

The Structure of Typed Programming Languages
(Chapter 2 - cont.)

CS386L- Programming Languages

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

CS386L: Lecture-08 - The Structure of Typed Programming Languages: Chapter 2 (cont).

Lazy Evaluation

- **call-by-name vs call-by-value**
 - call-by-value requires eager evaluation of arguments
 - like applicative order in the lambda calculus
 - recall that applicative order evaluation may not terminate
 - C, C++, C#, Java, Scheme, ML, etc.
 - call-by-name requires lazy evaluation of arguments
 - like normal order in lambda calculus
 - Miranda, Haskell, others?
- **In the Core Imperative Language, abstractions will use lazy evaluation with a "copy rule"**
 - the body of the named abstraction is not computed until the abstraction is invoked

3/29/04 2

Lazy Functions

- Example
 - $\text{fun } F = @loc_1 + 1$
in $loc_1 := 0; loc_1 := F; loc_2 := F + 2$
 - since F is lazily evaluated, $@loc_1 + 1$ is computed each time F is evaluated
 - let $s = \langle 2, 3 \rangle$, then $loc_1 = 2$ and $loc_2 = 3$
 - $s' = \langle 1, 4 \rangle$ because F evaluates to $(0+1)$ in the first invocation and to $(1+1)$ in the second evaluation
 - What would happen if F were eagerly evaluated at the point of declaration?

3/29/04

3

Copy Rule

- Lazy evaluation supports referential transparency via the copy rule
 - it's just substitution of a body for a name, which makes sense because abstraction is just "naming"
 - $[[\text{fun } I = E \dots \text{ in } \dots I \dots]] = [[\dots \text{ in } \dots E \dots]]$
 - just replace the invocation of I with the body E of I
 - but we have to avoid name conflicts as with β -reduction in the λ -calculus
 - copy rule for declarations of the form $D1, D2$ is
 - define $I_1 = V_1$, define $I_2 = V_2$, ..., define $I_n = V_n$ in U
 - $\Rightarrow [V_1/I_1, V_2/I_2, \dots, V_n/I_n]U$
 - which just says that we simultaneously substitute V_j for I_j in U , assuming each I_j is distinct
 - abstraction in this language is syntactic sugar

3/29/04

4

Substitution Example

- This is similar to preprocessor macro substitution
 - fun A=1, fun B = @loc₁ = 0
 - in while B do loc₁ := A+2 od
 - ⇒ [1/A, @loc₁=0/B] while B do loc₁:=A+2 od
 - = while @loc₁ = 0 do loc₁ := 1+2 od

3/29/04

5

Preprocessor Macro Substitution

- There are some gotchas, for example:
 - what's wrong with the following?

```
#define swap(a,b) temp = a; a = b; b = temp
...
if (cond == true)
  swap(x,y);
```

- does the following fix it in all cases?

```
#define swap(a,b) { temp = a; a = b; b = temp; }
...
if (cond == true)
  swap(x,y);
```

3/29/04

6

Preprocessor Macro Substitution

- What happens in this case?

```
#define swap(a,b) { temp = a; a = b; b = temp; }
...
if (cond == true)
    swap(x,y);
else
    ...
```

- What about this approach?

```
#define swap(a,b) do { temp = a; a = b; b = temp; }while(0)
...
if (cond == true)
    swap(x,y);
else
    ...
```

- macro copy rule: substitute a statement with a statement

3/29/04

7

What about type substitution?

- Could add it as a parameter to the macro

```
#define swap(a,b,t) do { t temp = a; a = b; b = temp;}while(0)
...
if (cond == true)
    swap(x,y,int);
else
    ...
```

- Better approach is to not use macros, but use inline, type-checked, polymorphic functions and let the compiler to the work

```
template<class T> void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

3/29/04

8

Simultaneous Substitution

- Given the CIL syntax rules thus far
 - $C ::= L := E \mid C_1; C_2 \mid \dots$
 - $E ::= N \mid @L \mid E_1 + E_2 \mid \dots \mid I$
 - $L ::= \text{loc}_i$
 - $N ::= n$
 - substitution is by recursive definition on the syntax rules
- For commands
 - $\text{assume } [V_j/I_j]U = [V_1/I_1, \dots, V_n/I_n]U$
 - $[V_j/I_j]L := E = [V_j/I_j]L := [V_j/I_j]E$
 - $[V_j/I_j]C_1; C_2 = [V_j/I_j]C_1; [V_j/I_j]C_2$
 - similarly for the other commands

3/29/04

9

Simultaneous Substitution

- For expressions
 - $[V_j/I_j] @L = @[V_j/I_j]L$
 - $[V_j/I_j]E_1 + E_2 = [V_j/I_j]E_1 + [V_j/I_j]E_2$
 - similarly for other expression forms
 - Invocations
 - $[V_j/I_j]I_k = V_k$ if $I = I_k$ for some $1 \leq k \leq n$
 - $[V_j/I_j]I = I$, if $I \neq I_k$ for all $1 \leq k \leq n$
- For locations and numbers
 - $[V_j/I_j] \text{loc}_i = \text{loc}_i$
 - $[V_j/I_j]n = n$

3/29/04

10

A Copy Rule for $D_1;D_2$

- Because of the sequential dependency of this form of declaration, the copy rule is a bit more complicated
 - $(\text{define } I_1=V_1, \dots, \text{define } I_n=V_n);D$
 - $\Rightarrow \text{define } I_1=V_1, \dots, \text{define } I_n=V_n, [V_1/I_1, \dots, V_n/I_n]D$
- What this rule means is that $D_1;D_2$ is transformed into D_1, D_2 by propagating the bindings of D_1 into D_2
 - note that if declarations cause side-effects, then this rule does not hold. Why?

3/29/04

11

Extending the Language Again

- other standard abstractions
 - command abstractions (proc & call)
 - numeral abstractions (const)
 - location abstractions (alias)
- ```

P ::= D in C
D ::= fun I=E | proc I=C | const I=N | alias I=L
 | D1,D2 | D1;D2
C ::= L:=E | C1;C2 | if E then C1 else C2 fi | while E do C od
 | skip | call I
E ::= N | @L | E1+E2 | E1=E2 | ~E | I
L ::= loci | I
N ::= n | I

```

3/29/04

12

## Useful Syntactic Sugar

- We can now name a lot more things and simplify our programs syntactically by using names for numbers, locations, and commands:

```
(const A = 1, alias X = loc1);
fun F = @X+A; // @loc1 + 1
proc P = X:=F // loc1 := @loc1+1
in
 call P; // loc1 := @loc1+1
 X := A; // loc1 := 1
 call P // loc1 := @loc1+1
```

3/29/04

13

## Evaluation

- Const and alias definitions do not depend on the store, so evaluation is meaningless
  - alias is not a variable declaration, just a name for a location
- Procedures are lazily evaluated like functions
  - procedures are names for command, which alter the store each time they are called
 
$$[[\pi \mid\text{- proc } I=C:\{I:\text{comm}\}\text{dec}]]e \text{ s} = \{I=p\}$$
 where  $p(s') = [[\pi \mid\text{- } C:\text{comm}]]e \text{ s}'$ 

$$[[\pi \mid\text{- call } I:\text{comm}]]e \text{ s} = p(s), \text{ where } (I=p) \text{ is in } e$$
- Could we evaluate procedures eagerly?
  - how do we deal with procedures that have loops?

3/29/04

14

## Variable Declarations

- Variables are more than aliases for locations
  - need to allocate a new location in the store
  - optionally initialize the location with a value
  - bind a name to the location
  - hence, a variable declaration is partly an abstraction and partly a command
    - we can think of it as an eagerly evaluated abstraction which side-effects the store
- Variable syntax
  - D ::= ... | var I
  - ...
  - L ::= I // no longer use location numbers, but var names

## Variable Declarations

- Example:
  - var X; var Y; proc P = Y := @X+1 in X := 0; call P
  - variable declarations create new locations in the store and names them X and Y
    - we no longer need to keep track of location numbers with variable abstractions
  - the previous copy rule does not apply to variable declarations

## Semantics of Variables

- Declarations now modify the store
  - previously, only commands altered the store  
 $[[\pi \mid \text{- var } I: \{I:\text{intloc}\}\text{dec}]]e \ s = (\{I=l\}, s')$   
 where  $(l, s') = \text{allocate}(s)$   
 and  $\text{allocate}: \text{Store} \rightarrow (\text{Location} \times \text{Store})$  is defined as  
 $\text{allocate } \langle n_1, n_2, \dots, n_m \rangle = (\text{loc}_{m+1}, \langle n_1, n_2, \dots, n_m, \text{init} \rangle)$   
 where  $\text{init}$  is some arbitrary integer value (e.g. 0)
  - what this says is that a variable declaration of type `intloc` defined on the current environment creates a pair consisting of a new binding  $\{I=l\}$  and an updated store  $s'$

3/29/04

17

## Modified Semantics of Abstractions

- for completeness, we need to modify the other abstractions to also return a (binding, store) pair, noting that lazily evaluated abstractions do not modify the store
  - $[[\pi \mid \text{- define } I=U:\{I:\theta\}\text{dec}]]e \ s = (\{I=f\}, s)$
  - where  $f \ s' = [[\pi \mid \text{- } U:\theta]]e \ s'$
- Semantics of programs changes slightly
  - $[[D \text{ in } C:\text{comm}]]s = [[\pi \mid \text{- } C:\text{comm}]]e_1 \ s_1$
  - where  $(e_1, s_1) = [[\text{Empty} \mid \text{- } D:\pi\text{dec}]]\text{Empty} \ s$
  - this says the environment and the store are built by declarations and the body of the program is evaluated relative to both

3/29/04

18

## Type-Structure Abstractions

- Variables declarations make storage allocations implicit, but we can make it explicit
  - introduce a new syntactic domain for "type-structures" that are command-like constructs that allocate storage

D  $\varepsilon$  Declarations  
T  $\varepsilon$  Type-structure

D ::= ... | var I:T  
T ::= **newint**

## Explicit Storage Allocation

- Each **newint** occurrence allocates a new storage location in the storage vector and returns the name of the location
  - has the semantics of the *allocate* function
  - var I:T "activates" T and binds the name I to the value (location) generated by T
    - compare to syntax & semantics in C++
      - `int* v = new int;`
  - we can extend the syntax slightly and allow initializations
    - var A:newint := 0
    - compare to C++
      - `int* v = new int(5);`

## Type-structure Abstractions

- We can extend the type-structure idea by introducing a "class" abstraction for it
  - $D ::= \dots \mid \mathbf{var} \ I:T \mid \mathbf{class} \ I=T$
  - $T ::= \mathbf{newint} \mid I$
- other languages use the keyword 'type'
- we use 'class' to give a more object-oriented flavor to the core language, for example:
 

```
class M = newint;
var A:M; var B:M; var C:newint
in A := 0; B := @A+@C
```
- class is a lazily evaluated abstraction, so  $\mathbf{var} \ X:M$  results in a new storage allocation for each declaration since the copy rule applies
  - note that a class is a type-structure abstraction that evaluates lazily and a variable declaration is a type-structure abstraction that evaluates eagerly

3/29/04

21

## Record Structure

- Classes are more interesting when they have more structure
  - $D ::= \dots \mid \mathbf{var} \ I:T \mid \mathbf{class} \ I=T$
  - $T ::= \mathbf{newint} \mid I \mid \mathbf{record} \ D \ \mathbf{end}$
- records allocate storage for each contained declaration and returns a "value" for the record
 

```
class R = record var X:newint, var Y:newint end;
var A:R; var B:R in ...
```

3/29/04

22

## More On Records

- records can also contain functions, procedures, etc.

```
var A:newint;
class R = record var C:newint; proc P = (C:=@A+1) end;
var F:R; var G:R
in ... A...F.C ... call F.P ... G.C ... call G.P
```

- unique storage cells are allocated for  $A$ ,  $F.C$  and  $G.C$ . Procedure calls then update those storage locations
- we need to introduce a new syntax domain of identifier expressions to access "fields" of a record and modify all invocation constructs

$X \in \text{Identifier-expr}$

```
C ::= L := E | C1;C2 | ... | call X
E ::= N | @L | ... | X
L ::= X
X ::= I | X.I
```

3/29/04

23

## Typing Rules for Type-structures

$\pi \vdash \text{newint: intloc class}$

$$\frac{\pi \vdash D: \pi_1 \text{dec}}{\pi \vdash \text{record } D \text{ end: } \pi_1 \text{ class}}$$

$$\frac{\pi \vdash T: \delta \text{dec}}{\pi \vdash \text{var } I: T : \{I: \delta\} \text{dec}}$$

$$\frac{\pi \vdash T: \delta \text{class}}{\pi \vdash \text{class } I = T : \{I: \delta \text{class}\} \text{dec}}$$

$\pi \vdash I: \theta$ , if  $(I: \theta) \in \pi$

$$\frac{\pi \vdash X: \pi_1}{\pi \vdash X.I: \theta} \text{ if } (I: \theta) \in \pi_1$$

3/29/04

24

## Declaration Abstractions

- We can also define an abstraction over declarations using a “module”
    - a module declaration binds an identifier to a record of bindings; storage is not allocated for variables in a module until the module is imported
    - declarations in a module are not visible until imported
      - similar to a package in Java or namespace in C++
- ```
D ::= ... | module I = {D} | import I
module M = { class K=newint; var A:K; proc P=A:=0 };
var B:newint;
import M
in call P; B := @A
```

3/29/04

25

Type Rules and Semantics

$$\frac{\pi \mid\!-\! D : \pi_1 \text{ dec}}{\pi \mid\!-\! \mathbf{module} \ I = \{D\} : \{I : \pi_1 \text{ dec}\} \text{ dec}} \quad \pi \mid\!-\! \mathbf{import} \ I : \pi_1 \text{ dec}, \text{ if } (I : \pi_1 \text{ dec}) \varepsilon \pi$$

$$[[\pi \mid\!-\! \mathbf{module} \ I = \{D\} : \{I : \pi_1 \text{ dec}\} \text{ dec}]]e \ s = (\{I = f\}, s)$$

where $f(s') = [[\pi \mid\!-\! D : \pi_1 \text{ dec}]]e \ s'$

$$[[\pi \mid\!-\! \mathbf{import} \ I : \pi_1 \text{ dec}]]e \ s = f(s) \text{ where } (I = f) \varepsilon e$$

3/29/04

26

Language Summary

- See section 2.15 for the complete language definition obtained from applying the abstraction principle.
 - note that locations have been dropped and replaced by variables and identifier expressions

Chapter 2 Homework

- Exercises 1.1, 2.1(a), 2.2, 5.1, 7.2(a), 10.2, 14.1(a), 14.1(c)
- READ CHAPTER 3