

The Structure of Typed Programming Languages
(Chapter 4)

CS386L- Programming Languages

Dr. Greg Lavender
Department of Computer Sciences
The University of Texas at Austin

CS386L: Lecture-11 - The Structure of Typed Programming Languages: Chapter 4

The Qualification Principle

- any semantically meaningful syntactic class may admit local definitions
 - want to define "lexical scopes" for declarations
 - so we introduce new syntax and semantics for blocks, which admit local declarations
- lexically (statically) scoped, block structured languages are the norm today
 - all descendents of Algol
 - imperative style: C, C++, Java
 - functional style: Scheme, ML, Haskell
- Lisp is dynamically scoped, not lexically scoped

4/12/04 2

Block Structures

- any construct that admits local definitions is a block structure, or just "block"
- For some syntax domain U , define:
 - $U ::= \dots \mid \text{begin } D \text{ in } U \text{ end}$
 - the semantics are that declarations D are only visible in the body of U
 - the typing rule relies on combining two type assignments, where bindings of the local declaration names "cancel out" bindings in the outer type assignment, and local bindings are only in U

$$\frac{\pi_1 \vdash D : \pi_2 \text{dec} \quad \pi_1 U \vdash \pi_2 \vdash U : \theta}{\pi_1 \vdash \text{begin } D \text{ in } U \text{ end} : \theta}$$

4/12/04

3

Correspondence with Parameters

- note the correspondence of local definitions and parameter definitions
 - define $I_1(I_2 : \theta) = U$ in $I_1(V)$
 - begin define $I_2 = V$ in U end
 - the two are semantically equivalent
 - i.e., parameters are just another form of name binding
 - recall that in scheme we have a correspondence between `let`, `let*` and lambda expressions
 - $(\text{let } ((I_2 V)) U) = (\text{lambda } (I_2) U) V$
 - $(\text{let } ((x 1) (y 2)) (+ x y)) = ((\text{lambda } (x y) (+ x y)) 1 2)$
 - $(\text{let}^* ((x 1) (y 2)) (+ x y)) = (((\text{lambda } (x) (\text{lambda } (y) (+ x y))) 1) 2)$

4/12/04

4

Command Blocks

- command blocks limit the visibility of declarations to the local body
 - since decls are only needed on entry to a block and are not required after block exit, we can arrange to allocate local storage on entry and "free" it on exit
 - a stack is a good data structure to implement this type of semantics for block structured languages
 - type assignments grow and shrink in LIFO order
 - the run-time stack was invented along with Algol, which introduced block structure, and hence recursive functions were then enabled

4/12/04

5

Command Blocks

$$\pi_1 \mid - D: \pi_2 \text{dec} \quad \pi_1 \cup \pi_2 \mid - C: \text{comm}$$

$$\pi_1 \mid - \mathbf{begin} D \mathbf{in} C \mathbf{end}: \text{comm}$$

```
begin
var A:newint;
proc P = begin var C:newint in C:=@A end;
proc Q = begin var B:newint; var A:newint in A:=1; call P end
in
  A:=0;
  call P;
  call Q
end
```

4/12/04

6

The Store as a Stack

- for local definitions, we need to treat the storage vector as a stack

Store = $\{ \langle n_1, n_2, \dots, n_m \rangle \mid n_i \in \text{Int}, 1 \leq i \leq m, m \geq 0 \}$

lookup: Location \times Store \rightarrow Int
 lookup($\text{loc}_i, \langle n_1, n_2, \dots, n_i, \dots, n_m \rangle$) = n_i

update: Location \times Int \times Store \rightarrow Store
 update($\text{loc}_i, n_i, \langle n_1, n_2, \dots, n_i, \dots, n_m \rangle$) = $\langle n_1, n_2, \dots, n_i, \dots, n_m \rangle$

allocate: Store \rightarrow Location \times Store
 allocate($\langle n_1, n_2, \dots, n_m \rangle$) = ($\text{loc}_{m+1}, \langle n_1, n_2, \dots, n_m, \text{init} \rangle$)

size-of: Store \rightarrow Int
 size-of $\langle n_1, n_2, \dots, n_m \rangle$ = m

free: Int \rightarrow Store \rightarrow Store
 free $i \langle n_1, n_2, \dots, n_i, n_{i+1}, \dots, n_m \rangle$ = $\langle n_1, n_2, \dots, n_i \rangle$

4/12/04

7

Semantics of Command Blocks

- See example in figure 4.3 on page 108
 - the command block controls the size of the store by allowing D to allocation store locations for the body C .
 - $[[\pi \mid \text{begin } D \text{ in } C \text{ end: comm}]] e s = \text{free}(\text{size-of } s) s_2$
 where $(e_1, s_1) = [[\pi \mid D: \pi_1 \text{dec}]] e s$
 and $s_2 = [[\pi \cup \pi_1 \mid C: \text{comm}]] (e \cup e_1) s_1$
 - note that the store behaves like a stack. What is the lifetime of the local variables?

4/12/04

8

Scope

- How is an invocation associated with the definition that it invokes?
 - it depends on the scoping policy of the language
 - static (lexical) scoping vs dynamic scoping
 - we often say lexical scoping because the bindings are *lexically apparent*
 - Algol descendents use static scoping policies
 - an invocation of I is associated with the definition of I whose scope contains the invocation
 - the scope of the definition of I in is U, minus any part of U that falls in the scope of the redefinition of I
 - begin define I=V in U end
 - begin
 - var A:newint;
 - proc P = begin var C: newint in C:=@A end
 - proc Q = begin var B:newint; var A:newint in [hole] end
 - in A:=0; call P; call Q end

4/12/04

9

Semantics of Static Scoping

- static scoping allows the calculation of the associations between invocations and definitions in advance of running the program
- the environment that applies to a set of commands holds the bindings of those identifiers whose scope is in force
 - e contains the global bindings before the command block creates new bindings e_1 , whose scope is the body C
 - like bindings in e_1 cancel out like bindings in e, so $e \cup e_1$ is the environment for C and the bindings in e_1 are only applied in C
 - $[[\pi \mid \text{being } D \text{ in } C \text{ end}; \text{comm}]]e \ s = \text{free}(\text{size-of } s) \ s_2$
 where $(e_1, s_1) = [[p \mid D:\pi_1 \text{dec}]]e \ s$
 and $s_2 = [[\pi \cup \pi_1 \mid C:\text{comm}]](e \cup e_1) \ s_1$

4/12/04

10

Dynamic Scoping

- static scoping detects errors before a command sequence is evaluated
- with dynamic scoping, you can encounter errors during evaluation
 - example: the call to P causes the inner binding of A to be used that a function identifier appears on the LHS of A:=0 in proc P. This is a type error since the type of A is not an intloc

```

begin
  var A:newint;
  proc P= A:=0
  in begin
    fun A = 1
    in call P end
  end

```

4/12/04

11

Dynamic Scoping

- With dynamic scoping, the environment that applies to the evaluation of a command is not determined until the command is evaluated
 - i.e., the environment acts like a store in that it is dynamic, not static
 - in general, dynamic scoping can lead to subtle type errors and poorly understood program behavior
 - it also becomes difficult, or sometimes impossible, to write the typing rules for the language!

4/12/04

12

Extent

- The lifetime or extent of a variable is the duration of time in which the variable is "live"
 - typical lifetimes are:
 - the entire program
 - the current procedure
 - the lifetime of the enclosing object
 - until it is explicitly killed (i.e., freed)
 - until it is implicitly killed (i.e., garbage collected)
 - Lifetime of a variable may differ from its scope
 - in the call to P in proc Q, B and A remain allocated but are not visible in the evaluation of P.
 - begin
 - var A:newint;
 - proc P = begin var C: newint in C:=@A end
 - proc Q = begin var B:newint; var A:newint in call P end
 - in A:=0; call P; call Q end

4/12/04

13

Extent

- the command block allocates locations for local variables on entry and frees them on exit
 - are we sure that we should free them on exit?
 - in what ways could a local variable "escape" from a command block?
 - the result of the command block is the location of a local variable
 - in C this is sometimes a mistake
 - char* f() { char s[] = "hello, world"; ...; return s; }
 - a value assigned to a non-local variable contains a local variable's location, i.e., aliasing
 - int* p; void f() { int x; ...; p = &x; ... } // bad idea!
 - a parameter is used to transmit a local variables location to a non-local abstraction that saves the location
 - void f() { int x; g(&x); ... } void g(int* x) { static int* p = x; ... }

4/12/04

14

Extent

- We limit the result of a command block to be a store, and since the store only contains values, and not locations, we avoid problem 1
 - but of course, in real languages like C and C++, the store can hold locations, and so aliasing introduces all kinds of opportunities to errors, but aliasing is also a form of powerful programming
 - we could add pointer variables to our language who values are restricted to locations
 - the problem is that we need to separate the store into a stack and a heap, so that we do not allow stack locations to be assigned to pointer variables, only heap locations
 - Java improves on C/C++ by only allowing reference variables to point to heap allocated (and garbage collected) objects

4/12/04

15

Other Forms of Blocks

- Declaration Blocks
 - allow the definition of modules that hide their implementation
 - information hiding
 - see examples in section 4.4
 - module M = begin var A:newint
 - in { proc INIT = A := 0
 - proc SUCC = A := @A+1
 - fun VAL = @A }
 - end
 - when M is evaluated, the location for A is allocated but only visible within the module. The scope of A is restricted to M but A's extent must escape the declaration block
 - a declaration block must not free storage upon completion of declarations. Only a command block can cause storage to be freed.
 - class K = begin var B:newint in record proc Q(X:intexp)=B:=X end;
 - proc P = begin import M; var C:K in call SUCC; call C.Q(VAL) end
 - in call M.INIT; call P end
 - when is M.A allocated and freed?

4/12/04

16

Other Forms of Blocks

- **Type-structure blocks**

- ```
class Personal-Stack =
begin
 var CTR: newint;
 var STACK: array[1..100] of newint
 in record
 proc PUSH(X:intexp) = if @CTR = 100 then skip
 else CTR := @CTR+1;
 STACK[@CTR] := X
 fi
 proc POP = if @CTR = 0 then skip else CTR:=@CTR-1 fi
 fun TOP = if @CTR=0 then fail else @(STACK[@CTR]) fi
 proc INIT = CTR := 0
 end
end
var A: Personal-Stack; var B: Personal-Stack ;
call A.INIT; call A.PUSH(0); call A.POP; ...
```
- like an "object" where an instance of a type-structure block: vars are private and methods (proc & fun) are public.

4/12/04

17

## C Examples

- **ANSI C**
  - program scope
    - file1.c: int x = -1; // visible in any file using extern int x
    - file2.c: extern int x;
  - file scope
    - file1.c: static int y = 0; // visible only in file1.c
  - block scope
    - { int y; ... { int x; float y; ... } ...}
    - if x or y exist in outer scope, they are masked by new decls
  - procedure scope - a named & parameterized block
    - int foo(int x){
 

```
register int y = 0; // register keyword is optional
auto float z = 1; // auto keyword is optional
static char buf[128]; // static keyword is required
// x, y, z, and buf only visible to statements in this procedure scope
...
}
```

4/12/04

18

## C++ Examples

- Much the same as C but with a few additions
- namespace scope - spans across files
  - file1.h: namespace Foo { ...; class X { ... }; ... }
  - file2.h: namespace Foo { ...; struct Y { ... }; ... }
  - file3.c: ...; using namespace Foo; ...;
- class scope - limited to same file
  - class X { private: ...; protected: ...; public: ...; };
  - struct Y { public: ...; private: ...; protected: ...; };
  - class scope opens private, struct scope opens public
- statement scope - decl scoped over 1 stmt
  - for (int i = 0; i < n; ++i) { ... }
    - syntactic sugar for: { int i; for (i = 0); i < n; ++i) {...} }
  - also temporaries have only statement scope
    - string x = string("hello") + string(" world");
      - two unnamed temporary string objects will be constructed, a third will result from operator+, and the result will be assigned to x, then all three temporaries will no longer be in scope and their destructors will be called

4/12/04

19

## C++ Examples

- What are the semantics of object construction/destruction under different C++ scopes?

```

string foo("external string object in program scope");
static string foo2("static string in file scope");
class X { string s; public: X(const char* v) : s(v) {...} ... };
int main() {
 X x("string instance in a class");
 string foo("stack string in proc scope");
 string* bar = new string("heap string in proc scope");
 while (cond) { string foo("local string in a loop"); ... };
 string a = string("temporary string object");
}

```

4/12/04

20

## C++ Scope Resolution Operator

- Scope resolution operator `::` in C++
  - used to refer to a name of another scope, or the global scope
    - `::operator new(s)`
      - refers to the global new operator
    - `std::cout << "hello, world";`
      - refers to the global name cout in the package std
    - `X::f(x, y);`
      - refers to a (static) method f in class or namespace X
    - `typedef void (X::*VFP)();`
      - refers to a point to a member function of the class X
    - `class X : public Y { ... virtual void f(x) { ...; Y::f(x); ... } ... }`
      - refers to the overridden member function f in an inherited class Y

4/12/04

21

## Imperative Scope Exit

- In imperative languages like C, C++, and Java a procedure scope is exited when control flow "falls off" the end of the scope
  - or there is a premature (conditional) exit using the return statement
    - `int foo { ...; if (error_cond) return -1; ...; return 0; }`
  - in C/C++ it is illegal to exit a scope using goto
    - `void f(); // forward decl`
    - `int main() { ...; f(); ...; L: exit(0); }`
    - `void f() { ...; goto L; ...; } // L outside of the scope of f!!`

4/12/04

22

## Java Examples

- The package and the class are the main scope mechanism
  - a Java program consists of one or more packages
    - builtin: java.lang, java.io, etc.
    - user defined: edu.utexas.cs.cs386.lavender.project
  - each package has one or more classes
    - file1.java: package Foo; class Bar {public ...; private...; ...; }
    - file2.java: import Foo; ...; Bar b = new Bar(...);
    - file3.java: ... System.out.println(...); ...
  - if a variable or method in a class is not public, private or protected, it defaults to package scope
    - visible to any class in the same package

## Functional Examples

- Haskell scopes
  - all definitions at the top-level have the whole program as their scope
    - function parameters are of course local names
  - local definitions can be defined in a couple of ways
    - let expressions
      - let  $x = 3+2$  in  $x^2 + 2*x - 4$
      - let  $x = 3+2; y=5-1$  in  $x^2 + 2*x - y$
    - where clauses
      - fact :: Integer -> Integer
      - fact n = tfact n 1
      - where
      - tfact 0 m = m
      - tfact n m = tfact (n-1) (n\*m)

## Haskell scopes

- list comprehension scopes
  - `[m + n | (m,n) <- zip list1 list2 ]`
    - `m` and `n` are variables local to the list comprehension
- forward declarations in mutual recursion
  - `isOdd, isEven :: Int -> Bool`
  - `isOdd n`
    - | `n <= 0 = False`
    - | `otherwise = isEven (n-1)`
  - `isEven n`
    - | `n < 0 = False`
    - | `n == 0 = True`
    - | `otherwise = isOdd (n-1)`

## Haskell Scopes

- forward declarations in where clauses
  - `maxsq x y`
    - | `sqx > sqy = sqx`
    - | `otherwise = sqy`
    - where
      - `sqx = sq x`
      - `sqy = sq y`
      - `sq :: Int -> Int`
      - `sq z = z*z`
  - note that `sq` is used before it is defined
  - the scope of `x` and `y` is over the whole body of the function, including the where clause
  - the scope of `z` is only the function `sq`

## Haskell Scopes

- Masking of local definitions
  - `maxsq x y`
    - | `sq x > sq y = sq x`
    - | `otherwise = sq y`
    - where `sq x = x * x`
  - the most local definition is used

4/12/04

27

## Haskell modules

- used to organize programs into separate modules
  - example from FunDNA
    - `module Base ( Base(A, C, G, T), watsonCrickComp, wcc )`  
 where
      - `data Base = A | C | G | T`  
 deriving (Eq, Enum, Ord, Bounded, Read, Show)
      - `watsonCrickComp :: Base -> Base -> Bool`  
`watsonCrickComp b b' = b == (wcc b')`
      - `wcc :: Base -> Base`  
`wcc A = T`  
`wcc T = A`  
`wcc C = G`  
`wcc G = C`

4/12/04

28

## Haskell Modules

- **Importing**
  - the standard Prelude (Prelude.hs) is automatically imported
  - other modules must be explicitly imported
    - `import Base`
  - importing can also hide selected named from the imported module
    - `import Prelude hiding (words)`
    - this allows us to define our own definition of "words"
    - alternatively, we can do a 'qualified' import and fully qualify the imported names
      - `import qualified Prelude ... Prelude.words`

## Homework

- **Chapter 4 Exercises:**
  - 1.1, 1.2, 3.1, 5.1, 6.2a,b, 6.3
- **Read Chapter 5**