Comp 411
Principles of Programming Languages
Lecture 5
Syntactic Interpreters

Corky Cartwright

September 9, 2012

A Syntactic Interpreter for LC

- Recall the context-free language LC from last lecture:

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

  A *proper* LC program is an LC expression M that is *closed*, *i.e.*, contains no *free* variables. An LC program is any LC expression.

- Recall the abstract syntax for LC in Scheme given in the last lecture:

```
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)

Lambda-Calculus Notation

The mathematical literature on the pure lambda calculus uses a leaner (perhaps more human friendly) notation for lambda-expressions. The syntax is:

```
M ::= Var | (M M) | (λ Var . M)
```

LC adds clauses for integer constants (`Num`) and applications of a binary addition operator (`Sum`).

In applications and abstractions, parentheses may be elided if no ambiguity results (assuming left associativity and maximal expression extent—as in parsing JAM). When parentheses are elided in applications, application associates to the *left, i.e.,*

`λx . x y` abbreviates `(λx . (x y))`

`L M N` abbreviates `((L M) N)`

The pure lambda-calculus lacks primitive data values (constants). To model programming languages we often add constants like numbers and the addition function (as in LC).

Why is this notation more compact (and readable) that abstract syntax? Application is implicit; parentheses are elided; `λ` instead of `Lambda`; `x` instead of `'x` or `Var('x)` depending on context.

Syntactic Interpretation

What does syntactic interpretation do?

- Reduce the AST for a complete program to a *value*.
- What is a value? A special AST representing a data constant. In LC (a subset of Scheme), a value V is either a number or a procedure:

  V ::= Int | Lambda(s, M)

- What are the syntactic evaluation rules for LC?

A Syntactic Interpreter for LC

Basic Rules of Evaluation

- Rule 1: For applications of the binary operation **`Sum`** to two arguments that are numeric values, replace the application by the sum of the two arguments.
- Rule 2: For applications of a lambda-expression to a value, substitute the argument for the parameter in the body, *i.e.*,

  `(λx.M) V` → **`M[x := V]`** (in λ-calculus notation)

  or

  `((λApp(Lambda(x, M), V)` → **`M[x := V]`** (in Scala notation)

  where **`M[x := V]`** means **`M`** with all *free* occurrences of **`x`** replaced by **`V`**. This rule is called *beta-value reduction*.
- Observation: the definition of *value* has a major impact on evaluation
- What happens if we define

  `V ::= n | Lambda(x, V)`
- Some evaluation strategies for the untyped lambda-calculus do this, but they have not proven relevant to defining the semantics of real programming languages. Why? What "goes wrong"? Consider

  `λy.(λx.xx)(λx.xx)`
- What if we allow arguments in procedure application reductions that aren't values.
  - Example:

    `(λy.5)((λx.xx)(λx.xx))`
  - This is a sensible choice in *functional* languages that prohibit side effects (the values of bound variables and fields never change). Haskell does this. This convention is often called *call-by-name*.

Syntactic Interpreter for LC cont.

Combining evaluation rules:

- Given an LC expression, we evaluate it by repeatedly applying the preceding rewrite rules until we get an answer.
- What happens when we encounter an expression to which more than one rule applies? In most real-world languages and in LC, the leftmost rule always takes priority.
- Other strategies are possible. Some "syntactic" (rewrite-rule-based) semantics for complex languages define formal syntactic rules (called evaluation contexts) to determine which reduction is done first.

Gotcha's in Syntactic Semantics

- In Rule 2 (called "beta-reduction" in the λ-calculus and program semantics literature), we confined substitution in the definition of `M[x := V]` to the *free* occurrences of `x` in `M`.
- If we had not confined substitution to *free* occurrences, the rule would have produced strange results, destroying the meaning of bound variables in `M`.
- In some unusual corner cases during program evaluation (and common cases when we use Rule 2 to transform programs replacing "equals by equals"), we must be particularly careful in how we perform the substitution of `V` for `x` in `M`. If not, free variables in `V` can be "captured" in replaced occurrences of `x` in `M`. Consider the following example:

```
Lambda('y, Var('x)) [Var('x) := Var('y)]
```

If we replace all free occurrences of `Var('x)` by `Var('y)`, we get

```
Lambda('y, Var('y))
```

which is wrong! The occurrence of `Var('y)` in `[Var('x) := Var('y)]` is *free*. Presumably, it is bound somewhere in the context surrounding

```
Lambda('y, Var('x))
```

If we substitute `Var('y)` for the free occurrence of `Var('x)` in `Lambda('y, Var('x))`, it becomes bound by a local definition of `'y`. The *solution* is to rename the variable introduced in the local definition of `'y` as a fresh variable name, say `'`

```
Lambda('y, Var('x)) [Var('x) := Var('y)] → Lambda('z, Var('z))
```

Safe Substitution

This revised substitution process (renaming local variables that would otherwise capture free occurrences in the expression being substituted) is called *safe substitution*. The results produced by safe substitution are *non-deterministic* in a trivial sense because the choices for the new names of renamed local variables are arbitrary (as long as they are *fresh, i.e. distinct from existing variables in the program text involved in the substitution)*. Hence,

```
Lambda(y, x) [x := y]  →  Lambda(z, y)
                       →  Lambda(u, y)
                       →  Lambda(v, y)
                       →  …
```

Of course, this non-determinism goes away if we used deBruijn notation instead of conventional notation. But the corresponding definition of substitution changes (also and no renaming occurs (it can't!). Note: deBruijn notation can only express substitutions within proper programs; it does not provide a way to name variables that are free at top-level.

α-Conversion (Variable Renaming)

The renaming of bound variables is an axiom (assumed equation) of the λ-calculus called α-conversion:

λx.M → λy.M[x:=y] (α-conversion)

where **y** does not occur in **M**. Hence, α-conversion does not presume any notion of safe substitution as primitive. Safe substitution can be defined using α-conversion.

α-conversion and β-reduction are the fundamental equations (laws) of the λ-calculus. Mathematicians often do not directly define safe substitution. They simply stipulate that β-reduction is not allowed to capture bound variables. Hence, an α-conversion may sometimes be necessary prior to a β-reduction. α-conversions are typically assumed with little or no comment. Beware.