

**The Structure of Typed Programming Languages**  
**(Chapter 2)**  
 CS386L- Programming Languages  
 Dr. Greg Lavender  
 Department of Computer Sciences  
 The University of Texas at Austin

CS386L: Lecture-07 - The Structure of Typed Programming Languages: Chapter 2.

## The Abstraction Principle

- The role of names in programming
  - names are not essential, but they sure are convenient
- Abstraction Principle
  - "the phrases of any semantically meaningful syntactic class may be named."
  - binding of names to things is a useful abstraction
    - e.g., we should be able to bind names to locations
- a named value is called an abstraction, which is created using definition
  - For example: define I = V
  - associates the **name** I (some identifier) with a value V
  - we call V the **body** in this case
  - the **define** command creates a **binding** of the name to the body
  - an abstraction is invoked by using its name: I, invoke I, call I
    - e.g., proc I = C .... call I

3/8/04
2

## What things should be named?

- Location: alias
- Numeral: const
- Expression: function
- Command: procedure
- Declaration: module
- Type structure: class or type

## Expression Abstractions

- I.e., functions
- Extend our core imperative language with new syntax domains and grammar rules:
  - D is a Declaration
    - $D ::= \text{fun } I = E \mid D_1, D_2 \mid D_1 : D_2$
    - notice that functions are not yet allowed parameters -- they are just names for expressions
  - I is an Identifier
    - $E ::= N \mid @L \mid E_1 + E_2 \mid E_1 = E_2 \mid \sim E \mid I$
    - we extend our expression grammar rule to allow functions to be used in place of expressions
  - Programs
    - programs allow us to define a list of (1 or more) declarations that apply over a command or command sequence
    - $P ::= D \text{ in } C$ 
      - notice that this is like a let expression in ML
        - let fun f x = x \* x in f(2) end;
- C, L and N are as before

## Function expressions

- **Function invocations**
  - called by name,  $I$
  - can appear in place of an expression
    - referentially transparent
    - $\text{fun } A = 1 + @loc_1$
    - $loc_1 = A+2$  is then legitimate as an expression
    - it is the same as  $(1+@loc_1)+2$
  - type structure is now a bit more complicated
    - each function must inherit the type of its corresponding expression when it is defined
    - the type of the function has to be communicated to any commands and expressions that mention the name of the function

3/8/04

5

## Type Assignment

- A type assignment is a set of "identifier:type" attribute pairs
  - $\text{fun } A = 1 + @loc_1$
  - $\text{fun } B = @loc_1 = 0$
  - has the type assignment
    - $\pi = \{A:\text{intexp}, B:\text{boolexp}\}$  *dec*
    - since the RHS of the definition of  $A$  is an integer expression and the RHS of  $B$  is a boolean expression
  - for any identifier  $I$  there is at most one pair  $(I:\theta)$  in the type assignment  $\pi$ 
    - this matches intuition, which says that a type for an identifier must be unique
      - $\text{fun } A = 1, \text{fun } A = 1=2$  is illegal since  $A$  is defined twice with two different typings
    - in a program  $P$ , we must ensure that each function is well-typed and that we union all the typings into a type assignment structure that can be used over commands

3/8/04

6

## Type Assignment Transmission

- consider the program P where
  - fun A=1, fun B=@loc<sub>1</sub> = 0  
in while B do loc<sub>1</sub> := A + 2 od
  - what is the type of A?
  - what is the type of B?
  - what is the type assignment for the while command?
    - $\pi = \{A:\text{intexp}, B:\text{boolexp}\}$
    - $\pi \vdash \text{loc}_1 := A+2:\text{comm}$
    - see figure 2.2 on page 33 for the annotated syntax tree of the program P
      - note that there are two branches, one for the declarations and one for the command
      - the type assignment is constructed as the union of the declarations in the declaration subtree and then transmitted to each node in the command subtree

3/8/04

7

## New Type Rules

- if E is a well-typed expression with type  $\tau_{\text{exp}}$ , then fun I = E is well-typed declaration with type  $\{I:\tau_{\text{exp}}\}\text{dec}$ .
- $\pi$  represents the set of global declarations visible at the point of use

$$\frac{\pi \vdash E: \tau_{\text{exp}}}{\pi \vdash \text{fun } I=E:\{I:\tau_{\text{exp}}\}\text{dec}}$$

- Function invocation I is well-typed if I is bound in the type assignment
  - this rule is the reason for the type assignment

$$\pi \vdash I: \tau_{\text{exp}} \text{ if } (I:\tau_{\text{exp}}) \text{ in } \pi$$

3/8/04

8

## New Type Rules

- Declarations are combined by  $D_1, D_2$ 
  - function declarations in  $D_1$  cannot collide with declarations in  $D_2$ , so identifiers in  $\pi_1$  must be distinct from those used in  $\pi_2$
  - bindings in  $D_1$  not visible to bindings in  $D_2$ 
    - Like let in Scheme: (let (( $I_1 V_1$ ) ( $I_2 V_2$ )...) body)
      - where the binding of  $I_1$  to  $V_1$  is not visible when evaluating of the binding of  $I_2$  to  $V_2$

$$\frac{\pi \vdash D_1 : \pi_1 \text{dec} \quad \pi \vdash D_2 : \pi_2 \text{dec}}{\pi \vdash D_1, D_2 : (\pi_1 \text{U}^* \pi_2) \text{dec}}$$

where  $\pi_1 \text{U}^* \pi_2 = \pi_1 \text{U} \pi_2$   
 if  $\{I \mid (I:\theta_1) \in \pi_1\} \text{intersect} \{I \mid (I:\theta_2) \in \pi_2\} = \text{Empty}$   
 otherwise it is undefined

3/8/04

9

## New Type Rules

- Sequencing of declarations  $D_1; D_2$ 
  - bindings of  $D_1$  are visible to bindings in  $D_2$
  - like let\* in Scheme
    - ( $\text{let}^* ((I_1 V_1) (I_2 V_2) \dots) \text{body}$ )
      - binding of  $I_1$  is visible to evaluation of the binding of  $I_2$
  - e.g., fun A = @loc<sub>1</sub>; fun B = A + 1

$$\frac{\pi \vdash D_1 : \pi_1 \text{dec} \quad \pi \text{U}^* \pi_1 \vdash D_2 : \pi_2 \text{dec}}{\pi \vdash D_1; D_2 : (\pi_1 \text{U}^* \pi_2) \text{dec}}$$

$$\pi \vdash D_1; D_2 : (\pi_1 \text{U}^* \pi_2) \text{dec}$$

3/8/04

10

## New Type Rules

- Once all declarations are defined, the resulting type assignment is transmitted to the command part of a program
  - $\pi_{dec}$  represents the type assignment obtained from the D-tree, which starts with an empty type assignment Empty
  - The C-tree gets  $\pi$  as its type assignment

$$\frac{\text{Empty} \vdash D: \pi_{dec} \quad \pi \vdash C: \text{comm}}{\vdash D \text{ in } C : \text{comm}}$$

## New Type Rules

- Revised type rule for commands and expressions
  - For example, for assignment, we start with type assignment  $\pi$  and pass it to the L and E subtree.

$$\frac{\pi \vdash L: \text{intloc} \quad \pi \vdash E: \text{intexp}}{\pi \vdash L := E : \text{comm}}$$

## Inherited vs Synthesized Attributes

- type assignments are inherited attributes
  - passed down from the top of the C-tree
- type attributes are synthesized attributes
  - passed up from the leaves of the D-tree

## Semantics of Abstraction

- In order to deal with declarations, we need to define a new structure called an **environment**
- the environment is like the store in that it must be present in the semantics for expressions and commands
  - it is a set of identifier-meaning pairs which resolves identifier references
- the meaning of a command is changed s.th.
  - $[[\pi \mid - C: \text{comm}]]$  is a function from  $\text{Env} \rightarrow \text{Store} \rightarrow \text{Store}$ , for example:
    - $[[\pi \mid - L := E: \text{comm}]]e \ s = \text{update}([[ \pi \mid - L: \text{intloc} ]]e, [[ \pi \mid - E: \text{intexp} ]]e \ s, s)$

## Semantics of Abstraction

- similarly the semantics of expressions is now based on the values of the environment and the store
  - $[[\pi \mid - E:\tau\text{exp}]]$  is a function from  $\text{Env} \rightarrow \text{Store} \rightarrow [[\tau]$
  - $[[\pi \mid - @L:\text{intexp}]]e s = \text{lookup}([[ \pi \mid - L:\text{intloc}]]e, s)$
  - $[[\pi \mid - I:\tau\text{exp}]]e s = v(s)$ , where  $(I=v)$  in  $e$ 
    - this says that function invocation causes a lookup in the environment structure. If  $I$  is bound to  $v$  in  $e$ , then  $v$  is used as the meaning of the invocation

## Semantics of Function Definitions

- The meaning of a function definition is the binding of the functions name to the meaning of its body
  - $[[\pi \mid - \text{fun } I = E:\{I:\tau\text{exp}\}\text{dec}]]e s = \{I = f\}$ 
    - where  $f(s') = [[\pi \mid - E:\tau\text{exp}]]s'$
  - this just says that  $I$  is bound to the meaning of  $E$ , creating an environment containing the binding
  - $E$  is *not* evaluated using  $s$  in a definition
    - $E$  will be evaluated using the store when  $I$  is invoked

## Semantics of Declarations

- the semantics of a compound declaration is the union of the environments
  - $[[\pi \mid - D_1; D_2; (\pi_1 \text{ U* } \pi_2) \text{ dec}]] e s = e_1 \text{ U } e_2$   
 where  $e_1 = [[\pi \mid - D_1; \pi_1 \text{ dec}]] e s$  and  
 $e_2 = [[\pi \text{ U* } \pi_1 \mid - D_2; \pi_2 \text{ dec}]] (e \text{ U } e_1) s$
  - notice that the environment used by  $D_2$  contains the bindings made by  $D_1$
  - the meaning of a declaration is a function that maps the current environment and store into a new environment that has the bindings made by the declaration

## Semantics of Programs

- The environment built up by the declarations is supplied to the command part of the program
  - $[[ \mid - D \text{ in } C; \text{ comm} ]] s = [[\pi \mid - C; \text{ comm}]] e_0 s$   
 where  $e_0 = [[\text{Empty} \mid - D; \pi \text{ dec}]] \text{Empty } s$
- See figure 2.4 for the complete revised semantics for the core imperative language up to this point