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

Corky Cartwright

September 5, 2012

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 Scala:

    ```
    (x: Any=>Any) => x ((x:Any) => x)
    ```

  We use *scoping* rules to distinguish them.

Some Scoping Examples

- Imperative Scala (similar to Java)

```
object Foo {
  def void doNothing {
  var Array[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 | App(Exp, Exp) | Lambda(Var, Exp) | Sum(Exp, Exp)

  where **Var** is the set of alphanumeric identifiers excluding **Lambda** and **App**; 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.)

Abstract Syntax of LC

- Recall that

  Exp ::= Num | Var | App(Exp, Exp) | Lambda(Var, Exp) | Sum(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 Scala, we define

```
trait Exp
case class Num(n: Int) extends Exp
case class Var(s: Symbol) extends Exp
case class App(rator: Exp, rand: Exp)
case class Lambda(s: Symbol, e: Exp)   // s is a Symbol, not a Var
case class Sum(left: Exp, right: Exp)
```

  Where

  App represents a function application

  Lambda represents a function definition (**λ**-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` (`x`, `y` are the same variable)
  - `(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.
- It is straightforward but tedious to define when a particular *occurrence* of a variable `x` (identified by a path of tree selectors) is *free* or *bound*; 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 has to identify which lambda abstraction introduces the variables 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 embolden 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, App(App('x,'z),App('y,'z)))))
 → Lambda(Lambda(Lambda(App(App(3,1),App(2,1)))))
```

- Note: in Scala, we must use wrapped **`Int`**s as *deBruijn indices,* but we can use an **`implicit`** delcaration to abbreviate wrapped **`Int`**s as ordinary **`Int`**s.

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)?

Generalized Static Distance

- General solution: use a pair of integers `[i,j]` to designate which variable is referenced:
  - a frame index `i` indicating which enclosing binding form (e.g., `Lambda`) is referenced
  - an offset `j` indicating which variable within the list of bindings for that form is referenced.
- In the GSD representation, each binding form must specify how many variables it introduces.
- Examples
  - `Lambda(('x,'y), Sum('x,'y))` → `Lambda(2, Sum([1,1], [1,2]))`
  - `Lambda(('x,'y), App(Lambda('z, 'x), 'y))` →
    `Lambda(2, App(Lambda(1, [2,1]), [1,2])`
- Computer scientists often choose to start counting at 0 instead of 1.