Comp 411
Principles of Programming Languages
Lecture 4
The Scope of Variables

Corky Cartwright

January 22, 2014

Variables

- What is a variable?
  A legal symbol without a pre-defined (reserved) meaning that can be bound to a value (and perhaps rebound to a different value) during program execution.
  - Examples in Scheme/Java

    ```
    x  y  z
    ```

  - Non-examples in Java

    ```
    +  null true false 7f throw new if else
    ```

  - Complication in Java: variables vs. fields
- What happens when the same name is used for more than one variable?
  - Example in Scheme:

    ```
    (lambda (x) (x (lambda (x) x)))
    ```

  We use *scoping* rules to distinguish them.

Some scoping examples

- Java:

```
class Foo {
  static void doNothing() {
    int[] a = ...;
    for (int i = 0; i < a.length; i++) { ... }
    ...
//   <is a in scope here?  is i in scope here?>
    ...
  }
}
```

What is the scope (part of the program where it can be accessed/referenced) of `a`?

What is the scope of `i`?

Formalizing Scope

- Let us focus on a pedagogic functional language that we will call LC. LC (based on the Lambda Calculus) is the language generated by the root symbol **Exp** in the following grammar

  Exp ::= Num | Var | (Exp Exp) | (lambda Var Exp) |
  (+ Exp Exp)

  where **Var** is the set of alphanumeric identifiers excluding **lambda** and **Num** is the set of integers written in conventional decimal radix notation. (LC is very *restrictive*; there are no operators on integers other than **+**. Later in the course, we will slightly expand it.)

- If we interpret LC as a sub-language of Scheme, it contains only one binding construct: lambda-expressions. In

  ```
  (lambda (a-var) an-exp)
  ```

  a-var is introduced as a new, unique variable whose scope is the body **an-exp** of the lambda-expression (with the exception of possible "holes", which we describe in a moment).

Abstract Syntax of LC

- Recall that

  Exp ::= Num | Var | (Exp Exp) | (lambda Var Exp) | (+ Exp Exp)

  where

  Num is the set of numeric constants (given in a lexer spec)

  Var is the set of variable names (given in a lexer spec)

- To represent this syntax as trees (abstract syntax) in Scheme, we define

```
; exp := (make-num number) + (make-var symbol) + (make-app exp exp) +
;          (make-proc symbol exp) + (make-add exp exp)
(define-struct (num n))          ;; n is a Scheme number
(define-struct (var s))          ;; s is a Scheme symbol
(define-struct (app rator rand))
(define-struct (proc param body))  ;; param is a symbol not a var!
(define-struct (add left right))
```

  where

  app represents a function application

  proc represents a function definition (**lambda** expression)

Free and Bound Occurrences

- An important building block in characterizing the scope of variables is defining when a variable *x* ***occurs free in*** an expression. For LC, this notion is easy to define inductively.
- Definition (Free occurrence of a variable in LC):
  Let ***x***, ***y*** range over the elements of Var. Let ***M***, ***N*** range over the elements of **`Exp`**. Then ***x occurs free in***:
  - ***y*** if ***x*** = *y*;
  - **`(lambda (y) M)`** if ***x*** != ***y*** and ***x*** occurs free in ***M***
  - **`(M N)`** if it occurs free either in ***M*** or in ***N***.

  The relation ``***x occurs free in*** *y*'' is the least relation on LC expressions satisfying the preceding constraints.
- Note that the variable name enclosed in parentheses following a **`lambda`** is not considered a conventional "occurrence" of the variable and is not classified as either *free* or *not free*. It is sometimes called a *binding occurrence* of a variable.
- It is straightforward but tedious to define when a particular *occurrence* (excluding binding occurences) of a variable ***x*** (identified by a path of tree selectors) is *free* or *noi free*; the definition proceeds along similar lines to the definition of *occurs free* given above.
- Definition: an *occurrence* of ***x*** is *bound* in ***M*** iff it is **not** free in ***M***.

Static Distance Representation

- The choice of bound variable names in an expression is arbitrary (modulo ensuring distinct, potentially conflicting variables have distinct names).
- We can eliminate explicit variable names by using the notion of "relative addressing" (widely used in machine language and assembly language): a variable reference simply identifies which lambda abstraction introduces the variable to which it refers. We can number the lambda abstractions enclosing a variable occurrence ***1***, ***2***, ... (from the inside out) and simply use these indices instead of variable names. Since LC includes integer constants, we will italicize the indices referring to variables to distinguish them from integer constants.
- These indices are often called *deBruijn indices*
- Examples:
  - `(lambda (x) x)` ➔ `(lambda 1)`
  - `(lambda (x) (lambda (y) (lambda (z) ((x z)(y z)))))`

    ➔ `(lambda (lambda (lambda ((3 1)(2 1)))))`

Generalized Static Distance

- In LC, `lambda` abstractions are unary; only one variable appears in the parameter list.
- In practical programming languages, parameter lists can contain any finite number (within reason) of parameters.
- How can we generalize deBruijn notation to accommodate lambda abstractions of arbitrary arity?
- Hint: does a variable reference have to be a scalar, physics terminology for a simple number? Lists are not scalars.

Generalized SD Example

```
(lambda (x y) (lambda (z) ((x z)(y z))))
→ (lambda
    (lambda (([2 1] [1 1])([2 2] [1 1]))))
```

Note that we are indexing the variables within a given parameter list starting at ***1***, not ***0***. In the context of intermediate represenations used for compilation, indexing typically starts at ***0*** because the corresponding addressing arithmetic uses an offset of ***0***.