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

Corky Cartwright

January 29, 2021

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 sampleMethod() {
    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 Using **LC**

- 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 abstractions. In **`(lambda (a-var) an-exp)`** Scheme encloses the parameter list in parentheses but otherwise conforms to the syntax we have used for LC.

- **`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)

  add represents an application of addition to two arguments

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 no variable `x` *occurs free in* a number.
- The variable name immediately following (“introduced by”) a `lambda` is not considered a conventional “occurrence” of the variable and is not classified as either *free* or *not free*. It is usually called a *binding occurrence* of a variable.
- It is straightforward but tedious to define when a particular *occurrence* (excluding binding occurrences) of a variable `x` (identified by a path of tree selectors) is *free* or *not 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` and it is not a *binding occurrence* (which is neither bound nor free).

Examples of Free and Bound Occurrences

- Consider the LC expression ***M*** = **`(lambda y (y x))`**.
- The first occurrence of ***y*** in ***M*** is a binding occurrence.
- The second occurrence of ***y*** in ***M*** is bound by the binding occurrence.
- The variable **`x`** occurs free in ***M*** ; the only occurrence of **`x`** in ***M*** is free.
- The variable **`y`** occurs free in **`(y x)`**; so does the variable **`x`**.

Nested Scope

- A lambda-expression of the form `(lambda Var Exp)` is called a *lambda abstraction*.
  - The expression `Exp` forming the body of a lambda abstraction can contain lambda abstractions. For example, the lambda abstraction `(lambda y (lambda x y))` defines a function that takes an input `y` and returns the constant function that always returns `y`.
  - The inner lambda abstraction `(lambda x y)` introduces a binding occurrence of the variable `x`. In LC, the scope a variable introduced in a lambda abstraction is simply the *body* of the lambda abstraction. The choice of the variable name `x` is *almost* arbitrary. We could use `z` or `v` instead. Of course, we would have to change the name of all free occurrences of `x` in the body to the new variable name. Nevertheless, we could use any variable name instead of `x` *except* `y`. Why? If we use `y` as the name of the variable introduced by the inner lambda abstraction, we would *shadow* the variable of the same name introduced by the outer lambda abstraction. No matter what name we choose for the variable introduced by the inner lambda abstraction, that variable hides any variable with the *same* name in an enclosing lambda abstraction.

Nested Scope cont.

- At any point in an LC program, a finite collection of variables—introduced in enclosing lambda abstractions—is visible. This collection is always finite because all programs are finite in size.
- If we try to access a variable that has not been introduced in an enclosing lambda abstraction, then the attempted access will generate a runtime error. It is easy to detect such references because they are precisely the free variables of the whole program.
- Think about how you could write a Java program (which could be purely functional [no mutation of fields or variables]) to return a list of the variables that appear free in an LC program fed as input to the Java program. If we control the content of an entire LC program, we can make sure that all variable names are unique, avoiding all shadowing.
- In practice, we typically do not have control over all of the code in a program, particularly code that may be revised in the future, so shadowing happens.

Static Distance Representation

- The choice of variable names introduced in a lambda expression is arbitrary (modulo ensuring distinct, potentially conflicting variables have distinct names).
- We can completely 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*.
- The numbering of deBruijn indices may start at 0 instead of 1; it a design choice in defining a deBruijn notation system.
- 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 simple scalar (physics terminology)? Lists and vectors 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 representations used for compilation, indexing typically starts at ***0*** (because the corresponding addressing arithmetic uses an offset of ***0***).