void sort (float a[ ], int size) {
2   int i, j;
3   for (i = 0; i < size; i++)  // i, size local
4     for (int j = i + 1; j < size; j++)
5       if (a[j] < a[i]) {  // a, i, j local
6         float t;
7         t = a[i];       // t local; a, i nonlocal
8         a[i] = a[j];
9         a[j] = t;
10      }
11    // invalided to reference j here
11 }

References

• Forward Reference
  – A reference to a name that occurs before the
    name has been declared
  – Required for many languages, e.g. Java/C++

• C is more restrictive
  – All declarations must precede all other
    statements in a block
Symbol Table

- A symbol table is a data structure kept by a translator that allows it to keep track of each declared name and its binding.
- Assume for now that each name is unique within its local scope.
- The data structure can be any implementation of a dictionary, where the name is the key.
  - Dictionary = Map (mapping from a key to a value)

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   a) If found, return the binding.
   b) Otherwise, repeat the process on the next dictionary down in the stack.
   c) If the name is not found in any dictionary, report an error.
Resolving References

- For static scoping, the *referencing environment* for a name is its defining scope and all nested subscopes.
- The referencing environment defines the set of statements which can validly reference a name.
Dynamic Scoping

- In dynamic scoping, a name is bound to its most recent declaration based on the program’s call history.
- Used be early Lisp, APL, Snobol, Perl.
- Symbol table for each scope built at compile time, but managed at run time.
- Scope pushed/popped on stack when entered/exited.
1 int h, i;
2 void B(int w) {
3    int j, k;
4    i = 2*w;
5    w = w+1;
6    ...
7 }
8 void A (int x, int y) {
9    float i, j;
10   B(h);
11   i = 3;
12   ...
13 }
14 void main() {
15   int a, b;
16   h = 5; a = 3;
17   b = 2;
18   A(a, b);
19   B(h);
20 }

Call history
main (17) → A (10) → B

Function Dictionary
B <w, 2> <j, 3> <k, 3>
A <x, 8> <y, 8> <i, 9> <j, 9>
main <a, 15> <b, 15>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to <i, 9> in A.
main(18) → B ? Reference to i(4)?

Visibility

• A name is visible if its referencing environment includes the reference and
  the name is not redeclared in an inner scope.
• A name redeclared in an inner scope effectively hides the outer declaration.
• Some languages provide a mechanism for referencing a hidden name; e.g.: this.x
  in C++/Java.
1 public class Student {
2    private String name;
3    public Student (String name, ...) {
4       this.name = name;
5       ...
6    }
7 }
Lifetime

• The *lifetime* of a variable is the time interval during which the variable has been allocated a block of memory.
• Earliest languages used static allocation.
• Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit.
• Usually scope = lifetime
  – Exception if local var declared static

Types

• A type is a collection of values and operations on those values.
  
  • Example: Integer type has values ..., -2, -1, 0, 1, 2, ... and operations +, -, *, /, <, ...

  • The Boolean type has values true and false and operations AND, OR, NOT.
Types

• Computer types have a finite number of values due to fixed size allocation; problematic for numeric types.
• Exceptions:
  – Smalltalk uses unbounded fractions.
  – Haskell type Integer represents unbounded integers.
• Floating point problems?

Type System

• A type error is any error that arises because an operation is attempted on a data type for which it is undefined.
• Type errors are common in assembly language programming.
• High level languages reduce the number of type errors.
• A type system provides a basis for detecting type errors.
Static and Dynamic Typing

- A type system imposes constraints such as the values used in an addition must be numeric.
  - Cannot be expressed syntactically in EBNF.
- Some languages perform type checking at compile time (e.g., C).
- Other languages (e.g., Perl) perform type checking at run time.
- Still others (e.g., Java) do both.

- A language is *statically typed* if the types of all variables are fixed when they are declared at compile time.
- A language is *dynamically typed* if the type of a variable can vary at run time depending on the value assigned.
- Can you give examples of each?
A language is *strongly typed* if its type system allows all type errors in a program to be detected either at compile time or at run time.

A strongly typed language can be either statically or dynamically typed.

*Union* types are a hole in the type system of many languages.

Most dynamically typed languages associate a type with each value.

---

**Types**

- **Basic Types**
  - Integer
  - Char
  - Real
  - String
- **Non-Basic Types**
  - Enumeration
    ```java
    enum Day { Monday, Tuesday, Wednesday; }
    for (Day d : Day.Values())
        System.out.println(d);
    ```
  - Pointers
    - Java?
  - Arrays and Lists
  - Classes
  - Strings
  - Structs
Variant Records and Unions

• Back when memory was scarce…
  – Variant records allowed two or more different fields to share the same block of memory
  – Called Variant in Pascal, Union in C

```c
union myUnion {
    int i;    // 32 bits of storage
    float f; // Same 32 bits of storage
};
Union myUnion u;
```

u.i accesses storage as Integer
u.f accesses storage as float

How might we do something in Java that allows accesses to a value that might be of different types?

Functions as Types

• Some languages allow functions to behave as “first class citizens”
  – Function can be treated like a data type or variable
  – Can pass a function as an argument
• Pascal example:
  – function newton(a, b: real; function f: real): real;
  – Know that f returns a real value, but the arguments to f are unspecified.
Java Example

```java
public interface RootSolvable {
    double valueAt(double x);
}

public class MySolver implements RootSolvable {
    double valueAt(double x) {
        ...
    }
}

public double Newton(double a, double b, RootSolvable f) {
    ...
    val = f.valueAt(x);
    ...
}
```

Not a true first-class citizen since a function can’t be constructed and returned by another function

Type Equivalence

- Sometimes we need to know when two types are equivalent, but this can be trickier than it sounds

```c
struct complex {
    float re, im;
};
struct polar {
    float x, y;
};
struct {
    float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
// which are equivalent types?
```
Type Equivalence

• Name Equivalence
  – Two types are the same if they have the same name
• Structural Equivalence
  – Two types are the same if they have the same structure

Polymorphism

• Skipping this topic in the book, you should be familiar with it already
Generics

• Apparently not everyone is familiar with Generics, so we'll cover this with respect to Java a little later in the class
• Generics allow us to instantiate some class that takes as input another class
  – E.g.
    • ArrayList<Integer>
    • ArrayList<MyClass>
    • GenericSort<MyClass>

Type Checking

• Concern for program reliability and early detection of errors has led to the strengthening of type checking in languages
  – 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
• Type checking is done either at compile time or at run time (or both)
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)

```c
// 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;
    }
}
```

Informally, the variables are the same type and are declared, seems OK
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., $\{<n, \text{int}>, <i, \text{int}>, <\text{result}, \text{int}>,\}$
  - Can implement as a hash table
  - Function typing creates a type map
  - Function typeOf retrieves the type of a variable:
    $\text{typeOf}(id) = \text{type}$
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;
}
```

TypeMap

```java
public class TypeMap extends HashMap<Variable, Type> {
    // TypeMap is implemented as a Java HashMap.
    // Plus a 'display' method to facilitate experimentation.

    public void display () {
        System.out.print("{ ");
        String sep = "";
        for (Variable key : keySet()) {
            System.out.print(sep + "<" + key + ", " + get(key).getId() + ">");
            sep = ", ";
        }
        System.out.println(" }"壹)
    }
}
```
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++)
        for (int j=i+1; j<d.size(); j++) {
            Declaration di = d.get(i);
            Declaration dj = d.get(j);
            check( ! (di.v.equals(dj.v)),
                   "duplicate declaration: " + dj.v);
        }
}
```

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

```java
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

```c
// compute the factorial of integer n
void main ( ) {
    int n, i, result;
    n = 8; // n is declared,
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        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

```plaintext
// 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;
    }
}
```

This loop is valid if
i < n is valid,
i < n has type bool, and
the loop body is valid

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 $op \; term1 \; term2$, then:
    • If $op$ is arithmetic, then the (common) type of term1 or term2
    • If $op$ is relational, && or ||, then bool
  – If e is a Unary $op \; term$, then:
    • If $op$ is $!$ then bool
    • ...

Rule 6.5 and 6.6 Example

```c
// 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;
    }
    result = result * i;
}
```

This Expression is valid since: op is arithmetic (*) and the types of i and result are int. Its result type is int since: the type of i is int.
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
- \( f = i - \text{int}(c); \)
- \((f, i, \text{ and } c \text{ are float, int, and char variables}).\)
- 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 \( itof \) denotes conversion from int to float.

- Note: \( c2i \) is an explicit conversion given by the operator \( \text{int()} \) in the program.

Formalizing the Clite Type System

Type map: \[ tm = \{ <v_1, t_1>, <v_2, t_2>, ..., <v_n, t_n> \} \]

Created by:
(Type Rule 6.1)

\[ \text{typing}: \text{Declarations} \rightarrow \text{TypeMap} \]

\[ \text{typing}(d) = \bigcup_{i \in [1, n]} <d_i, v, d_i, t> \]

Validity of Declarations:
(Type Rule 6.2)

\[ V: \text{Declarations} \rightarrow \text{B} \]

\[ V(d) = \forall i, j \in [1, n] (i \neq j \Rightarrow d_i, v \neq d_j, v) \]
Validity of a Clite Program

(Type Rule 6.3)

\[ V : Program \to B \]
\[ V(p) = V(p.\text{decpart}) \land V(p.\text{body}, \text{typing}(p.\text{decpart})) \]

Validity of a Clite Statement

(Type Rule 6.4, simplified version for an Assignment)

\[ V : Statement \times TypeMap \to B \]
\[ V(s, tm) = \text{true} \quad \text{if } s \text{ is a Skip} \]
\[ = s.\text{target} \in tm \land V(s.\text{source}, tm) \land \text{typeOf}(s.\text{target}, tm) = \text{typeOf}(s.\text{source}, tm) \]  
\[ = V(s.\text{test}, tm) \land \text{typeOf}(s.\text{test}, tm) = \text{bool} \land V(s.\text{thenbranch}, tm) \land V(s.\text{elsebranch}, tm) \]
\[ = V(s.\text{test}, tm) \land \text{typeOf}(s.\text{test}, tm) = \text{bool} \land V(s.\text{body}, tm) \]
\[ = V(b_1, tm) \land V(b_2, tm) \land \ldots \land V(b_n, tm) \quad \text{if } s \text{ is a Block} \]
Validity of a Clite Expression
(Type Rule 6.5, abbreviated versions for Binary and Unary)

\[ V : \text{Expression} \times \text{TypeMap} \rightarrow \text{B} \]
\[ V(e, tm) = \text{true} \quad \text{if } e \text{ is a Value} \]
\[ = e \in tm \quad \text{if } e \text{ is a Variable} \]

\[ = V(e.\text{term}_1, tm) \land V(e.\text{term}_2, tm) \land \]
\[ \text{typeOf}(e.\text{term}_1, tm) \in \{\text{float, int}\} \land \]
\[ \text{typeOf}(e.\text{term}_2, tm) \in \{\text{float, int}\} \land \]
\[ \text{typeOf}(e.\text{term}_1, tm) = \text{typeOf}(e.\text{term}_2, tm) \quad \text{if } e \text{ is a Binary} \land \]
\[ e.\text{op} \in \text{ArithmeticOp} \cup \text{RelationalOp} \]

\[ = V(e.\text{term}, tm) \land e.\text{op} = ! \land \]
\[ \text{typeOf}(e.\text{term}, tm) = \text{bool} \quad \text{if } e \text{ is a Unary} \]

Type of a Clite Expression

(Type Rule 6.6, abbreviated version)

\[ \text{typeOf} : \text{Expression} \times \text{TypeMap} \rightarrow \text{Type} \]
\[ \text{typeOf}(e, tm) = e.\text{type} \quad \text{if } e \text{ is a Value} \]
\[ = e.\text{type} \quad \text{if } e \text{ is a Variable} \land e \in tm \]

\[ = \text{typeOf}(e.\text{term}_1, tm) \quad \text{if } e \text{ is a Binary} \land e.\text{op} \in \text{ArithmeticOp} \]
\[ = \text{boolean} \quad \text{if } e \text{ is a Binary} \land e.\text{op} \notin \text{ArithmeticOp} \]

\[ = \text{typeOf}(e.\text{term}, tm) \quad \text{if } e \text{ is a Unary} \land e.\text{op} = - \]
\[ = \text{boolean} \quad \text{if } e \text{ is a Unary} \land e.\text{op} = ! \]
Summary

• Once the formal type system is defined, the translation into code becomes relatively straightforward.
• With a strong type system, many run-time errors can be prevented at compile time or at least more clearly understood at run time.