Programming Languages

Lecture 6: Bindings

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Lecture Outline

- Variables
- Scope
- Lifetime
- Value
- Names vs. Locations

Binding Time

- Attributes of parts of programs must be "bound" to object before or during computation.
- A binding fixes a value or other property of an object (from a set of possible values)
- Time at which choice for binding occurs is called binding time.
  - Dynamic binding — at execution
  - Static binding — at translation, language implementation, or language definition

Dynamic Binding

- At entry to block or subprogram
  - Bind actual to formal parameter
  - Determine location of local variable
- At arbitrary times in program — bind values to variables via assignment

Static Binding

- At translation
  - Determined by programmer — bind type to variable name, values to constants
  - Determined by translator — bind global variable to location (at load time), bind source program to object program representation
- At implementation
  - Bind values to representation in computer
  - Bind operations and statements to semantics (if not uniform may lead to different results with different implmentations)

Static Binding (cont)

- At language definition
  - Structure of language
  - Built-in and definable types
  - Notation for values

Binding Time Examples

1. When is meaning of "`+`" bound to its meaning in "`x + 10`"?
   - Could be at language definition, implementation, or at translation
   - May also be execution time — could depend on type of `x` determined at run-time
2. Difference between reserved and keywords has to do with binding time
   - Both bound at language definition, but reserved word binding can't be changed
   - Ex. "`DO`" is reserved word in Pascal, but not FORTRAN (can write `DO = 10`)
   - Ex. "Integer" may be redefined in Pascal, but not FORTRAN or Ada.

Late vs. Early Binding Time

- Many language design decisions relate to binding time
  - Late — more flexible
  - Early — more efficient
- Ex. More efficient to bind "+" at translation than execution
- Early — supports compilation, late — supports interpretation
- Programming choices may delay binding time
- Ex. recursion forces delay in binding time of local variables to locations (FORTRAN allows choice: static allocation vs stack-based allocation)
- Generally considered useful to bind ASAP

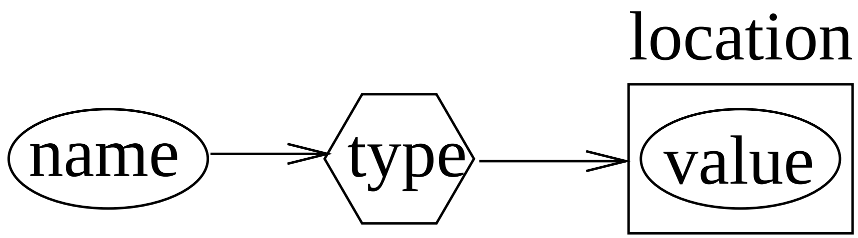
Managing Bindings

- Bindings stored both at compile and at run-time.
- Compilation
  - Declarations stored in Symbol table ($Names \rightarrow Attributes$)
  - Most bindings used only in the compilation process
- Execution
  - Run-time environment keeps track of meanings of names ($Names \rightarrow Locations$)
  - Contents of locations stored in memory (also called the *state*) ($Locations \rightarrow Values$)
- An interpreter keeps all 3 kinds of bindings together in one environment

Variables

- Variable has 6 components
  1. Name
  2. Type
  3. Location or reference (l-value)
  4. Value (r-value)
  5. Scope - where variable accessible and manipulable - static vs dynamic
  6. Lifetime - interval of time in which location bound to variable
- Scope and Lifetime same in some languages — different in others (FORTRAN)

Variables (cont)



Using Variables

- What does "`N := N + 1`" mean?
- First `N` refers to location (l-value)
- Second `N` to value (r-value).

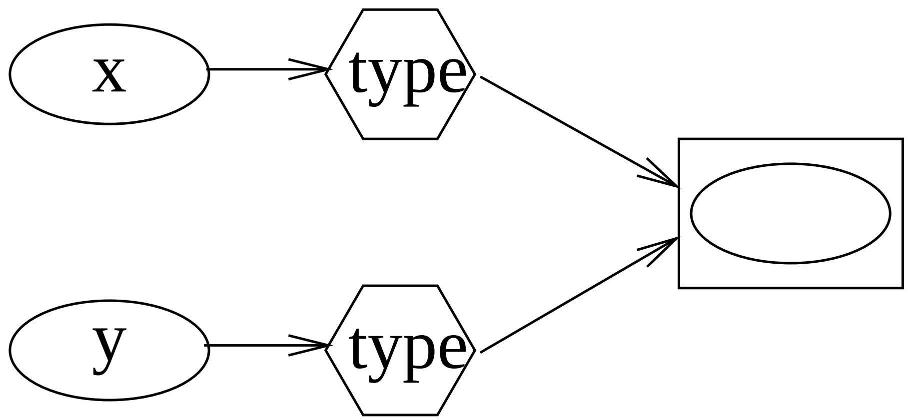
Dereferencing

- *Dereferencing* — obtaining value of variable
- Explicit in some languages. In ML write `N := !N + 1`
- Explicit pointer dereferencing (Pascal: `p^`, C: `*p`)
- Array access (`A[i]`) is a reference valued expression — most expressions only give r-values

Changing Variable Attributes

- Common to think of changing value of variable at runtime
- Other attributes may change
- Ex. Name can change (via call-by-reference parameter)
- Called *aliasing*

Aliasing



- Call-by-reference parameters
- Assignment of variables (e.g., `x := y`)
  - Copying: target variable retains its location, but copies new value from source variable.
  - Sharing: target variable gets location of source variable, so both share the same location (like objects in Java).

Denotable Values

- Can classify languages by sorts of entities that can be bound to an identifier
- Ex. Pascal
  - Primitive values and strings (in constant definitions)
  - References to variables and associated types (in variable declarations)
  - Procedure and function abstractions (in procedure and function definitions)
  - Types (in type definitions)

Scope

- Scope of a variable is the range of program instructions where variable is known.
- Can be *static* or *dynamic*

Static Scoping

- Used by most languages (e.g., Pascal, Modula-2, C, ...)
- Scope is associated with the static text of the program
- Can determine scope by looking at structure of program
- May have holes in scope of variable

```
program ...
    var M : integer;
    ....
    procedure A ...
        var M : array [1..10] of real; (* hides M in program *)
        begin
              ...
        end;
begin
        ...
end.
```

Static Scoping (cont)

- Symbol table keeps track of which declarations are currently visible.
- Symbol table like stack — search from top, so when enter a new scope, push new declarations. When exit scope, pop declarations.

Dynamic Scoping

- Scope determined by the execution path
- An occurrence of an identifier in a procedure may refer to different variables in different procedure invocations
- With dynamic scoping, symbol table built and maintained at run-time
- Push and pop entries when enter and exit scopes at run-time
- Dynamic scoping usually associated with dynamic typing
- LISP and APL use dynamic scoping (though Scheme has default of static)

Dynamic Scoping (cont)

```
program ...
        var A : integer;
        procedure Y(...);
                begin  ...; B := A + B; ...
                end; {Y}
        procedure Z(...);
                var A: integer;
                begin ...; Y(...); ...
                end; {Z}
        begin {main} ...; Z(...);...
        end.
```

Question: Which variable with name `A` is used when `Y` is called from `Z`?

- Static: globally defined `A`.
- Dynamic: local `A` in `Z` (declaration in `Z` is most recent)

Lifetime

- Static allocation (FORTRAN)
  - All variables are allocated storage before execution of program begins.
  - When return to a procedure local variables still have value left at end of previous invocation.
- Dynamic allocation (Pascal, C...)
  - When enter procedure any local variables are allocated and are then deallocated when exit.
  - Uses activation records

Activation Records

- In block-structured language (Pascal, C, Modula-2, ...)
- Activation record has space for local variables and parameters of procedure, function, block, etc.
- Allocate space for activation record on run-time stack at invocation
- Pop record when exit unit
- Note that a procedure may have several activation records on stack if called recursively.
- May have several distinct variables on stack with same name

Heap Memory

- Dynamic allocation (pointers) uses “heap” memory
- Lifetime determined by use of `new` and `dispose` functions
- Pascal has three kinds of memory:
  - static (occupied by global variables),
  - stack-based or automatic (occupied by parameters and local variables of procedures and functions),
  - heap-based or manually allocated.
- ML: automatically allocate from heap when needed and deallocated when no way of accessing it (by garbage collector).
- Java similar.

Value Bindings

- Value not necessarily bound at execution time
- *Language defined constant* — bound at language definition time (e.g., maxint, true, false)
- *Program constant* — bound at compilation

```
const size = 100;
      doubleSize = 2 * size; (* manifest constant *)
```

- Some languages allow binding at procedure entry

```
procedure ... (n : integer) is
  var x: constant integer := 3 * n - 2; (* binding *)
  A:  array[1..n] of real;
```

Value Bindings (cont)

- Initialization of variables at declaration.

```
var x : integer := 5;
```

- Initialization can be done first or every time procedure is entered
  - FORTRAN only first time
  - Java every time

Names vs Locations

- Two expressions are said to be aliases if they denote the same location
- If have `p(&x,&y)`, then the call `p(z,z)` makes `x` and `y` aliases in `p`
- Aliasing often producing undesirable behavior in functions
- Ex. If the body of `p(x,y)` first increases `x` by one and then `y` by one, `z` increases by 2
- Aliasing with pointers:

  ```
  int *x, *y; ...  x := y;
  ```

  Then `*x` and `*y` are aliases — changing one changes the other
- In languages with assignment by sharing (e.g., Java), get aliasing automatically with all assignments.

Pointers

- Recognized as major cause of run-time errors.
- Problems:
  1. If type not specified (PL/I), then can break type system.
  2. Dangling pointers
     (a) If pointers can point to object on run-time stack (named variable — PL/I, C), then object may go away before pointer.
     (b) User may explicitly deallocate pointer even if other variables still point to same object. Possible solutions involve reference counting or garbage collection.

Pointers

- Problems (cont)
  1. Dereferencing uninitialized or nil pointers may cause crashes.
  2. Garbage: Unreachable items may clog heap memory and can't recycle. Garbage collection or reference counting may solve.
  3. Holes in typing system may allow arbitrary integers to be used as pointers (through variant records in Pascal)
- Pointer arithmetic possible in C
  - Note that $p + 1$ for pointer is not same as $p + 1$ for integer
  - For pointer, address incremented by size of object pointed to (e.g., array indexing)