

The Structure of Typed Programming Languages
(Chapter 3)
 CS386L- Programming Languages
 Dr. Greg Lavender
 Department of Computer Sciences
 The University of Texas at Austin

CS386L: Lecture-10 - The Structure of Typed Programming Languages: Chapter 3

The Parameterization Principle

- phrases from any semantically meaningful syntactic class may be parameters
 - numerals, expressions, locations, declarations, commands, type-structures, etc., can all be used as parameters
- not all languages allow such a complete set of parameters
 - what restrictions do C, C++ and Java impose?
 - how would you write a swap procedure in each?
 - you can do this in C by passing a pointer to an int
 - `void swap (int* a, int* b) { int t = *a; *a = *b; *b = t; }`
 - you can do this in C++ by passing a pointer/reference to an int
 - `void swap (int& a, int& b) { int t = a; a = b; b = t; }`
 - how do you write a procedure in Java to swap two integer values?
 - `public void swap(?) { ?? }`

4/5/04
2

Abstractions as Parameters

- For now, we only define abstractions
 - use single parameters (parameter lists come later)
 - define $I_1(I_2:\theta) = V$
 - invoke $I_1(U)$
 - we call I_2 the formal parameter of type θ
 - we call U the actual parameter (i.e., the argument value), which must also be of type θ
 - note that I_1 is a function type mapping a value of type θ_1 to a value of type θ_2 ($V:\theta_2$)

4/5/04

3

Typing Rules

- typing rules for a V-abstraction parameterized by a U-phrase
 - $\pi_1 \cup \pi_2 = \pi_2 \cup (\pi_1 - \{(I:v) \mid (I:w) \in \pi_2 \text{ and } (I:v) \in \pi_1\})$
 - the body V uses a type assignment that contains all global bindings plus the formal parameter and its type. However, the formal parameter "cancels out" any global binding using the same name.

$$\frac{\pi \cup \{I_2:\theta_1\} \vdash V:\theta_2}{\pi \vdash \text{define } I_1(I_2:\theta_1) = V: \{I_1:\theta_1 \rightarrow \theta_2\}\text{dec}}$$

$$\frac{\pi \vdash U:\theta_1}{\pi \vdash \text{invoke } I(U):\theta_2} \quad \text{if } (I:\theta_1 \rightarrow \theta_2) \in \pi \quad \pi \vdash U:\theta, \text{ if } (I:\theta) \in \pi$$

4/5/04

4

Parameter Name Binding

- The canceling out of the parameter bindings in the global type assignment is intuitively what one would expect
 - the names visible to a procedure or function should be those that are in the global scope, overridden by any "local" names
 - local names include parameters
 - and as we will see in Chapter 4, local variable names

4/5/04

5

Parameter Name Binding

- example:


```
var x:newint; const A = 1;
proc P(A:boolexp) = if A then x := 0 else skip fi
```

 - the name A is in conflict.
- The body of P is governed by the type assignment
 - $\pi = \{x:\text{intloc}, A:\text{int}\} \cup \{A:\text{boolexp}\}$
 - $= \{x:\text{intloc}, A:\text{boolexp}\}$
- The formal parameter name is used in the body the same way an abstraction is invoked --- by just mentioning the name
 - call P and P are the same when P is provided as a parameter
 - so parameter reference and abstraction invocation are semantically the same
 - this is called the correspondence principle

4/5/04

6

Extended Core Language

- Extend the core imperative language with parameter forms for procedures, functions, modules, classes, etc.
- for example, the command abstraction parameter form is as follows:

$$\frac{\pi \vdash U : \{I_2; \theta_1\} \quad \vdash C : \text{comm}}{\pi \vdash \text{proc } I_1(I_2; \theta_1) = C : \{I_1; \theta_1 \rightarrow \text{comm}\} \text{dec}}$$

$$\frac{\pi \vdash U : \theta \quad \text{if } (I; \theta_1 \rightarrow \text{comm}) \in \pi}{\pi \vdash \text{call } I(U) : \text{comm}}$$

4/5/04

7

Expression Parameters

- Extended syntax

$D ::= \dots \mid \text{proc } I_1(I_2; \tau \text{exp}) = C, \text{ where } \tau \in \{\text{int}, \text{bool}\}$
 $C ::= \dots \mid \text{call } I(E)$
 $E ::= \dots \mid I$

- Example:


```
var A:newint;
proc P (M:intexp) = A:=M in call P(@A)
```
- Note the call $P(0=1)$ is illegal, since it is a boolexp, not an intexp
- What if we had this instead? Does this work?


```
var A:newint;
proc P(M:intloc) = M := @M+1 in call P (A)
```
- is this pass-by-value or pass-by-reference?
 - value parameters are members of the expression domain
 - reference parameters are members of the variable domain

4/5/04

8

Actual Parameter Evaluation

- What evaluation strategy do we take with expression parameters?
 - eager - evaluate them immediately before invocation, i.e., call-by-value
 - lazy - delay evaluation until we are ready to use the parameter, i.e., call-by-name
- example


```
var A:newint; proc P(M:intexp) = A :=M; A:=M
in A := 1; call P (@A+1)
```

 - what happens if we eagerly evaluate A? Lazy?
 - the copy rule we saw before is used to implement lazy evaluation
 - eager evaluation however is more efficient and more predictable

4/5/04

9

Semantics of Parameter Transmission

- Similar to semantics of expression abstractions
- let's consider just lazily evaluated procedure abstractions for now
 - the meaning of a parameterized procedure is a function that requires an environment, the value of the actual parameter, and the store so that it can compute a new store

$$[[\pi \mid \text{proc } I_1(I_2:\tau\text{exp}) = C: \{I_1:\tau\text{exp} \rightarrow \text{comm}\}\text{dec}]]e \text{ s} = (\{I_1=p\}, s)$$

where $p \vee s' = [[p \cup \{I_2:\tau\text{exp}\} \mid C:\text{Comm}]](e \cup \{I_2=v\})s'$

$$[[\pi \mid \text{call } I_1(E):\text{comm}]]e \text{ s} = p([[\pi \mid E:\tau\text{exp}]]e \text{ s}), \text{ where } (I_1=p) \in e$$

4/5/04

10

Semantics

- The environment for the procedure is modified to contain the binding (I=u) by removing from the environment any binding (I=v)
- In the invocation, the parameter E is evaluated with the environment and store
- A parameter reference is just like the semantics of an eagerly evaluated abstraction invocation

$$[[\pi \mid \text{proc } I_1(I_2:\tau\text{exp}) = C: \{I_1:\tau\text{exp} \rightarrow \text{comm}\}\text{dec}]]e \text{ s} = (\{I_1=p\}, s)$$

where $p \text{ v } s' = [[p \cup \{I_2:\tau\text{exp}\} \mid C:\text{Comm}]] (e \cup \{I_2=v\})s'$

$$[[\pi \mid \text{call } I_1(E):\text{comm}]]e \text{ s} = p([[\pi \mid E:\tau\text{exp}]]e \text{ s}) \text{ s}, \text{ where } (I_1=p) \in e$$

$$[[\pi \mid I_2:\tau\text{exp}]]e \text{ s} = v, \text{ where } (I_2 = v) \in e$$

Other Varieties of Parameters

- Numeral and command parameters
 - command parameters use lazy evaluation
- Example:

```

const K=2; var A:newint
proc P(M:comm) = A := K+1; M;
proc Q(X:int) = A := X;
in
  call P (Call Q(2));
  call P (call P(call Q(K)));
  call P(A:=K)

```

Declaration Parameters

- Used by modules to build bigger modules


```

module COUNTER (M: {X:intloc}, INIT:comm)dec) =
{ import M; proc P = (call INIT; X := @X+1)};
module N = { var X: newint; proc INIT = X:=0;};
import COUNTER (import N) in ...
      
```

Type Structure Parameters

- Build different versions of classes


```

class STACK1 = record var A:newint; proc PUSH = A:=0 end;
class STACK2 = record var A:newint; proc PUSH = A:=1 end;
class K = (T:{A:intloc,PUSH:comm}class) =
  record var S:T; proc PUSH2 = call S.PUSH; call S.PUSH end;
  var X:K(STACK1); var Y:K(STACK2)
      
```
- The type-structure parameter makes K "generic" in the data structure that it handles, but the type information means that the parameter is type checked at compile time
 - like templates in C++ or generics in Java 1.5

Homework

- Chapter 3 Exercises:
 - 1.3a, 2.2, 3.1, 3.2, 3.4, 4.1, 7.2a-c, 8.2a-b
- Read Chapter 4